

Health Informatics

Series Editors: Kathryn J. Hannah · Marion J. Ball

Prakash M. Nadkarni

Metadata-driven Software Systems in Biomedicine

Designing Systems that can adapt
to Changing Knowledge

 Springer

Prakash M. Nadkarni

Kathryn J. Hannah • Marion J. Ball
(Series Editors)

Metadata-driven Software Systems in Biomedicine

**Designing Systems that can adapt
to Changing Knowledge**



Springer

Author

Dr. Prakash M. Nadkarni, M.D.
Yale University
School of Medicine
Cedar St. 333
06510 New Haven Connecticut
USA

ISBN 978-0-85729-509-5 e-ISBN 978-0-85729-510-1
DOI 10.1007/978-0-85729-510-1
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2011928228

© Springer-Verlag London Limited 2011

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

Product liability: The publisher can give no guarantee for information about drug dosage and application thereof contained in this book. In every individual case the respective user must check its accuracy by consulting other pharmaceutical literature.

Cover design: eStudioCalamar, Figueres/Berlin

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Metadata-driven software – a system that relies on detailed, structured description of a problem domain to facilitate automation, extensibility, and maintainability – represents an evolutionary advance in the design and implementation of systems that have a large number of interlocking parts. The motivation underlying this approach is similar to that behind the development of the first high-level programming language, FORTRAN in the 1950s. John Backus, the leader of the effort, stated in an interview conducted by fellow software legend Grady Booch that FORTRAN was a consequence of “laziness, [which was] my motivating force in most of what I did, how to avoid work.... programming is a pretty low-level enterprise, and somebody ought to be thinking about how to make it higher level...”¹

In both cases, productivity is facilitated by working at a higher level of abstraction: a framework takes care of the lower-level details. The difference is that the programmer working with a high-level language has a ready-made framework – a compiler, and today an integrated development environment as well – while metadata-driven frameworks must be constructed from scratch. Nonetheless, for systems intended to have longevity, the effort of framework construction is ultimately amortized through increased productivity – chiefly through the delegation of mechanical tasks that would have required programming to domain-knowledgeable non-programmers. This book is intended to provide guidance on framework construction.

Production systems in biomedical environments – such as electronic medical record (EMR) systems, clinical study data management systems, and basic-research laboratory information management systems – are ideal candidates for the application of metadata-driven techniques, and the use of such techniques in this area – not all successful – precedes this book by at least three and a half decades, long before the term “metadata” was in use. I therefore lay no claim to originality of many of the ideas described here. I have, however, attempted to provide a synthesis as well as identify underlying principles that seem to be important for ultimate success.

Metadata-driven techniques are also being employed in large systems oriented to business, such as Enterprise Resource Planning and Customer Relationship Management packages. Unfortunately, many implementations eliminate several essential aspects of the approach, as I will explain in the text, and so result in certain undesirable side-effects.

¹Computer History Museum, Ashland, OR. Oral History of John Backus. 9/5/2006. Available at http://archive.computerhistory.org/resources/text/Oral_History/Backus_John/Backus_John_1.oral_history.2006.102657970.pdf.

Motivation for This Book

A major impetus behind my writing this book is my concern with the current state of medical software, which lags greatly in functionality-to-price ratio compared to the non-medical world. The open-source/freeware movement has stellar examples of industrial-strength software – the Linux operating system, the Apache Web Server, Open Office, the MySQL RDBMS, the Eclipse integrated development environment, the Subversion Source-Code-Control System, and innumerable special-purpose bioinformatics tools – which are good enough to keep the large commercial vendors honest in terms of providing enough added value to justify their products’ purchase.

By contrast, the number of open-source medical applications of similar quality is miniscule. I don’t believe that doctors are significantly less altruistic or idealistic than others, though the Hippocratic Oath appears to be largely an antiquated oddity in today’s world of profit-oriented managed-care conglomerates: the USA seems to be the only developed country where basic healthcare is not regarded as a fundamental right.

The shortfall of quality medical-software applications is possibly due to lack of a critical mass of developers: to have sufficient good programs, there must be a large number of programs to start with, so that the Darwinian/Spencerian process of survival of the fittest can begin. Current medical-informatics training programs, which do not, in general, emphasize the acquisition of software-development skills to the extent that bioinformatics programs do, may contribute to this shortfall. The doctors who have earned my respect for their software knowledge have been largely self-taught.

Therefore the shortfall may have to be corrected by non-medically-trained individuals getting into the medical software field. The Obama administration’s declared goal of moving all US healthcare providers to electronic records offers a significant opportunity for those who are prepared to create the equivalent of Volkswagen Beetles rather than Boeing 747s: the typical group practice or solo practitioner simply cannot afford the multi-million-dollar systems offered by the current EMR market leaders. Hopefully, some of the affordable offerings will become available as open-source code that relies on a support/maintenance model for revenue generation, such as is followed by several Linux vendors.

Open-source projects often begin as efforts by a single person or a handful of driven individuals, where the artificial divide between “architect” and “programmer” does not exist. To build good systems, one needs both good development skills as well as a thorough knowledge of the problem one is trying to solve. Knowledge of software history – what has worked and what hasn’t – also helps. This book, in its modest way, aims to lay down some of the foundations of that necessary knowledge.

Target Audience

This book is aimed at individuals who have to interact with large biomedical database systems – such as Electronic Health Record or Clinical Research Information Systems – at

a level beyond that of end-user. In other words, as opposed to merely operating the software, they need to peek under the hood, either occasionally or as a major part of their daily work. Such people may work in the capacity of informatician or information-technology professional (application programmer or administrator).

The book can also be studied by 3rd–4th year undergraduates and graduate students/fellows who are in bio/medical informatics degree programs, or in bachelors/masters programs in IT with a concentration in Health Information Technology.

The readers who can potentially benefit most from the book are advanced developers or system architects who may actually need to build software interfaces to such systems, or may even be thinking of designing new systems, but who have just begun transitioning to the healthcare field. Such individuals may require some reorientation because they will discover that the way many large biomedical systems are actually designed differs quite significantly from the business-IT systems that they may have been used to, or from the textbook-based database designs they learned during their formal education.

This book focuses on complex problems. Here the metadata has a complex structure, which is therefore best managed, I believe, by using a relational database management system (RDBMS). Therefore some knowledge of this area is a pre-requisite. You should have used at least a microcomputer RDBMS to develop a multi-table application, and you should have at least looked at an Entity-Relationship (ER) diagram (or its more generalized counterpart, the Unified Modeling Language Class Diagram). If you have worked with SQL – you may have composed it using a graphical user interface without even being aware of it – you will appreciate this book more.

Similarly, some programming knowledge will also benefit you greatly: I have, however, tried to divide most chapters into a non-technical overview, followed by technical details. A few chapters, however, notably those dealing with decision support, require modest or minimal technical knowledge.

The book necessarily uses many examples from the biomedical world, but hopefully these are not too abstruse. You should not have taken a college-level bioscience course to understand them, and I try to provide a layperson's overview here even at the risk of having medically trained readers tune out. However, biased by my own medical background, I may unconsciously take for granted the reader's knowledge of what I consider elementary medical facts. If so, I apologize in advance, and would appreciate specific feedback from non-medical readers: I have had input, however, from at least one beta-tester who lacks significant life-science exposure.

While this book deals mostly in facts, it also discusses themes where the final word has not been written. The topic of metadata is rapidly advancing, and what is dictum today may be obsolete dogma tomorrow. However, I cannot shy away from the responsibilities of conveying to you what, based on both a survey of the literature as well as our own experience, has worked, and what has not. Therefore, I will state my opinions, flagging them clearly as such, and back them with evidence that I intend you to weigh, while fully understanding that contradictory evidence may exist or arise later. I intend to provoke thought, not pretend to a spurious infallibility. If I get you to engage in an internal dialogue with this book that vigorously challenges some of its assertions on occasion, I will have succeeded in my goals.

Acknowledgments

The text benefited greatly from alpha-testing by my brother and mentor Dr. Ravi Nadkarni, and beta-testing by Dr. Hemant Shah (currently of Henry Ford Health System, Detroit, MI), Randy Neall of Veriquant LLC, Hendersonville, NC and Gary Argraves of Prosoft Systems and Array Genetics, Newtown, CT. I acknowledge my contributions of my colleagues at Yale, notably Dr. Luis Marenco and Dr. Kei-Hoi Cheung.

My thinking in medical informatics has been influenced by Dr. Randy Miller, my former Editor in Chief at the Journal of the American Medical Informatics Association, and Dr. Daniel Masys, currently chair of the Department of Biomedical Informatics at Vanderbilt.

Contents

1	What Is Metadata?	1
1.1	Introduction	1
1.2	What Are “Metadata-driven” Software Systems?	2
1.2.1	Historical Perspective: Database Management Systems	2
1.3	The Importance of Metadata to Large Biomedical Systems	4
1.3.1	Rapid Evolution of Knowledge	4
1.3.2	Extensibility by the Customer/User: Metadata-Driven User Interface Generation	5
1.3.3	The Need for Inter-operability	6
1.4	Classifying Metadata	8
1.5	How This Book Is Organized	8
1.5.1	A Word on Entity-Relationship Diagrams Used in the Book	9
1.6	Principles of Implementing Metadata-driven Systems	10
1.6.1	How can Metadata “drive” Software?	10
1.6.2	Representation of Metadata: Guidelines	11
1.7	Safeguarding Metadata	12
1.8	Pitfalls in Metadata-driven System Development	13
1.8.1	Making Metadata Comprehensible: Detailed Documentation	15
	References	15
2	Data Types in the Medical Record	17
2.1	The Categorical (Nominal) and Ordinal (Ranked) Data Types	17
2.1.1	The Boolean Data Type	18
2.2	Numeric Data Types	19
2.2.1	Integers	19
2.2.2	Real Numbers	21
2.3	The Date Data Type	22
2.4	The Text Data Type	23
2.4.1	Pseudo-Numeric Data	23
2.4.2	Using Text to Store Data of Another Intrinsic Type: A Warning	23
2.5	The Binary Data Type	24
2.6	Custom (User-Defined) Data Types	25
2.7	Structured vs. Unstructured Data	25

2.8	Representing Enumerated Data.....	26
2.8.1	The Older Approach.....	26
2.8.2	The Recommended Approach: Consolidating Definitions	27
2.8.3	Missing Values: Nulls	29
	Appendix: Ensuring Referential Integrity with a Consolidated-Definition Design.....	29
	References	31
3	Metadata and the Use of XML.....	33
3.1	Introduction to XML	33
3.2	Strengths of XML	35
3.2.1	XML Tool Base.....	37
3.2.2	XML Data Formats Are More Flexible	38
3.2.3	XML Is Embraced and Implemented by Large Organizations	38
3.3	Misconceptions About XML Use	39
3.3.1	Misconception: XML Is Easy for Humans to Work with and Understand	41
3.3.2	Misconception: XML Allows the Semantics of Document Content to Be Adequately Described.....	42
3.3.3	XML is a Misfit for Procedural Programming.....	43
3.3.4	Misconception: XML Technologies Will Eventually Render Relational Databases Obsolete.....	44
3.4	Metadata-Related Uses of XML in Biomedicine.....	47
3.4.1	Biomedical Data Modeling	47
3.4.2	Annotation of Reference Content	49
3.4.3	XML for Data Interchange: Guidelines	49
3.4.4	XML for Data Export.....	50
3.4.5	Medical Natural-Language Processing	51
3.5	Conclusions.....	52
	References	52
4	Representing Structured Clinical Data	55
4.1	Introduction.....	55
4.2	The Protocol	56
4.3	Representing Structured Data: The Challenge.....	57
4.4	Storing Sparse Data.....	59
4.4.1	Speeding up Search of Keys: Indexes.....	60
4.4.2	Storing Key-Value Pair Data.....	61
4.4.3	Entity-Attribute-Value Tables	63
4.4.4	Using EAV Modeling Judiciously: Caveats.....	65
4.5	Aside for Software Developers: Should You Use EAV Tables or Hierarchical Key-Value Pairs?	70
4.5.1	MUMPS: Myth and Reality	70
4.5.2	Hierarchical Key-Value Pairs as a DBMS Foundation: Caché.....	72
4.5.3	Summary	72
	References	73

5	Building the User Interface for Structured Clinical Data Capture	75
5.1	Introduction.....	75
5.1.1	Pitfalls of a User Interface Metaphor That Directly Reflects EAV Storage.....	76
5.1.2	Representing the Protocol.....	77
5.2	Building the User Interface.....	77
5.2.1	Hand-Coding of Interfaces.....	77
5.2.2	Automatic Interface Generation.....	78
5.3	Metadata for Form-Generation Frameworks: Requirements.....	79
5.3.1	Ensuring Validity of Data Input.....	80
5.3.2	Facilitating Ergonomic Operation.....	81
5.3.3	Protocol-Subcomponent Metadata: Consistency Checks	83
5.4	A Metadata Sub-schema for Interface Generation.....	84
5.4.1	Definitions of Form Elements.....	84
5.4.2	Implementing Computed Fields.....	89
5.4.3	Modeling and Implementing Skip Logic	91
5.4.4	Modeling Complex Validation and Alerts	93
5.4.5	Implementing Complex Computations and Validations That Access the Database	94
5.4.6	Implementing Conditional Lists.....	95
5.4.7	Practical Issues Related to Expressions for Computation and Validation	95
5.5	Testing Generated Forms and Troubleshooting	96
5.6	Form Generation Issues.....	97
5.6.1	Static vs. Dynamic Components.....	98
5.7	Capturing Data Edits.....	98
5.7.1	Clinical Data Sub-Schema	100
5.7.2	Mapping Between Form Elements and Database Elements	102
5.7.3	Converting the User's Edits into SQL Operations.....	103
5.8	Choice of Technology for the Forms Runtime Environment.....	104
5.8.1	Browser Code Development	104
5.8.2	Web Server Code: Compiled Versus Interpreted Environments.....	104
5.8.3	Newer Technologies.....	105
5.9	Flowsheets: Allowing Users to Create Their Own and Forms in EMRs	106
5.10	Open-Source EMRs	107
5.10.1	VAMC Vista.....	107
5.10.2	OpenMRS.....	108
	References	108
6	Medical Decision Support Systems: General Considerations	109
6.1	Introduction to Alerts	110
6.2	Implementation of Alerts	111
6.3	Accuracy of Alerts	113
6.4	The Choice of Thresholds	115

6.5	Workflow and Software-Integration Considerations.....	116
6.6	Alerts and the Law of Unintended Consequences	117
6.7	The Lessons of History: The Microsoft Office Assistant.....	119
6.8	Conclusions and Outstanding Issues: Customizability	120
	References	121
7	Challenges in Electronic Decision Support Implementation.....	123
7.1	Differences in EMR Designs.....	123
7.2	Sharing Medical Algorithms: Arden Syntax.....	124
7.2.1	A Brief Arden Syntax Example	124
7.2.2	Essential Features of Arden	125
7.2.3	Limitations of Arden Syntax.....	126
7.3	Complex Clinical Guidelines	130
7.3.1	Nature of a Complex Guideline	131
7.3.2	Making Narrative Guidelines Actionable: Challenges	132
7.4	Achieving Schema Independence for Decision-Support Software.....	135
7.4.1	Schema Independence Through Views.....	136
7.4.2	Schema Independence Through a Subroutine Library.....	137
7.4.3	The Virtual Medical Record.....	138
7.5	Summary	141
	References	141
8	Infrastructure for Complex Clinical Decision Support: Rule Engines	143
8.1	Introduction.....	143
8.2	Operation of the Rule Engine: Forward and Backward Chaining	145
8.3	How Rule Engines Work.....	146
8.4	The Strengths of Rule Frameworks.....	147
8.5	Challenges and Pitfalls of Rule-Based Approaches.....	148
8.5.1	Conflicting or Redundant Rules.....	148
8.5.2	Software-Engineering Skills Do Not Become Obsolete.....	149
8.5.3	Comprehensiveness and Reliability	150
8.5.4	Alternative Approaches May Be More Appropriate.....	151
8.6	Case-Study: Implementing a Rules Management System	151
8.7	Conclusions	154
	References	154
9	Decision Support Infrastructure: Workflows and Service-Oriented Architectures.....	157
9.1	Introduction.....	157
9.2	What Is a Workflow?.....	157
9.3	Is a Workflow Software Framework Essential?	159
9.4	Workflow and Web Technologies.....	160
9.4.1	Web Service Basics	161
9.4.2	Challenges in Implementing SOA	162

9.5	Languages for Workflow Specification	164
9.5.1	Business Process Execution Language	165
9.5.2	Business Process Markup Notation.....	166
9.5.3	BPEL and BPMN Limitations	168
9.6	General-Purpose Workflow Toolkits	170
9.6.1	WF Essentials.....	170
9.7	Caveats in Applying Workflow Technologies.....	171
9.7.1	Flexibility of Workflow (Re) Design	171
9.8	Conclusions.....	174
	References	175
10	Complex Decision Support in Practice.....	177
10.1	Introduction	177
10.2	Table Driven Applications	177
10.2.1	Example: Drug Interactions	178
10.2.2	Table-Driven Consultation Systems.....	179
10.3	Complex Decision Support: Guideline Representation Languages.....	181
10.3.1	Reasons for Lack of Success.....	181
10.3.2	Expression-Language Standardization Efforts.....	186
10.4	Web Services for Clinical Decision Support.....	187
10.5	The Proteus Guideline Framework	189
10.6	Conclusions.....	191
	References	192
11	Extending the Entity-Attribute-Value Model	193
11.1	Additional Indications for EAV Design	193
11.1.1	Numerous and Volatile Data Classes, Modest Data Volumes.....	194
11.1.2	Hybrid Classes	194
11.1.3	Maintaining an Audit Trail Across Numerous Tables in a Schema.....	195
11.1.4	Special-Purpose EAV: The Semantic Web.....	197
11.2	Requirements for Mixed Schemas	197
11.2.1	Limitations of the Basic EAV Model	197
11.2.2	Supporting Routine Operations.....	198
11.2.3	The Need for Additional Data Type-Specific EAV Tables	198
11.3	The EAV/CR Approach.....	198
11.3.1	RDBMS Catalog Tables: An Introduction	199
11.3.2	Dictionary Tables: Summary	200
11.4	The Dictionary/Metadata Tables	200
11.4.1	Synchronizing Dictionary and Catalog Table Contents.....	202
11.4.2	Attribute-Definition Reuse: User-Defined (Custom) Data Types ...	204
11.4.3	Implementing Custom Data Types.....	204

11.4.4	Aside: The Deliberate Absence of Descriptive Metadata Can Obfuscate Schemas	205
11.5	EAV/CR Data Sub-schema.....	206
11.5.1	The Object Dictionary Approach.....	206
11.5.2	Representing Sub-structure for EAV Data	208
11.5.3	Links to Controlled Terminologies	208
11.5.4	Representing Arrays.....	209
11.6	Data Presentation and Search/Browsing Considerations	209
11.7	The User Interface for Mixed Schemas: Self-mapping.....	214
11.7.1	Self-mapping for EAV/CR Designs	214
11.8	Generating SQL for CRUD Operations	215
11.8.1	Fundamental Security Considerations: SQL Injection.....	215
11.8.2	Implementing the Delete Operation.....	217
11.8.3	Implementing the Update Operations	217
11.8.4	Implementing the Create Operation.....	219
11.9	Selective User Access to Individual Classes or Attributes	219
11.10	Case Study: Representation of Phenotype.....	219
11.10.1	Representing Phenotypic Data and Metadata in a Repository.....	220
Appendix 11.1:	Metadata Schema Details.....	222
	Meta_Classes	222
	The Meta_Attributes Table	222
Appendix 11.2:	Alternative Means of Representing EAV Data	224
	Using an XML Column for Sparse Attributes	224
	Microsoft SQL Server Sparse Columns.....	225
	References	225
12	Descriptive Metadata: An Introduction to Terminologies.....	227
12.1	Introduction.....	227
12.2	Types of Terminologies.....	228
12.2.1	Elementary Terminology Designs.....	230
12.2.2	Support of Hierarchies: Taxonomies.....	231
12.2.3	Support of Non-hierarchical Relationships: Thesauri.....	232
12.3	Combination with Object-Oriented Modeling and Algorithmic Knowledge: Ontologies	233
12.4	The Unified Medical Language System.....	234
12.5	Requirements of a Terminology: Cimino's Criteria.....	235
12.6	Terminologies as the Basis for Data Exchange.....	238
12.7	Recognizing Concepts from Terminologies in Clinical and Scientific Text.....	239
12.8	Composing New Concepts from Existing Ones.....	241
12.8.1	Pre-coordinated Concepts vs. Post-coordinated Concepts: Concept Recognition.....	241
12.8.2	Description Logics	244
12.8.3	Practical Issues with SNOMED-CT	246
Appendix:	Working with the UMLS	246
	References	247

13 Descriptive Metadata: Implementing Large-Scale Biomedical Ontologies	249
13.1 Functional Requirements of Large-Scale Ontologies	250
13.2 Ontology Design Principles: Some Warnings.....	251
13.2.1 Ontologies vs. Databases	251
13.2.2 Prototypes vs. Production Systems	253
13.2.3 The Closed-World vs. The Open-World Assumption	255
13.2.4 Semantic Web Technologies: OWL and SWRL	256
13.3 The Elements of a Large-Scale Ontology Schema.....	257
13.3.1 The Thesaurus Kernel	258
13.3.2 The Metadata and Data Subschema Schemas.....	260
13.4 Special Issues: Dealing with Imported Content.....	261
13.4.1 Class and Attribute Structure in UMLS.....	261
13.4.2 Challenges in Dealing with Types and Enumerations	261
13.4.3 Separating Lower-Level from Higher-Level Metadata.....	262
13.4.4 Challenges in Dealing with Imported Inter-concept Relationships.....	264
13.5 Scaling Up: Security and Audit Trails	265
13.5.1 Enforcing Selective Access.....	265
13.5.2 Maintaining Audit Trail Information: Change Management.....	266
13.5.3 Reporting Metadata and Data Changes.....	268
13.6 Case Study: The SNOMED CT Machine-Readable Concept Model.....	269
References	271
 14 Clinical Study Data Management Systems	273
14.1 Introduction.....	273
14.2 Differences Between CSDMSs and EMRs	273
14.2.1 Patients vs. Subjects.....	274
14.2.2 Recruitment and Eligibility.....	274
14.2.3 The Study Calendar.....	275
14.2.4 Projects.....	278
14.2.5 Differences in Data Capture Between CSDMSs and EMRs.....	278
14.2.6 CSDMSs Implement User Roles That Are Specific to Research Designs	282
14.2.7 The Scope of a CSDMS May Cross Institutional or National Boundaries	283
14.3 Data Storage in CSDMSs.....	285
14.3.1 Drawbacks of the Separate-Database Approach.....	285
14.4 CSDMS-Related Processes During Different Stages of a Study	287
14.4.1 Study Planning and Protocol Authoring	287
14.4.2 Protocol Management and Study Conduct.....	289
14.4.3 Analysis and Reporting.....	290
14.5 Summary	291
References	291

15 Data Retrieval for Heterogeneous Data Models	293
15.1 Types of Retrieval Operations.....	293
15.2 General Principles of Framework Creation for Data Retrieval.....	294
15.2.1 Complex Problems Defy Simple Solutions	294
15.2.2 Focusing Your Approach	295
15.2.3 Ad Hoc Query: Implementation Considerations.....	296
15.2.4 Infrastructure Considerations.....	297
15.2.5 Restricting the Output Based on User Access Permissions	300
15.3 Query by Form (QBF).....	301
15.3.1 Patterns for Criterion Specification.....	301
15.3.2 Presenting Results to the User	303
15.3.3 Controlling the Number of Records Returned	304
15.4 Reporting.....	304
15.4.1 Composing SQL Using Querying Tools: ODBC-Related Issues....	305
15.4.2 Composing SQL: Dealing with a Heterogeneous Data Model.....	306
15.4.3 Presentation/Formatting Issues	306
15.4.4 A Metadata-Based Framework for Managing Reports	308
15.4.5 Meta_Reports	310
15.4.6 Meta_report_steps	311
15.4.7 Meta_report_columns	313
15.4.8 Meta_report_params	314
15.4.9 Meta_Reports_Roles.....	315
15.4.10 Requirements of the Reporting Framework Runtime	315
15.5 Ad Hoc Query	316
15.5.1 The Sparse-Data Problem	317
15.5.2 Temporal Queries.....	319
15.5.3 Selecting Attributes of Interest	320
15.5.4 Specifying Selection Criteria and Output Options for Attributes...	322
15.5.5 Saving Queries for Reuse: The Schema Volatility Problem	324
15.5.6 Schema for Query Representation	326
15.5.7 Generating SQL for Ad Hoc Query	330
15.6 Special Topics in Query and Reporting	340
15.6.1 Pre-compilation of Queries	340
15.6.2 Accessing Non-relational Data: XML and Narrative Text	341
15.7 Bulk Data Extraction.....	343
15.7.1 Extraction of Clinical Data	344
15.7.2 Extracting of Data with Sub-structure	344
15.8 Exporting Metadata to Make Extracts or Ad-Hoc Query Output Comprehensible.....	350
15.8.1 Metadata for Meta-analysis.....	350
Appendix: Composing SQL that Uses Relationship Metadata	352
The Universal Database Concept.....	352
Weaknesses in the UDB Premise	353
Cycle Detection and Elimination	354

Can Database Schemas be “Hidden” Successfully Using a UDB Approach?	355
Join Metadata in EAV/CR Designs	356
Dealing with Simulated Relationships.....	357
References	358
16 Metadata for Data Warehousing.....	359
16.1 Introduction: Data Warehousing 101	359
16.1.1 Star-Schemas Are Not Always Mandated.....	361
16.1.2 Approaches for Creating a Warehouse.....	361
16.2 The Role of Metadata in Data Warehousing	364
16.3 Toolset for Warehouse Metadata Management.....	365
16.4 Categories of Warehouse Metadata.....	365
16.4.1 Data-Source Metadata.....	365
16.4.2 Metadata Generated During the ETL Process.....	368
16.4.3 Front-End Metadata	369
16.5 Case Study: The I2B2 Star Schema	370
16.5.1 Security Architecture	371
16.6 Conclusions.....	371
References	372
17 Biomedical Metadata Standards.....	373
17.1 Self-Describing Attribute-Value Pairs.....	374
17.2 Standards for Descriptive Metadata: ISO/IEC 11179	375
17.3 Technical Metadata Standards: CDISC.....	376
17.4 Interchange Standards for Gene Expression and Proteomics	377
17.5 Proteomics Standards.....	378
17.6 Metadata for Medical Observations: OpenEHR Archetypes.....	379
17.7 The HL7 Version 3 Reference Information Model (RIM).....	380
17.7.1 Overview.....	380
17.7.2 Limitations and Controversies	382
17.7.3 The HL7 Clinical Document Architecture (CDA).....	384
17.8 Final Words on Standards	385
References	386
Index.....	387

1.1 Introduction

The one-phrase definition of “metadata” – “data about data” – is too broad to have practical value. However, the uses of metadata itself across all of computing, and the ways in which it is represented, are so numerous and diverse that it would be hard to find two practitioners who think about it in exactly the same terms. An inspection of Wikipedia’s article on the subject,¹ especially the accompanying discussion, will make it clear that the topic has grown to the point where its editors believe that the article has become too large and unwieldy – and yet it barely touches on the themes that are of specific relevance to this book.

For the purposes of this book, I will define metadata as data whose primary purpose is to describe, define and/or annotate other data that accompanies it. The accompanying data may be in the same physical unit as the metadata (e.g., the same file on disk), or in the same logical unit (e.g., the same database, or the same organization/enterprise). The information stored in the metadata may be in a form that can be understood by non-technical humans, or in a form that can be used by software: often both kinds of information co-exist. This book will concentrate on metadata that accompanies data in the same database.

In his book, “Building and Managing the Metadata Repository”,² which was something of a landmark work in the business Information Technology (IT) field when first published, David Marco says “when we mention metadata, we are really talking about knowledge”. From the practical perspective, this means that if you have metadata accompanying some data, it becomes possible to do things with the data that it would be much harder or even impossible to do if that metadata did not exist.

A Note on Style – “We” versus “I”: In this book, I’ll mostly use first-person singular rather than the editorial first-person plural. Ben Zimmer, in an amusing article in his “On Language” series in the New York Times magazine,³ cites Admiral Hyman Rickover’s quote that only three types of individual are entitled to use “we”: “a head of a sovereign state, a schizophrenic and a pregnant woman”. He also cites post-civil-war Senator Roscoe Conklin’s admonition that only three classes of people habitually use “we”: “emperors, editors, and men with tape-worms”. In this book, I’ve only edited my own work. When I use “we”, it will be

in two circumstances: where I engage the reader in a joint exploration of ideas, and in reference to work that my colleagues and I have accomplished together, to acknowledge a collective effort.

- “*Metadata*”: *Singular or Plural?*

Strictly speaking, the singular of “data” is “datum”. In common usage, the word “data” is now used as frequently in the singular form, to refer to a set or collection of data items, as in the plural form. Nobody seems to be using the word “metadatum” to describe a single unit of metadata (Microsoft Word has even flagged it as a spelling mistake). “Metadata” has evolved into one of those words which, as for articles of clothing such as trousers, shorts and briefs, have become their own singular. Throughout this book, therefore, the singular usage of “metadata” will predominate.

1.2

What Are “Metadata-driven” Software Systems?

Metadata is adjective information that describes primary data. The coupling of metadata and primary data in most cases is vital for systems operation, but this not always so. For example, a web page may include optional metadata specifying what language it’s written in, what tools were used to create it, where to go for more on the subject and so on.

Metadata-driven software systems are systems that rely on a metadata component to the extent that they would either not work at all, or become unusable, if the metadata did not exist or if it became corrupted or inconsistent. This is because they are designed in such a way that the program code constantly consults the metadata in order to determine courses of action during various system operations: changing the metadata changes the system’s behavior.

Deliberately designing a system with an Achilles heel would not seem to be good software-engineering practice, but as we shall see, metadata is a component of practically any non-trivial software application today: it is only a matter of the extent to which it is utilized. Therefore there must be significant benefits to designing and implementing such systems that more than offset the supposed vulnerability.

In introducing a well-known example of a metadata-driven system below, I will take the risk that it may be too elementary to those with a background in computer science or information technology. Since this book is aimed at a broader audience it is prudent to demonstrate using excessive simplicity rather than force the non-technically adept to interrupt their stream of thought by forcing them to look up an external reference.

1.2.1

Historical Perspective: Database Management Systems

In the early days of business data processing with COBOL, programmers worked with individual data files, which were typically variations on text files. As data processing needs

evolved in sophistication, applications worked with *collections* of data files – a “data base” – where data consistency had to be maintained across the individual files. The need was felt for software that assisted consistency management of this file collection/database firstly, by cataloging the descriptions of the individual files within them, and secondly, by treating the separate files as a single logical or even physical unit. The data catalogs of the early database engines were among the first instances of metadata, though this term had not yet gained currency.

The data catalogs played a very limited role for a long time. Much later, through work that began with Edgar F. Codd, the founder of relational database theory,⁴ it was shown that, if the catalog information was detailed enough, software could consult it to simplify the task of data retrieval significantly. Currently, there seems to be universal agreement that there’s no point putting data into a system unless you can get it out when you need it, and in the format that you need it. In the 1960s, however, the problem of data retrieval was considered onerous enough that many data systems had something of a black-hole-like quality, with practically unidirectional information flow across an event horizon.

Today, we take for granted Structured Query Language (SQL),⁵ the universal data-manipulation and data-definition language of relational databases. It is both amusing and educative, however, to browse through 1970s-vintage books by authorities of the time such as James Martin (who effectively conveyed wild enthusiasm over the prospect of using relational systems in his book “Computer Data-base Organization”⁶) to realize that a data-retrieval task that we can accomplish with two or three lines of SQL – or even less effort, by using a graphical user interface (GUI) front end that generates SQL – literally took pages of error-prone code written over a week or longer by an experienced programmer.

Martin, in fact, seriously considered, and indeed looked forward to, the possibility that business-application programmers might become an endangered species, because he believed that SQL could be employed by end-users: his 1981 book, “Application Development without Programmers”,⁷ reflected this belief. This hasn’t quite happened today, but only because the realm of the possible has expanded and the problems have become more challenging and sophisticated.

Be that as it may, SQL’s operation would not be possible without the presence of catalog metadata. The power of today’s relational database management systems (RDBMSs) stems from the fact that they store metadata that not only describes a database’s tables and columns, but also on which columns are indexed for fast access, which columns in one table are logically linked to columns in another, distribution of data values in individual columns, and so on. This metadata not only insures that a SQL query can be answered, but that it will be answered as efficiently as possible.

In this book, when I talk of metadata-driven software systems, I will refer to systems where the designers, while building their architecture on top of existing commercial or open-source technology such as RDBMSs, add value by introducing *their own* metadata component for a purpose that the technology vendor could not have anticipated, or to address a problem that the vendor software is not really intended to solve. However, as far as the underlying technology itself is concerned, the designer-defined metadata is treated just as any other data would be. Metadata is just a kind of data: it is only its purpose that makes us confer a separate label.

1.3

The Importance of Metadata to Large Biomedical Systems

Up to this point, we have discussed software systems in general. Large-scale biomedical software systems, however, have certain aspects that make them particularly suitable to the application of metadata-based software design. By “large-scale”, I imply that the systems perform a mission-critical role within an organization or even across organizations, are used concurrently by a diverse base of users of varying sophistication, and managed by a team dedicated to the support task.

1.3.1

Rapid Evolution of Knowledge

Biomedicine is not the only field where new knowledge accumulates rapidly, but it is certainly represents one of the largest in terms of financial impact. From the software perspective, new knowledge means that new types of data arise, or that existing data elements need to be put together in different ways. Software systems that cannot be extended to allow capture or representation of this data soon become unfit for their intended purpose.

For those readers who have designed and operated their own databases, it may seem a simple enough matter to extend an existing database design by adding new tables or new columns. However, it is not merely a question of adding new data elements: one needs to build a user interface to these elements as well. For developer teams that know their own system thoroughly, the task of hand-crafting additional user interface screens is one of the significant chores involved in system maintenance.

In the case of commercial packages such as electronic patient record systems or clinical study data management systems, the problem is far more complex. Source code is not typically distributed to the customer base. It may be held in escrow in the eventuality that the customer may need to access it if the vendor declares bankruptcy, but in any case, customers are not expected to be familiar with it. Today’s clinical data systems still leverage commercial DBMS technology, however, so the database’s design, or schema, remains mostly an open book.

In many cases, customers insist on the ability to make modest extensions to the system to capture new kinds of data that the vendor could not have anticipated. However, as stated previously, simply adding new database elements directly by modifying or adding tables does not solve the problem: These elements would not appear in the user interface because the source code, unless also modified, would not reflect their existence. Even if a customer chose to build a custom user interface to these elements, this interface would not be integrated with the existing interface to the rest of the system, making its seamless use problematic.

1.3.2

Extensibility by the Customer/User: Metadata-Driven User Interface Generation

Customers who have spent several million dollars purchasing a system and training their staff are not pleased if they find out that for every supposedly minor or even trivial modification that they need to make, they are forced to contact the vendor. Sometimes, new data elements (e.g., for a public health emergency) may need to be put in place and used for data capture within days or even hours. Modification requests to the vendor take much longer to make, verify for correctness, document and distribute to customers. Even if a vendor was equipped to respond so rapidly, making changes to a system that might benefit only a single customer (e.g., for a geographically limited disease outbreak) is more trouble than it is worth: one risks ending up eventually with multiple, subtly different versions of the source to maintain, one at each customer site.

The designers of extensible systems therefore provide mechanisms through which customers can define new elements, or modify the definition of such elements, which can then be made to appear in standard user interfaces. When I discuss the architecture of electronic medical records (EMRs), I shall describe certain database designs that facilitate such processes greatly. These designs are critically reliant on the presence of metadata that records, in very great detail, the definitions of user-defined data elements in terms of what their contents should be as well as how they should be presented in the user interface. These definitions are consulted by software that performs semi-automatic user-interface generation.

I shall later make the case that, even in circumstances where a given organization has built a large home-grown system that is being used organization-wide, there are major advantages to adding a metadata component that is coupled to an interface generator, and allow designated super-users to extend the system by defining new data elements via the metadata. User-interface maintenance is one of the more mechanical tasks involved in system maintenance, and especially when the support staff is small, developer resources are scarce. It is more cost-effective and time-effective to let non-programmer super-users take responsibility for interfaces that their departments will use.

1.3.2.1

Tradeoffs with Automatically Generated Interfaces

Automatically generated interfaces are much faster to set up and modify than those that have been built by hand: the more advanced generators make it possible to generate functionality that would be very tedious as well as error-prone to implement on a case-by-case basis. On the minus side, the user of a generator sacrifices extensive customizability and possibly some esthetics. However, while an automatically generated interface may not be as esthetic as one that is meticulously hand-crafted by a skilled designer, this does not necessarily mean that it will be more difficult to use. If the interface generator utilizes thoughtfully-chosen standard styles and layout metaphors, the resulting interfaces may

actually be easier to use than one-off screens designed by programmers who have not been specially trained in interface design, and they will provide a consistent user-interface experience. Such consistency makes them easier to learn, and less error-prone to use routinely.

Clinical Study Data Management Systems (CSDMSs) share numerous similarities with EMRs, as I shall discuss later, but serve the purpose of supporting research primarily, and patient care secondarily or coincidentally. Given that many data elements originate in the leading-edge research of individual investigators using the CSDMS and may not be known outside a single research group, the need to allow customers to define their own data elements becomes even more important. Metadata consequently plays a very large role in such systems.

There is a caveat here: if the user-interface generation capabilities are sufficiently powerful and sophisticated, it takes time for users to become skilled in being able to use them fully. In many organizations, power-users who will expend the effort to do so may not be identifiable. Accepting this situation, certain commercial CSDMS vendors implement a strategy of selling the base system at a very affordable price, with the commitment to design new data elements whenever a new clinical study comes along, with pricing on a per-element/per-form basis. This strategy is akin to K.C. Gillette's marketing strategy of selling safety razors at a nominal price in order to make a profit selling razor blades.⁸ Since clinical study setup is typically a somewhat more leisurely process where a study does not need to live in less than a week (as opposed to a public-health emergency, where the elements must be set up in a day or less) such a business strategy is viable. Such vendors, however, use an internal (i.e., customer-inaccessible) metadata-based framework to facilitate their own GUI-generation chores and achieve much faster turnaround times than could be achieved by hand-coding these interfaces.

In brief, a metadata-driven design facilitates *flexibility* and *adaptability*. The first time a new problem arises, you have to step back a bit and abstract a solution, but once the solution has been created, you don't need to re-create it over and over again.

1.3.3

The Need for Inter-operability

Even the largest biomedical software vendors are unlikely to sell every kind of software that a large institution may need. They will often partner with smaller vendors who specialize in niche markets and who promise the customer that their offerings will be compatible with those of their larger partner. But even in such circumstances, getting systems to communicate with each other is still far from trivial.

The problem is particularly acute for systems supporting the electronic health record. I know of cases where one vendor's system is used for in-patient care and another vendor's offering is used for ambulatory (out-patient) care, and getting the two to exchange data (which is useful because one would have a complete picture of a given patient) requires all the delicacy and skill of a heart-lung transplant, and that is something of an understatement: implementing such capability has taken far longer, and has been several orders of magnitude more expensive, than surgery on a human patient would have been.

This difficulty is partly the consequence of the design of the earlier version of the standard communications protocol for passing messages between systems, Health Level Seven (HL7) version 2.3.⁹ HL7 v2.x was so underspecified that while a vendor could claim compliance with the standard because their software put out messages in HL7 syntax, many aspects regarding the actual *meaning* of the clinical content of the message were left up to the vendor to interpret. Understandably the probability that two vendors would likely to follow an identical approach was very low.

The latest version of HL7, version 3.0 tries to address the meaning issue, as we shall see later. As a change from the previous version, however, it is revolutionary rather than evolutionary, so that existing software would require major overhaul. Resistance from established vendors, who are voting members of the HL7 organization, has ensured that version 3.0 is still a draft standard rather than one with an officially mandated status. The most enthusiastic adopters of HL7 v.3 currently are the creators of new systems, who do not have to worry about backward compatibility with legacy versions of their software.

When multiple non-communicating systems store related data on the same patient, there is a real danger of creating a scenario akin to the one where multiple clocks in the same house show different times: it is hard to know which one to believe. Failure of systems to communicate has resulted in life-threatening medical errors such as a mismatched blood transfusion, because the blood-bank software used to record the blood group did not communicate with the inpatient system.

1.3.3.1

Issues Related to Data Semantics

For semantic inter-operability to occur, interchange of data must occur in a form where all systems are agreed on what a specific data element means. One source of the difficulties is that the same clinical parameter may be identified in separate systems using different internal codes, and sometimes different names as well.

Because medicine has evolved over a long time, it has a larger proportion of synonyms than most other fields. Greco-Latin and Anglo-Saxon word forms for the same concept are used interchangeably by clinicians: gastric/stomach, hepatic/liver, renal/kidney. Further, in laboratory medicine, the same parameter may be known by multiple names and abbreviations. Thus, a serum enzyme that measures liver function is known by both the names Alanine Aminotransferase (ALT) and Serum Glutamic-Pyruvic Transaminase (SGPT). Correct communication requires the creation of inter-system mappings that denote the identical nature of differently named or coded parameters.

I will discuss later how using controlled biomedical vocabularies, which serve the role akin to thesauri, and whose contents can be used as metadata, offer a potential solution. These vocabularies record biomedical concepts, associate them with standard codes (which are typically numeric), and record their synonymous forms. If the implementers of a system wish to make its data interoperable with other systems, then they must perform the initial effort of mapping the concepts in that system to a standard biomedical vocabulary. (Different biomedical vocabularies are appropriate for different purposes.) If every vendor follows suit, then if there are N systems that need to communicate with each other, instead

of creating $N C_2$ mappings, one need only N mappings to the vocabulary, whose contents now act as a lingua franca. This scenario represents a desirable Utopia rather than what is actually happening today: however, government sponsorship of controlled-vocabulary efforts has the goal of making it a reality.

1.4

Classifying Metadata

Several ways of classifying metadata exist, based on the domain one is working with and the specific problem that one seeks to address. While classification serves valuable purposes in most fields, I am not convinced that major efforts at classifying metadata are worthwhile: as we shall see later, one application's metadata is another's data. I will use Kimball's basic dichotomous classification,¹⁰ simply because in doing so I will introduce terms that I will use frequently in the rest of the book. I distinguish *technical* metadata, which is intended primarily to be interpreted by computer programs (i.e., software), from *descriptive* metadata, which is meant primarily to be interpreted by human users. The boundaries between the two are not fixed: certain metadata is both human-interpretable, being created manually by people, as well as processed by software for specific purposes. The reverse is also true: software can create metadata automatically behind the scenes during the course of its routine employment by users that knowledgeable individuals can inspect later.

Data catalogs are an example of technical metadata created by software, but which can be inspected by knowledgeable developers. The contents of biomedical thesauri serve as both technical metadata (when used by clinical systems to exchange information) as well as a descriptive purpose: in the latter role, they can be used as key-phrases to index and annotate clinical text. Data-element definitions in EMRs serve a dual purpose: they provide descriptions in ordinary language, but as we shall see later, they also contain information that software can use to determine how they are presented to the user in a GUI, and how their contents, as entered by the user, can be checked for correctness.

A special subcategory of technical metadata is *process metadata*: statistics generated by a system during the course of its operation. In the case of RDBMSs, information such as the number of rows in individual tables, space occupied by a table, and frequency distributions of data values in individual columns is computed automatically by the RDBMS and used by the query optimizer. In data warehousing, statistics generated during data preparation – e.g., error rate, time to complete a task, rows of data processed per unit time – provide quality and performance indicators.

1.5

How This Book Is Organized

This chapter is intended to serve as a roadmap to the rest of the book. I will lay down certain principles that will be reinforced in the subsequent chapters through detailed examples. Toward the end of certain chapters, case studies – mostly brief, but occasionally

extended – will be introduced. Some case studies will be based on work performed by others that I regard as noteworthy: the selection is biased toward work that has been described in sufficient detail that it can be reproduced or reverse-engineered, and where the design or code is available as open source. One case study serves as an “anti-pattern”: a lesson in how not to do things. As in music, art and architecture, learning of good systems designs is fostered by the detailed study of examples.

1.5.1

A Word on Entity-Relationship Diagrams Used in the Book

I have used two different kinds of ER diagrams in this book. Some were composed using Microsoft Visio 2007, while others (the majority) were composed using Microsoft Access 2003. The differences between the two diagrams are summarized below:

- **Column Names and Labels:** With Visio, the Primary Key and Foreign Key columns are automatically labeled, using the convention PK and FK (with suffixes 1, 2... for foreign keys). With Access, the names of Primary Key columns are bolded, but Foreign Key columns are not visually distinguished in any way.
- **Inter-Table Relationships:** With Visio, a relationship line simply begins somewhere in the Foreign Key table and end somewhere in the Primary Key table. *Within* Visio, determining the meaning of a particular relationship line is not difficult: one simply chooses the line/Relationship and then views the Properties of that relationship using the right-mouse context-sensitive popup menu.

However, *outside* Visio, interpretation is more problematic: you need to read the documentation accompanying the figure. Sometimes, looking at the name of a column labeled with an “FK” will provide a clue: its name may correspond exactly to the name of a primary-key column in another table.

This clue only works if the schema designer has been disciplined enough to ensure that the primary key column of a table is named using the convention <table-abbreviation>_ID. An abysmal convention I’ve seen recommended in a couple of texts – Microsoft Access will also let you shoot yourself in the foot if you allow it to suggest default column names for you – is to simply use the ambiguous column name “ID”. In this case, the schema will be much harder to figure out.

By contrast, with Access, a line denoting a relationship begins at the Foreign-Key column in the Foreign-Key Table and ends at the Primary-Key column of the Primary-Key table. The line may be straight or oblique, and may be obscured by the rectangle representing a table. The Foreign-Key end is typically labeled with an infinity symbol (for “many”), and the Primary-Key end with a “1” (one).

Overall, for modest-sized schemas, Access diagrams are easier to interpret in isolation, which is why I have preferred to use them. In production scenarios, however, I would be helpless without Visio. Visio can reverse-engineer a diagram automatically by querying a database schema. Further its automatic layout algorithm for tables (Access has none) is quite robust, and when tables that should be reasonably close together in the diagram are not, it indicates that a necessary inter-table relationship has not been defined.

1.6

Principles of Implementing Metadata-driven Systems

1.6.1

How can Metadata “drive” Software?

The term “driving” means controlling the flow of logic in software. *Lookup tables* are a widely used software technique where, as an alternative to hard-coding elaborate if-then-else logic, program code looks up a value in the table and decides on a path of action depending on what it encounters. In his classic treatise on software engineering, “Code Complete” (2nd Edition),¹¹ Steve McConnell gives the example of the problem of classifying a character as a letter, a number or punctuation. The table-driven approach stores the class of each character as an array that is indexed by the character, and replaces 12 lines of if-then-else code with a single line only containing four tokens.

The contents of lookup tables may be populated in several ways:

1. Software developers or subject-matter experts may create the contents. Often, the experts will specify what they need to contain, and the programmers will implement the specification. This is the approach used for *decision tables*,¹² which are a component of many medical electronic decision-support systems. Content-creation is typically performed through a forms-based graphical user interface. Such an interface can save the developer from having to worry about the physical layout of the metadata, as well as prevent a variety of data-entry errors.
2. A program may generate the tables, as in lexer/parser generators such as the Lex/Yacc combination,¹³ which are used to implement computing languages – whether full-fledged programming languages or the miniature languages used by scientific calculators. For those unfamiliar with this field, a *lexer* is a program that recognizes and classifies the tokens (the equivalent of words) that are encountered in a document (e.g., a file of program code) that uses the language. A *parser* is a program that receives the output of the lexer, and ensures that the tokens are put together in a way that conforms to the grammar of the language.

Tables enable a lexer to determine what tokens have a special meaning in the language, as opposed to something introduced by a programmer using the language. Parsers use tables to process a given sequence of tokens (typically by looking a certain number of tokens ahead) to determine if the current sequence is grammatical: if it is, the sequence itself is classified as belonging to one of several patterns permitted by the language.

3. A program computes values the first time it needs them, but then stores them and looks them up later when needed in order to avoid the expense of re-computation. This approach is used in the computational approach called *dynamic programming* originally described by Bellman,¹⁴ which is the basis for the original algorithms used in biological sequence comparison such as Needleman-Wunsch¹⁵ and Smith-Waterman.¹⁶ The more efficient heuristic algorithms such as BLAST,¹⁷ which trade less rigorous accuracy for vastly increased speed and are intended to compare a sequence against a database of sequences, also use a lookup-table representation of the test sequence to achieve their high performance.

Metadata-driven systems are essentially table-driven systems. The major difference is that there are typically multiple tables that address diverse sub-problems within the overall system. For example, the user-interface-generator component of an EMR or a CSDMS will use some tables to determine how certain clinical parameters are to be presented on the screen (e.g., as text boxes, check boxes, or pull-downs), while other tables will be brought to bear on the problem of validating the content of what the user enters and/or providing ergonomic aids to maximize the consistency of what is entered across multiple fields. Still other lookup tables will store rules for clinical decision support, and yet more tables facilitate the task of exporting data to external systems such as statistical packages or data marts.

Because metadata-driven systems typically contain multiple tables whose contents must be synchronized, it is not reasonable to expect humans to create metadata in their final format manually. Therefore the contents are typically created a mixture of approaches 1 and 2: that is, subject-matter experts specify the metadata using software, which then populates the tables. Such software typically uses a graphical user interface to assist rapid, easy and accurate entry.

1.6.2

Representation of Metadata: Guidelines

Throughout this book, where I consider applications that are data-centric, I strongly recommend the use of RDBMS technology to represent metadata. I recommend particularly that the metadata be stored in the same physical store (i.e., database) as the data that it seeks to describe. Specifically, I advocate a *sub-schema*: a set of tables that, for reasons that will be explained shortly, are protected in that only a very limited proportion of the system's users may modify their contents. There are certain situations where the metadata may be cloned to a separate data store (a so-called *metadata repository*) that stores the metadata of several systems. However, in the production systems that I will describe later, the contents of several fields in the data tables will link to the metadata tables, so it is simple to have both sets of tables within the same physical database.

The primary reason for my advocacy of RDBMS for data-centric applications is that in a large system, the structure of the metadata can rapidly become complex enough to mandate the use of a data model. In the systems that I will describe, the number of metadata tables is larger than the number of data tables by a factor of at least two or three. As I shall make clear later, certain database design solutions that are used in electronic medical record systems achieve simplicity and adaptability in *data* structure by trading off complexity in *metadata* structure. It is not advisable to attempt to manage a large number of disparate but inter-related units of metadata-related information from first principles. It makes more sense to utilize RDBMSs, which excel in such scenarios. (Historically, one of the first metadata-driven electronic medical record systems, TMR (=The Medical Record), invented at Duke University by Stead and Hammond¹⁸ in the pre-relational, pre-XML era, managed all metadata in a single file with a structure that was so complex that the creators trusted only themselves to edit it.)

A major advantage of an RDBMS-based metadata model is that there are numerous third-party tools available that facilitate the metadata's editing through a GUI. Due to

vigorous vendor competition, the tools that have survived have stood the test of time in terms of facilitating developer productivity.

Extensible Markup Language (XML), which I'll discuss in a subsequent chapter, has a number of important roles in metadata, but for now, I'll just state that managing significant amounts of highly diverse technical data is, in my opinion, not one of them.

1.7

Safeguarding Metadata

Given that metadata's correctness is so essential to the correct operation of a system, system designers must do their best to safeguard its integrity. In the case of RDBMS data catalogs, some of the catalog metadata, mostly structural information on tables, columns, inter-table relationships can be browsed in tabular form by those who are sufficiently knowledgeable. However, while the software developer is grateful for the existence of catalog metadata, s/he has no possibility of modifying it in any way. Indeed, given that developers are human and can make mistakes, the risks of system failure by the accidental introduction of an error or an inconsistency are far too great to allow such developer freedom. Catalog metadata can be read, but the only way to modify it is through indirect means – specifically, defining new structural elements or altering existing ones using SQL or the vendor's GUI.

When designing one's own metadata-driven system, the precautions that must be taken are similar. The metadata is typically allowed to be modified only through software, which must ensure that the metadata as a whole is consistent before and after the attempted metadata-change operation. It makes the developers' task significantly simpler if the technology used to implement metadata editing and maintenance has built-in support for operations such as:

- *Declarative Referential Integrity*: In a large system that stores diverse kinds of data, the individual data elements are interlinked. Thus, in an electronic medical record, a patient has multiple encounters, and each encounter has associated information such as the progress note or laboratory investigations. Referential Integrity implies that these elements (which “refer” to each other) must be consistent. That is, you can't have an encounter record for a patient unless that patient's record has been created first, and a laboratory value must also be associated with both a specific laboratory test definition as well as a patient to whom it applies. “Declarative” means that the technology lets a system designer specify an integrity association between two tables (e.g., lab-test values and patients) using either a short phrase at data-definition time, or through a GUI. That is, the designer “declares” one's intentions, and the technology obeys and does what is needed. (In the days before declarative ability, implementing referential integrity involved writing modestly elaborate case-by-case code.)
- *Transactions*: Transactions are a mechanism to maintain consistent system state by allowing the typically multiple steps in a data change operation to be treated as a single unit: either all of the steps must succeed, in which case the operation is “committed”, or

none do, in which case the operation is “rolled back”. Partial completion, where some of the steps succeed but others fail, leading to inconsistency, is not permitted. (In database parlance, this all-or-nothing property of transactions is called *atomicity*.) The consistent behavior is maintained even in circumstances such as hardware or power failure, which would cause roll-backs: this property is termed *durability*.

Both referential integrity and transaction support are built into modern RDBMS technology, which is yet another reason to use it.

1.8 Pitfalls in Metadata-driven System Development

Throughout this book, I will cite historical examples of missteps in the field of software, both biomedical and general (or things that could simply have been done better). The reason for this emphasis is George Santayana’s dictum: those who fail to learn the lessons of history are condemned to repeat it. Post-mortems are as educative here as they are in clinical medicine and forensics.

As we proceed through this book, it will become clear that metadata-driven systems are much harder to build than their hard-coded-logic equivalents (McConnell’s example is the very rare exception.) Creating such systems is difficult for two main reasons.

- Their implementation requires abstraction of the general problem and representation of logic in a computer-interpretable (ideally, structured and tabular) form. Getting the data structures right is an important aspect of problem solution.
- The GUI code must be robust enough to facilitate intuitive and reasonably error-free creation and/or modification of the tables’ contents by subject-matter experts who may not necessarily be programmers.

Much development effort will be spent in refining the interface, and improving error-prevention and diagnostics, based on the patterns of errors that you find your users making. While the rare user of stellar incompetence does exist, it is your responsibility not only to have your framework catch as many errors as possible, but to devise ways to prevent their occurrence through better usability design and adequate documentation and examples.

Metadata-driven development is not for everybody, or for all situations. I list some caveats here.

- *Software Requirements*: The main efforts in the creation of a metadata-driven system go into facilitating its intuitive as well as error-proof creation: this involves designing the metadata subschema and the metadata-creation interface. This work, which can be considerable, pays off only for systems that are intended to have longevity, where various aspects of it are expected to change significantly over time, and where the metadata is intended to be created mostly by non-programmers. *All* these criteria must exist to justify a decision to go with a metadata-driven approach.

- *Choice of Development Tools:* One lesson I have learned is that the metadata-creation GUI involves multiple cycles of iterative development with your power-user base. This up-front effort is significant enough that you should, to the extent possible, use rapid-application-development (RAD) software, which facilitates iterative development, to create it. Even as of 2009, the best Web-interface-creation development tools do not yield the same degree of productivity as the best two-tier client-server tools, which interact with a database backend over a network using non-HTTP mechanisms. Currently, the juxtaposition of the words “Web” and “RAD” is still an oxymoron.

While client-server tools are admittedly less scalable than those that run on Web platforms – and a Web-based program can be run from any networked machine with a Web browser on it – scalability may not be required in most circumstances because only a handful of power-users can be entrusted to modify the metadata.

This does not mean that Web front ends have no place in metadata-creation GUIs. For the typical metadata-driven systems that contain multiple metadata tables, creating interfaces to some types of metadata may be significantly simpler than for others. A similar form of reasoning applies to power-user skill level: some metadata is much easier and safer to modify than other kinds. In this case, it may be desirable to partition the metadata-creation interfaces: a Web-based toolset allows simple modifications where errors can also be readily detected and corrected, and is made accessible to a lower level of power-user, while only the most advanced and knowledgeable power-users have access to the client-server based tool.

- *The Nature of the Software Development Team:* Adopting a metadata-driven approach mandates the presence of a specific mindset in the team’s programmers, and definitely in the software project manager. It is not just a case of having more software-development talent. The team’s leader must think *long-term*: it is a case of working much harder now so that one does not have to work needlessly hard later. This trait is described in both McConnell’s treatise and in the well-known “Camel” book, “Programming Perl”¹⁹ as “creative laziness” – the creation of a framework that streamlines the mundane chores (the hard part), which can now be delegated to non-programmers, so that the developers can now relax and move on to new horizons (the pleasurable part).

A trait almost as important as the good kind of laziness is a sense of programming esthetics: metadata-driven systems must strive for both elegance and clarity. The constant striving toward these twin goals requires unrelenting focus: in our experience, once you commit to a metadata-driven approach to building a particular system, there is no going back. System code that reflects a mixture of table-driven, generalizable solutions to some sub-problems and quick-and-dirty, ad hoc solutions to others are, in our opinion, worse than those that are entirely ad hoc in design, because the persons who inherit such a system have no guidance as to what approach was used to solve which sub-problem. Such a mixture of code not only reflects the dashing of good intentions, but also a failure or absence of software project management: the quality of the code is left entirely to the discretion of individual programmers, and the system reveals the lack of a unifying architectural vision.

Quick-and-dirty approaches may have their place in toy, one-off programs, but not as the basis of non-trivial software systems, where only dirtiness is guaranteed. Over the lifetime of the software, system-maintenance chores will take much longer than they should have, effectively constituting a time-sink.

1.8.1

Making Metadata Comprehensible: Detailed Documentation

A fundamental requirement of metadata-driven systems is *transparency*. Such systems essentially operate at one or more levels of indirection: code consults metadata to determine what to do rather than doing it directly. Therefore metadata-driven systems are intrinsically harder to understand, at first glance, than their non-metadata-driven counterparts.

It follows that such systems must be documented much more thoroughly and clearly than the typical system. Tweaks to the design as new capabilities are deemed necessary are inevitable, and if the purpose of each and every existing component is clearly documented, modifications or enhancements can also be made efficaciously. In a subsequent chapter, I'll emphasize the importance of capturing the details of a database schema – including the metadata sub-schema – in metadata, so that the metadata describes itself.

In the absence of such documentation, the modestly hard problem of understanding the system soon becomes almost insurmountably hard. The consequences of absent or inadequate documentation become readily apparent. For example, it predisposes to coordination failure among software-development team members: numerous large-scale schemas (including those of EMRs) are littered with tables that aren't used any more, or multiple tables that perform the same function – not to mention redundant code.

Detailed metadata documentation, I believe, should be mandatory for any package that is sold to a customer for an appreciable sum, and which is maintained by a team locally. The design and contents of metadata must above all be understandable to *all* who need to work with the system's internals – not just the vendor. Unfortunately, the transparency principle is violated deliberately by many CRM and ERP packages that use metadata-driven approaches. The metadata structure (indeed, the entire schema design) has been deliberately obfuscated and non-documented for the sole purpose of keeping the design proprietary.

The consequence of such opacity is an incredible *inflexibility* – for many of these packages, one must modify the customer organization's processes to fit the software, because it is impossible to tailor the package to the needs of the organization. The only person who can possibly modify the metadata safely is the vendor's representative or a specially certified consultant, to whom the software is a source of continuing revenue.

I am continually astonished at how organizations have bet the bank on systems that they have no hope of comprehending even superficially – but not at all surprised when so many ERP/CRM efforts fail. Not all failures are due to failure to comprehend the system, but a large percentage of these are. If you don't know how your system works, you can't be sure that a vendor's or a consultant's promises to fix specific shortcomings have a chance of succeeding.

References

1. Wikipedia. Metadata. 2009 [cited 8/1/09]; Available from: <http://en.wikipedia.org/wiki/metadata>.
2. Marco D. *Building and Managing the Metadata Repository*. New York: Wiley; 2000.

3. Zimmer B. On language: we. *The New York Times*. 2010.
4. Codd E. A relational model for large shared data banks. *Commun ACM*. 1970;13(6): 377-387.
5. Melton J, Simon AR, Gray J. *SQL 1999: Understanding Relational Language Components*. San Mateo: Morgan Kaufman; 2001.
6. Martin J. *Computer Data-base Organization*. Englewood Cliffs: Prentice-Hall; 1977.
7. Martin J. *Application Development without Programmers*. Englewood Cliffs: Prentice-Hall; 1981.
8. Wikipedia. King Camp Gillette (biography). 2009 [cited 8/1/09]; Available from: http://en.wikipedia.org/wiki/King_Camp_Gillette.
9. Health Level Seven Inc. *HL7 Standard V2.3.1*. Ann Arbor: Health Level Seven Inc; 2000.
10. Kimball R, Reeves L, Ross M, Thornthwaite W. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses*. New York: Wiley; 1998.
11. McConnell S. Table-driven Methods. In: *Code Complete*. 2nd ed. Redmond: Microsoft; 2004.
12. Chapin N. An introduction to decision tables. *Data Process Manag Assoc Q*. 1967;6(3):2-33.
13. Levine JR, Mason T, Brown D. *Lex & Yacc*. Sebastopol: O'Reilly & Associates; 1992.
14. Bellman R. *Dynamic Programming*. Princeton: Princeton University Press; 1957. Dover paperback edition (2003).
15. Needleman S, Wunsch C. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*. 1970;48(3):448-453.
16. Smith TF, Waterman MS. Identification of common molecular subsequences. *J Mol Biol*. 1981;147:195-197.
17. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. *J Mol Biol*. 1990;215:403-410.
18. Stead WW, Hammond WE. Computer-based medical records: the centerpiece of TMR. *MD Comput*. 1988;5(5):48-62.
19. Wall L, Christiansen T, Orwant J. *Programming Perl*. 3rd ed. Sebastopol: O'Reilly; 2000.

This chapter partly recapitulates databases/statistics 101. The principles laid here are essential to appreciating the design decisions that are made for electronic medical records. It may be elementary to those with a database-design or statistics background, who may wish to skim rapidly or skip ahead to the very last section of this chapter, where I recommend and describe a specific approach for representing enumerated data types.

In the account below, I use the general term *parameter* to describe any finding that is determined from the history, clinical examination or through investigations such as radiology or laboratory tests. The topic of data types bears on how data for a given parameter is represented in computer storage. When values of a parameter are stored in a database, the unit of storage definition is called a *field* (or a *column*).

I will now describe the data types that are relevant to the Electronic Medical Record (EMR). This discussion also applies to Clinical Study Data Management Systems, which use an underlying design that is highly similar to EMRs.

2.1

The Categorical (Nominal) and Ordinal (Ranked) Data Types

RDBMSs define numeric data types by the valid range of numbers that a given type can hold and the precision to which it can be represented. For biomedical applications, however, it is necessary to consider the “scale of measurement”, a term originally introduced by Stanley Stevens in 1946,¹ which helps you decide what types of statistical tests can be applied to a given data type. Stevens classified scales as belonging to one of the following categories: nominal, ordinal, interval and ratio.

With a *categorical* or *nominal* data type, the parameter’s values fall into a limited number of categories: each category is given a particular name. While a given category may be associated with a numeric code, the only operation permitted between two values is comparison for equality/inequality. An example of an enumerated parameter is “Type of Surgical Procedure on Tumor” where the categories may be:

- 1/Biopsied,
- 2/Completely Resected,
- 3/Partially Resected,

4/Not Resected,
5/Not Found,
6/Not Applicable.

Statistical tests of significance are practically limited to the chi-squared test. The only statistical aggregate operations on nominal data are counts of individual categories: arithmetic operations on the numeric codes, or aggregates such as average, are meaningless.

An *ordinal* or *ranked* data type is similar to the categorical data type, but the individual categories can be ranked in some way. An example is Severity of Subjective Symptom, which is graded as: 0/Absent, 1/Mild, 2/Moderate, 3/Severe, 4/Life-threatening.

In addition to equality, two ordinal values can also be compared for relative magnitude. Note that for certain ordinal parameters, the descriptive name associated with a given numeric code need not be specified – only the extremities of the scale will be. For example, a patient can self-rate pain on a scale of 0/absent to 10/unbearable.

From the schema-design perspective, Nominal and Ordinal data types are often lumped into a broader category called the *Enumerated* data type. In the end-user interface, the individual categories for a given enumerated type are presented either as a pull-down menu (“combo box”, in Microsoft lingo), or as a set of radio buttons. Because the user has to select one item from the set presented, the set of categories for a given parameter is also called a *Choice Set*. I will later discuss how enumerated data are represented. For now, I will just state that, while values are stored as numbers (typically integers), representation of their definitions that allows interpretation of those numbers relies on metadata tables.

Mathematical operations such as differences or ratios are not permissible on ordinal values. Thus, a pain rating of 4 is *more* than a pain rating of 2, though it does not mean that the pain is twice as intense, or that the difference between 0 and 1 is the same as that between 1 and 2. The additional permissible statistical aggregates are the maximum, minimum and median. While arithmetic means have been used on such data, there is a controversy about their validity, as discussed by Michell.² In addition, numerous “non-parametric” statistical tests and measures have been devised specially for ranked data.

2.1.1

The Boolean Data Type

A special case of the Nominal data type is the **Logical** or **Boolean** data type (the word “Boolean” honors the nineteenth-century British mathematician George Boole). Here there are only two values permissible, Yes (“True”) and No (“False”). Many database engines have built-in support for the Boolean: in Microsoft Access, for example, Booleans are presented to the user as check boxes (checked=Yes).

I will discuss Booleans toward the end of this chapter, when we introduce the problem of representing missing values.

2.2 Numeric Data Types

A given numeric parameter may have positive, zero or negative values. For certain parameters (such as weight or height) negative values are meaningless. From the statistical perspective, numeric data is classified in one of two ways.

1. In *interval* data, the zero is chosen arbitrarily, so that a value twice as large does not necessarily imply double the magnitude. In other words, ratios are meaningless: an example is temperature as measured in degrees Fahrenheit.
2. In *ratio* data (such as weight or degrees Kelvin), on the other hand, the zero is absolute, so that ratios are meaningful.

The additional statistical aggregates that apply to interval and ratio data are identical: average, standard deviation. Most of the traditional statistical tests of significance assume that the parameter is at least interval in nature.

We now consider numeric data types from the perspective of computer storage. Numeric data are generally thought of as “numbers used in the real world” and are of two types:

1. *integers*: non-fraction whole numbers, i.e., 13
2. *real*: rational numbers having a fractional portions represented as floating-point or decimal numbers, i.e., 628.37

2.2.1 Integers

Integers are the most widely used numeric data type. Most database engines allow specification of the maximum size of an integer: this influences compactness of storage. Microsoft SQL Server, for example, allows 1, 2, 4, or 8 bytes for an integer, the respective data types being designated as tinyint, smallint, int and bigint. The bigint can represent a number as large as $\pm 2^{63}-1$, while the tinyint represents a number between 0 and 255.

4-byte and 8-byte integers are often used for *auto-incrementing machine-generated unique identifiers* (also called *auto-number* fields or *UID* fields: UID=unique identifier). UID fields are widely used in databases, because they allow a means of identifying a record in a database table uniquely: in database lingo, the UID field serves as an *artificial primary key*.

Most database engines simply allow you to designate a field as an auto-number: they will take care of automatically generating a new number for a newly created record. The UID field in each record is guaranteed to be unique, and the UID generated will never be reused if that record is later deleted.

You needn't worry about “holes” in the sequence of records: 2^{63} is an enormous number, and all that matters is that a record with a larger UID value was created more recently

than a record with a smaller UID value. Obviously, you don't do arithmetic with UIDs, so from the statistics viewpoint, they are ranked data.

The UID field's value is not allowed to be edited by the user (or even, interactively, by developers and administrators). This never-changing property turns out to be very useful, because the ID is typically created so as to be used in other tables, where it serves as a compact foreign key. Modern relational databases support a feature called *referential integrity*. Here, if you attempt to delete a record in a table whose primary key is used in other tables, then one of two things can happen, based on how an individual link has been set up:

1. If the related table contains rows with the value of the primary key that you are trying to delete, deletion of the record in the primary-key table will be prevented, to forestall "orphaning".
2. Very rarely, the database designer may specify that if the row in the primary-key table is deleted, all rows with this value in the second table should be automatically deleted: this is called a *cascading delete*. Cascading deletes are dangerous, but may be justifiable in special circumstances (e.g., when the contents of the table are populated by programs rather than user interaction).

Similarly, attempted changes (updates) to the primary key value will either be prohibited, or the changes will cascade to other tables.

I should emphasize that referential integrity is a feature that is independent of whether or not you are using UIDs. However, the more compact the key's value, the more efficiently integrity checks will run. Also, with UIDs, the cascading update issue is something you don't have to worry about.

2.2.1.1

Universally Unique Identifiers

The problem with large-integer UIDs is that they are unique (and hence meaningful) only to a particular instance of a database. (If two separate organizations that have been using large-integer UIDs merge, and eventually merge their product lines and electronic systems, the UIDs have to be regenerated, and all the data dependent on them also fixed.)

There are circumstances where multiple machines working independently of each other need to generate artificial primary keys that will later be merged together into a common dataset, but where a guarantee needs to be provided that each UID will still be unique. The Open Software Foundation has specified algorithms for generating such identifiers, called Universally Unique identifiers (UUIDs): Microsoft's implementation of the UUID, employed in a variety of their products, is called the GUID (Globally Unique Identifier).

The UUID/GUID is a 128-bit number normally expressed as a hyphenated 32-character hexadecimal string. There are thus 2^{128} or 10^{38} possible UUIDs. Among the techniques for creating UUIDs are pseudo-random-number generators or message digests (see later) of a URL; earlier approaches used a combination of the Media Access Control (Network) address of the computer on which the algorithm ran, the time on the computer's clock (at

A *message digest* is a cryptographic transform of some data – this could be the contents of a message, or even a large document – which is typically much smaller in length than the original message, but still fairly lengthy – e.g., 256, 384 or 512 bits long: it has the properties that

- The chance that two random messages will produce an identical digest are extremely small.
- Given a digest, it is not possible to reconstruct the original message – such algorithms are termed “trap-door” or “one-way” algorithms.
- Even a small change in the original message – e.g., alteration of a single bit – will create a dramatic difference in the resulting digest.

Thus, a digest of a message (sent separately from the message) can be checked to ensure that the original message has not been tampered with: message digests thus form the basis of *digital signatures*. The algorithms for generating message digests are being continuously refined to become more secure: the latest family of digest algorithms is the SHA-2 (Secure Hash Algorithm-2).

an accuracy of 100 ns) and a random number: this algorithm was vulnerable to a *privacy leak*, since given a UUID, the machine could be identified.

2.2.2

Real Numbers

Real numbers are numbers with digits after the decimal point. They can either be represented to a maximum number of digits before and after the decimal point (the *decimal* representation), or to a maximum number of *significant figures*, irrespective of how large or small the number is (the *floating-point* representation).

The floating-point representation is based on converting a number, no matter how large or small, into a mantissa (a number between 0 and 1, precise to a certain number of significant figures) and an exponent. Thus, if we could record a maximum of eight significant figures, the number negative 24,587,489,123 would be represented, conceptually as $-0.24587489 \times 10^{12}$, though the internal computer representation is in terms of base-2 (binary) rather than base-10 (decimal) as shown here. The computer stores real data to the precision allocated for the storage, which is mostly either 4 or 8 bytes for data types *single precision* and *double precision* respectively.

Some database engines, notably Oracle, support only the decimal representation: Microsoft SQL Server supports both.

Decimal numbers are more widely used in business applications (where two or three decimal places are used), while floating point is more convenient for scientific applications, where very large as well as very small numbers must be represented (and often multiplied or divided). The decimal representation is the one that is more useful for biomedical applications other than those dealing with imaging.

In particular, real numbers that must be presented to and edited by the user for most medical applications are best represented decimally. This is because certain decimal numbers, such as 0.2, cannot be represented exactly in the floating point representation, just as the fraction $1/3$ cannot be represented exactly as a decimal number. Within computer storage, are converted into infinite recurring decimals (i.e., 0.19999999...): While an error in the 10th decimal place means that the distance from the earth to the sun is miscalculated by less 170 meters, presenting such values to a user may cause puzzlement unless you write code to round it to a certain number of decimal places. Using a decimal representation forestalls this issue. David Goldberg's widely reproduced article³ provides a detailed, if somewhat technical, article on the challenges of dealing with floating-point; Dan Clarke's article⁴ is somewhat more accessible.

2.3

The Date Data Type

The 'date' data type is used for fixing a time component to a data record. Modern database engines allow dates to be stored with or without a time part (e.g., 12/01/2008 4:31:34 PM). The range of dates, and the precision to which the fractional part of the time can be stored, depends on the database engine: some will offer you a variety of date/time types, and will also allow you to specify the time-zone as a positive or negative numeric offset from Coordinated Universal Time (i.e., Greenwich Mean Time). The offset is particularly important in a database that is being accessed over the Internet across several time zones.

A date data type comes with built-in validation and operations/functions. For example, users are prevented from entering the value "Feb 29th" for a non-leap year. Similarly, *date arithmetic* becomes possible. For example, one can:

1. Subtract two dates to return a number in one's time-interval unit of choice, such as hours, months, days, or weeks.
2. Add or subtract a time-unit interval from a date to compute another date.
3. Determine the day of week for a given date.
4. Decompose a date-time into its component parts: year, month, etc.

It's fair to say, however, that the number of built-in operations depends a lot on the DBMS you are working with: some DBMSs are richer than others. Interestingly, most DBMSs will let you perform simple arithmetic without the use of special functions: thus, adding 1 to today's date using the will give tomorrow's date, while subtracting two dates using the minus sign will give the difference in days.

It is worth noting that the root cause of the notorious "Year 2000 problem" was that COBOL, the language in which the vulnerable software had been developed, lacked the date data type and consequently did not support built-in date arithmetic: COBOL developers were dependent on (non-standardized) third-party software libraries for this purpose, many of which assumed that dates would be stored with only two digits for the year (thus implicitly assuming that all dates applied to the twentieth century).

2.4

The Text Data Type

Textual data can contain any characters from the keyboard – letters, numbers, or punctuation. It is represented in one of two ways: *short* text, where the maximum length is reasonably modest (this size limit varies from 255 to 8,000 characters, depending on the vendor whose database engine is being used for storage) and *long* text, where the upper limit on length is much larger (e.g., 4 GB).

The short text data type, which is given different names in different database engines, is used for fields like Name, Street Address and City, Long text is used to store text that contains embedded markup that is also textual – examples of this marked-up text are HTML and XML. (I discuss XML in the next chapter.)

The differences between short and long text have to do with efficiency of storage and search. Historically, database engines supported only short text at first. Storage for short text fields is much more compact and efficient: they can be indexed for fast search. Certain database engines allow the creation of full-text indexes on long text (analogous to the indexes that search engines such as Google create on the contents of Web pages), but others do not.

2.4.1

Pseudo-Numeric Data

Certain types of data, such as phone numbers or social security numbers, may superficially seem to be numeric, but because they often contain leading zeros (as well as punctuation for formatting purposes), they should be stored as short text, not as integers. (Further, phone numbers that contain an optional extension would be problematic if represented as numeric.)

Similar considerations apply to quasi-numeric codes that may use more than one decimal point. Examples are Internet addresses in numeric form, which have three decimal places (e.g., 128.38.66.216).

2.4.2

Using Text to Store Data of Another Intrinsic Type: A Warning

Nominal, ordinal, numeric or date data can be stored as short-text. Certain data models for the Electronic Medical Record as well as Clinical Study Data Management Systems such as Oracle Clinical™ actually do this, ostensibly for the purpose of simplifying their data model. I prefer to avoid this approach because the DBMS's built-in operations and validation for these data types become unavailable, so that the application has to do extra work needlessly.

It is important to note that when numbers are stored as text, using ASCII or Unicode, the nature of comparisons between two numbers is not what you would expect. Thus,

while the number 98 is *less* than 1,245, the text “98” is *greater than* “1,245”, because text is compared using the leftmost character/s. Thus, if you query a text column that is intended to store numbers, and ask for all values between 93 and 99, the results returned will be nonsensical: you will get back values such as “9,456”. The query should really be:

```
select columnName from tablename where  
    convert (columnName, integer) between 93 and 99
```

where *convert()* is the function your DBMS provides to convert text to another data type.

To run this query, the DBMS must convert every value in the table on the fly using the *convert()* function in order to get the results you need. This is inefficient when the table being searched has thousands or millions of values. For the same reason, most indexes on such columns, which are also used for range searches, are useless. In some DBMSs, you can index on a function of a column’s values rather than on the values themselves, but if you’re going to do this, there’s no reason not to use the appropriate data type to begin with.

2.5

The Binary Data Type

The binary data type is used for signal data such as images, though it can also be used for data formats such as .DOC or .XLS (used respectively by Word and Excel). The word “binary” comes from the fact that historically, this data was not human-readable, and was simply treated by the database as a collection of 1s and 0s (i.e., binary data, which is how all data is ultimately represented in the computer).

The binary data type typically has the same size limit as long text. You should note that many databases don’t do anything with binary data, other than storing it and fetching it. It is up to the application software to make sense of it.

Until recently, the storage of binary and long-text data types in RDBMSs was very poorly optimized: the contents were sliced on disk into chunks that varied in size from 2–8 K. Each chunk had some data that “pointed” to the next chunk, and the chunks were not necessarily physically contiguous, so that, if you had to fetch a 100 K unit of data, and the DBMS used 2 K chunks, it had to assemble 50 units from disk. In these days of multi-megabyte multimedia, such an assembly process is needlessly inefficient, especially if the contents are fragmented. Also, because the DBMS manages its storage independently of the operating system, utilities such as disk defragmenters do not help improve performance.

Consequently, many software developers used the DBMS only to record the path to an external file: the danger here was that the file was unprotected by the DBMS and could be accessed by unauthorized individuals, accidentally altered or deleted. MS SQL Server 2008 finally addressed the problem by giving you the best of both worlds: letting you store binary content as external files but at the same time putting those files under DBMS control.

2.6

Custom (User-Defined) Data Types

Modern relational database management systems (RDBMSs) allow developers to define their own data types. For a long time, these were little more than conveniences that helped to avoid repetitive typing (when used in multiple places). For example, one could define a subtype of integer that had to be non-null and in the range 0–100, or a subtype of string that was six characters long and could only contain the letters A–Z. Even such a basic use has much to recommend it: if, for some reason, the definition of such a data type has to change, it needs to change only in one place: as a result, a database that uses custom data types is more modular and maintainable.

With present-generation RDBMSs, one can achieve greater sophistication, defining data types using a modern programming language (e.g., Java or a .Net language) that not only embody validation constraints but also specify a set of operations that apply only to that data type.

Thus, one could create a data type called “GIF_Image” or “Fingerprint”. For the fingerprint data type, which would be used in forensic applications, one could define an operation that, given a fingerprint, returned a classification code that allowed rapid search of a fingerprint database for candidate matches. A source-code example of a non-trivial custom data type – the Complex data type used in mathematics – is provided in the text by Itzik Ben-Gan et al. on SQL Server programming⁵.

Custom-built data types are often distributed commercially as software libraries, or are provided by the vendor. The XML data type mentioned earlier has appeared in database engines only in the last few years: it is an example of a custom data type built by the vendor on top of the long-text data type.

2.7

Structured vs. Unstructured Data

Long text and binary data represent *unstructured* data: the other data types discussed above are *structured*. As we’ll see, the goal of designing an EMR is to maximize the proportion of structured data. However, some unstructured data is unavoidable. Images and signal data such as electrocardiography data are intrinsically “binary”. Long text is also the most convenient means of recording the status of a patient, as when recording a surgical note or a discharge summary that is intended to be passed on to the patient’s primary care provider.

Until the late 1990s, unstructured data was something of a step-child in RDBMSs. After the advent of the full-text search engines – Alta Vista, Yahoo and eventually Google – it could not be ignored anymore and RDBMSs now actually let you store long text, index it and search it usefully. However, this is one area where vendor offerings differ greatly both with respect to capabilities and even syntax. While the ISO SQL-1999 standard⁶ made some very modest recommendations regarding such a feature set, every vendor has gone their own way.

2.8
Representing Enumerated Data

I will now switch from describing common knowledge to detailing a specific approach that is used in later chapters of this book. This approach is not widely publicized, though many designers of complex biomedical schemas are aware of it and use it routinely. It would seem to be non-controversial, but I’ve encountered what I consider technically obsolescent arguments against using it. The appendix that follows this section is intended to counter these arguments and is intended for database developers who are familiar with the use and creation of stored functions. (While it uses SQL Server syntax, it is trivial to translate into an alternative SQL dialect such as Oracle PL/SQL.)

Each category in an enumerated parameter consists of a pair of values: a *code* (which is typically numeric) and an associated *description*. (One may also add a third item, a *long description*, in case the description is too terse.) The set of codes and descriptions for a parameter is called a *domain*.

2.8.1
The Older Approach

The older approach to modeling an enumerated data type is to create one table for each set of code-description pairs. This is illustrated in Fig. 2.1:

The table *Symptom_Severity_Definition* has two columns, Code and Description. It contains five rows, whose values range from 0 (Absent) to 4 (Life-threatening). The second table in the figure, *Patient_Symptom_Severity*, which contains patient data that uses this definition, is shown merely for illustration purposes. It may contain numerous

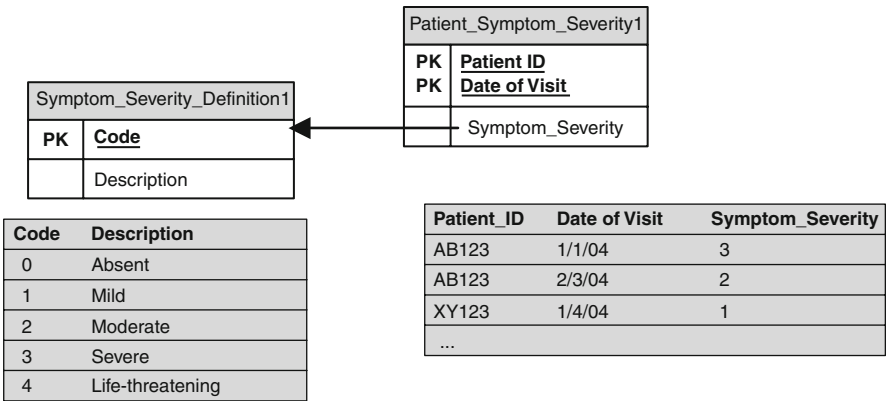


Fig. 2.1 Representing Enumerated Parameters using one table for each parameter definition. The upper part of the diagram shows the table definitions, the lower part shows the tables’ contents. The *Patient_Symptom_Severity* table contains the actual data for individual patients, and records only a numeric code (the *Symptom_Severity* field) for a patient at a given date/time. The *Symptom_Severity_Definition* table contains the rows that define the meaning of each numeric code

columns (and thousands of rows, for individual patients whose symptom severity was recorded on various days), but the one column that we are concerned with, `Symptom_Severity`, is an integer column that is a foreign key into the first table. Thus, for a particular patient and a particular day, if we find a Severity of 3, we look up its meaning in the Definition table, which allows us to interpret it as Severe.

The drawback of this design approach is that, in a complex database design (as in EMRs and certain Enterprise-Resource-Planning systems), literally hundreds of definition tables need to be created in the database. For example, you need individual tables for Type of Surgery, Outcome of Surgery, Type of Radiotherapy, Pain Intensity, and so on. This leads to the following problems:

- A large database is maintained by a team of developers. As in all software projects, there is personnel turnover: one of the fundamental tasks of new personnel is to understand the database design. The more numerous the tables in the database, the more the effort required to understand it. In the database diagram, a definition table that contains only four to five rows is as prominent as another table that may contain several million rows, and special naming conventions are needed to indicate that certain tables serve the purpose of storing definitions of enumerated parameters.
- The enumerated-type definitions are *metadata*. One purpose of this book is to convince database designers that metadata should be concentrated in as few tables as reasonable, with a clean separation between data and metadata. I believe that there's something fundamentally wrong with a design approach where three-quarters or more of the tables in the database contain only enumerated-parameter definitions: in my opinion, these separate definitions litter the database schema like pigeon droppings.
- The definitions are not static: they may occasionally need to change. The individuals responsible for maintaining the domain values are content experts rather than database developers. These persons must be given read-write access to the definitions, while most other users of the system should have read-only access.

2.8.2

The Recommended Approach: Consolidating Definitions

The approach that I recommend is illustrated in Fig. 2.2. Here, the definitions of all enumerations are stored in three tables:

1. **Choice_Sets** contains one record for each enumerated data type: it has two fields, `Set_ID` (a UID) and a descriptive `Name`. This table defines all domains used in the system.
2. **Choice_Values** contains one row for each category in the Choice Set, defined by a `Code` and `Description`: the `Set_ID` column references the `Choice_Sets` table.
3. **Field_Definitions** records which Choice Set is used for a particular Field in a particular Table. As we will see in a subsequent chapter, this table's role is not limited to documenting that a particular field is an enumerated data type: it is an essential component of almost any metadata-driven database design. For now, however, the three columns we are concerned with are: *Tablename*, *FieldName* and *Set_ID*.

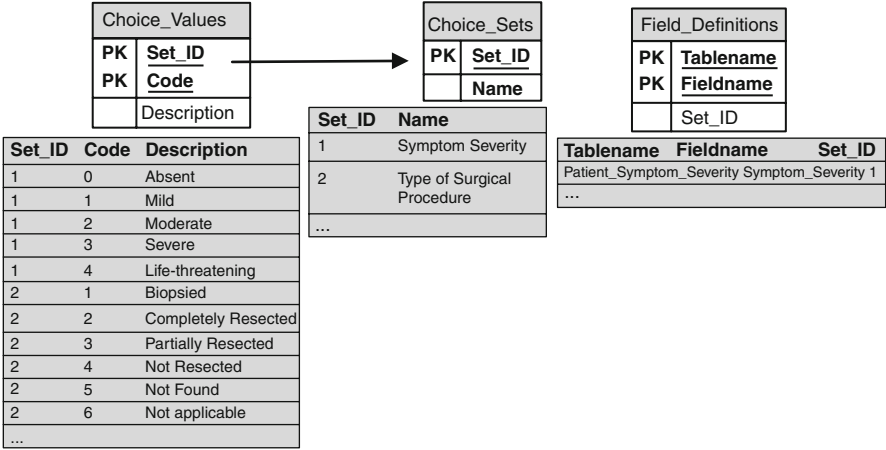


Fig. 2.2 Consolidating the definitions of enumerated parameters. The table **Choice_Sets** contains one row for each enumerated data type (defined with an ID and a Name). The table **Choice_Values** contains one row for each category in each enumerated data type: here, the data for Subjective Symptom Severity (Ser ID=1) and Type of Surgical Procedure (Set ID=2) are shown. The **Field Definitions** table records which Set is used for a given Field in a given table

Note that, with this design, the same set of enumerated values can be used for multiple parameters. This commonly occurs where a ranked Likert scale applies to multiple fields in a questionnaire, e.g.,

- 1=strongly disagree
- 2=disagree somewhat
- 3=neutral
- 4=agree somewhat
- 5=strongly agree

The additional benefits of the consolidated design are:

- It simplifies the creation of user-interface generation code, or the use of generic routines to present these values in the interface. Such code has to look only in one place to consult metadata, which is then used to populate the contents of a control such as a combo-box or radio-button-list.
- It simplifies the administration of content maintenance.: the content-curators can be given read-write access to these three tables, rather than the database administrator worrying about selective access to hundreds of tables. A single editing user-interface can be built for the **Choice_Sets** and **Choice_Values** tables.
- Reuse of enumerated categories is facilitated because these can be searched in a single place.

The job of a developer who has to build a domain-maintenance user interface is simplified considerably. The Domain/Domain-value tables can be presented for editing through a

simple form/sub-form interface. A given domain can be searched by a keyword that is part of its description, or by the Table or Column that maps to it, and its associated values can then be edited, or new values added.

2.8.3

Missing Values: Nulls

For any of the data types mentioned above, relational database engines have the capacity to represent an “unknown” or Null value. By default, Null capability is enabled for any column in a table, but the developer can turn it off selectively (for any field other than a UID field, which, as I’ve already stated, is not user-edited). Null capability means that a user may leave that field in a record *empty*: if it is turned off, a value must be entered.

For a Boolean field, if Nulls are enabled, they have the effect of creating *three-valued logic*, where the field may take one of three values: Yes, No and Null: the last can be interpreted as Unknown. This would seem to make intuitive sense: for example, a clinical history parameter like “Are you allergic to strawberries” may have three possible responses: Yes, No and Don’t Know.

However, things are not always so simple, because in certain circumstances, “Unknown” may have multiple possible meanings. For example:

- The response to a very personal question may have four possible answers: Yes, No, Don’t Know and Refuse to Answer.
- If the Date of Birth for a patient is not recorded, is this because no one remembered to ask for it, because the patient refused to provide the answer, or because the patient was unable to answer? (The last may happen for a young child whose guardian is unavailable, a person who does not speak the language of the care provider, or someone who has been brought in incoherent or unconscious.)
- For certain questions, the response may be “not applicable” (as in a possible response to “Have you stopped beating your wife?”).

In general, if you need to distinguish between the different kinds of unknown responses – and sooner or later you will – don’t use Boolean fields with a Null option. Instead, create a Nominal data type and define the possible unknown/missing values explicitly.

It is not too difficult to create a set of “universal” missing values – a special choice set using the design of Fig. 2.2 – that can be used across an application: the individual codes are typically assigned large negative integer values to indicate that their use should be exceptional.

Appendix: Ensuring Referential Integrity with a Consolidated-Definition Design

Davidson⁷ criticizes the consolidated-design approach, though in his example he uses a non-normalized single-table design rather than the three-table design illustrated above. The argument against consolidating is that there is no referential-integrity check that

prevents incorrect values from being entered. For example, even though symptom severity is supposed to range only from zero to four, one could try to bulk-import patient data from another system, where an incorrectly entered severity of 6 would not be caught. (In an end-user interface, this would not be an issue, since the definitions serve as the basis of pull-down lists which present only the appropriate codes and descriptions.)

This argument may have been valid during an era where RDBMSs did not have stored-function capability. It has ceased to be valid since 2000, by which time the big three (IBM DB2, Oracle, MS SQL Server) all implemented stored functions.

The following general-purpose stored function, **CheckEnumeratedCol**, (illustrated with MS SQL Server syntax) can be used to constrain *any* column to belong to a specific enumeration.

```

/* Code to create the definition tables */

CREATE TABLE Choice_Sets
  Set_ID int IDENTITY(1,1) NOT NULL primary key,
  NAME varchar(50) NOT NULL, DESCRIPTION varchar(255) NULL
)
go

CREATE TABLE Choice_Values (
  Set_ID int NOT NULL,
  CODE smallint not null,
  DESCRIPTION varchar(200) NULL,
  CONSTRAINT PRIM_Choice_Values PRIMARY KEY CLUSTERED (Set_ID, Code) )
GO

Create table Field_Definitions (
  Set_ID int not null,
  Tablename varchar(255) not null,
  FieldName varchar(255) not null,
  constraint Prim_Dom_Map Primary Key Clustered (Set_ID, Tablename, FieldName)
)
go

ALTER TABLE Choice_Values ADD CONSTRAINT FKEY1 FOREIGN KEY (Set_ID)
  REFERENCES Choice_Sets (Set_ID) ON DELETE CASCADE
GO

/* create a single Choice set for Symptom Severity */
insert into Choice_Sets (name, description) values('Symptom Severity Scale, 'Ordinal Measure of
Severity of Clinical Symptoms')
go
/* the above record gets the primary key value 1 */
insert into Choice_Values(Set_ID,Code,Description) values (1,0,'Absent');
insert into Choice_Values(Set_ID,Code,Description) values (1,1,'Mild');
insert into Choice_Values(Set_ID,Code,Description) values (1,2,'Moderate');
insert into Choice_Values(Set_ID,Code,Description) values (1,3,'Severe');
insert into Choice_Values(Set_ID,Code,Description) values (1,4,'Life-threatening');
go

/* create the Patient_Symptom_Severity table (see fig. 1) for recording patient data - */
create table Patient_Symptom_Severity (
  patient_ID varchar(20),
  Date_of_Visit datetime,
  Symptom_Severity smallint)

```

```

go
/* record the use of Set ID 1 for the column Symptom_Severity in the above table */
insert into Field_Definitions values(1, 'Patient_Symptom_Severity', 'Symptom_Severity')
go
/* create the general-purpose function that allows validation */
CREATE FUNCTION CheckEnumeratedCol (@Val smallint,@Tablename varchar(255),
@ColumnName varchar(255) ) RETURNS bit AS
BEGIN
DECLARE @retval int
DECLARE @Set_ID int

Select @Set_ID =Set_ID from Field_Definitions
where Tablename = @Tablename and FieldName = @ColumnName

SELECT @retval = COUNT(*) FROM Choice_Values DV
where Code = @Val and Set_ID= @Set_ID

RETURN @retval
END;
GO

/* associate the above function with the Symptom_Severity column as a constraint */
ALTER TABLE Patient_Symptom_Severity ADD CONSTRAINT chkSeverity CHECK
(dbo.CheckEnumeratedCol (Symptom_Severity, 'Patient_Symptom_Severity', 'Symptom_Severity') = 1);
GO
/* now try deliberately entering a bad value for the Symptom_Severity column*/
insert into Patient_Symptom_Severity values('ABC',convert (datetime,'10/2/2009'), 8)
/* this will fail */

```

Note that the Set_ID itself is not a parameter to the function CheckEnumeratedCol: this value is determined by looking up the Field_Definitions table which has been populated with the appropriate data. Therefore all that the function needs to constrain a given enumerated column is the column name without quotes, the table name to which it belongs in quotes, and the column name in quotes. Such a definition makes it hard to make a mistake, provided that Field_Definitions has been correctly populated.

In a developer interface, one would have to make the selection of the Set ID in the Field_Definitions table user-friendly by letting the developer browse the Choice Sets and Choice Values tables.

References

1. Stevens SS. On the theory of scales of measurement. *Science*. 1946;103(2684):677-680.
2. Michell J. Measurement: a beginner's guide. *J Appl Meas*. 2003;4(4):298-308.
3. Goldberg D. What every computer scientist should know about floating-point arithmetic computing surveys. *ACM Comput Surv*. 1991;23(1):1-44.
4. Clarke D. The trouble with rounding floating point numbers. *The Register*. London; 2006.
5. Ben-Gan I, Sarka D, Wolter R. *Inside SQL Server 2005: T-SQL Programming*. Redmond: Microsoft; 2005.
6. Melton J, Simon AR, Gray J. *SQL 1999: Understanding Relational Language Components*. San Mateo: Morgan Kaufman; 2001.
7. Davidson L. Ten common database design mistakes. 2007 [cited 1/5/09]; Available from: <http://www.simple-talk.com/sql/database-administration/ten-common-database-design-mistakes/>.

3.1

Introduction to XML

I will try and keep this section brief, and reasonably simplified: more extensive introductions, such as the Wikipedia entry, are freely available. Software developers who have worked with XML may wish to skip this section.

XML (eXtensible Markup Language) is a set of syntactic rules for embedding markup in a document: the important characteristic of this markup is that it follows a *hierarchical* structure that can be nested to an arbitrary depth of complexity. (As I'll discuss later, the markup can serve numerous purposes.) The markup consists of *begin-tags* and *end-tags*, which together enclose a unit of information called an *element*.

A *begin-tag* has the structure

```
<SomeToken attribute1="value1" attribute2="value2" ....>
```

where *SomeToken*, *attribute1*, *attribute2* are all single words (without spaces or punctuation) and *value1*, *value2*, etc. are textual content within the quotes. The attribute-value pairs may not always be present: below, we use the convention of using square brackets to indicate their optional nature.

An *end-tag* has the structure:

```
</SomeToken>
```

where *SomeToken* has been used earlier in a *begin-tag*.

Between a *begin-tag* and *end-tag* there can be textual content (without quotes), which may itself contain additional tags, nested to any depth.

Sometimes, a *begin-tag* can be its own *end-tag* as well: in this case, it has the structure.

```
<SomeToken [attribute-value list]/>
```

This structure can also be used to indicate that the tag has no embedded content.

Comments are indicated within the tags `<!--` and `-->`.

```
<!--this is a comment -->
```

Comments are content that is not interpreted as data and are used to provide explanation.

An example of XML content is shown in Fig. 3.1 below:

```
<STUDENT_LIST>
<STUDENT First_Name="John" Last_Name="Miller" SSN="012-09-8765">
  <PHONE_LIST>
    <PHONE Phone_Type="Home"> 212-987-7665 </PHONE>
    <PHONE Phone_Type="Mobile"> 212-764-1346 </PHONE>
  <PHONE_LIST>
</STUDENT>
<STUDENT First_Name="Susan" Last_Name="Chang" SSN="014-02-1543 ">
  <PHONE_LIST /> <!--No phone numbers listed -->
</STUDENT>
</STUDENT_LIST>
```

Fig. 3.1 Sample XML content

Either attributes or tags can be used to represent atomic data: the decision to use one or the other is arbitrary and depends on the document’s author. For example, the `First_Name` and `Last_Name` attributes could be represented as nested tags within `STUDENT`, i.e.,

```
<STUDENT>
  <First_Name>John</First_Name>
  <Last_Name>Miller</Last_Name>
  ...
</STUDENT>
```

Attributes are less verbose than tags, but obviously cannot be used for data that has sub-structure.

XML tags and the attribute names within them are *case-sensitive*: e.g., `<STUDENT>` is considered different from `<Student>`. Case-sensitivity could cause major problems if XML documents are composed by hand in a text editor – which they were in the old days – but nowadays, this is less of an issue, because the documents are generally created with software assistance.

Well-formed XML content is content that is syntactically correct with respect to XML’s rules. Specifically, the begin-tags and end-tags must always match up and be nested correctly. In addition, the document must not contain invalid characters: for example the symbol “<” cannot occur by itself because it has a special use in tags: instead, you use the HTML/XML symbol “<” (less than).

For those of you who are familiar with HTML, you can see that XML resembles HTML (and was greatly influenced by it). The differences between HTML and XML are:

- With HTML, the set of tags is fixed, while with XML, you can invent your own tags for your own purposes: hence the word “extensible”. At this point, however, we must distinguish between *well-formed* XML content and *valid* XML content. Since a particular XML document is created with a specific purpose, it is highly desirable (in some cases, mandatory) to first define the ground rules indicating what tags you will use in a given document (or set of documents), what the permissible attributes for a tag are, how tags will be nested within each other, and so on.

The “ground rules” specification is created in an ancillary document called an *XML Schema Document* (XSD).¹ These rules, for example, would flag a tag like `<LASAGNA>` as an error in the sample document above, which is intended to deal with student information. Further, one can specify validation on individual elements, which can belong to specific data types that closely mirror the structured data types in databases. Validation additionally includes features such as enumerations, range checking, string length, and regular expressions, as well as elements that cannot be omitted or empty. (For example, one may mandate that in the `StudentList` example above, the `SSN` and `First_Name` attributes cannot be omitted, and that the `SSN` attribute match the pattern `999-99-9999` where “9” represents a digit.) In other words, valid XML content is that which is both well-formed and confirms to the specification in an XSD.

- When HTML is processed by a Web browser, the processing is very “forgiving” in that the browser will still try to render a Web page even if it is not well-formed. This was a decision from the days when Web page designers composed HTML by hand, and often made mistakes. (As part of its forgiving nature, HTML tags are not case-sensitive.) If browsers were not forgiving, the vast majority of the world’s Web pages would not work. XML, by contrast, has much broader uses, including information exchange, and so validity failures must be caught. So most XML processors are unforgiving in that they will tell you what error has occurred and at what line in the document: we will shortly see how the syntax of XML facilitates accurate error diagnostics.

Developer Note: These days, most developers use software to create HTML pages, and such software typically generates HTML that is actually well-formed XML: the “stricter” specification of HTML, which is XML-conformant, has been given a special name, XHTML, the “X” indicating that it is a flavor of XML. “Strict” XHTML may seem to be a nuisance that merely increases the Web page designer’s efforts. However, if you’ve ever seen pages where the latter half of the text is in a much larger, bolded font compared to the first half – a page on the New York Times website once exhibited this problem – because somebody forgot to include an end-tag, you’ll appreciate that Web page-creation software can now tell you about such errors when you are creating the page, rather than having end-users discover such errors when the page is posted on to a production Website.

3.2 Strengths of XML

XML has become one of the buzzwords of the early twenty-first century: almost every software organization supports it in some form or another, and consequently, knowledge about it has become essential for serious developers. There are several factors that are responsible for the momentum behind XML.

- XML has a very small set of syntactic rules, so its *principles* are easy to understand, even for someone who is not an advanced programmer. Its syntax may appear familiar even to those who encounter it for the first time, in part due to the ubiquity of HTML.
- XML can support arbitrarily nested (i.e., hierarchical) information: such information is common in numerous real-world scenarios. Prior to XML, the format used for textual representation of hierarchically structured information – especially for applications like networks/telecommunications – was Abstract Syntax Notation 1² (ASN.1), which uses curly braces to indicate nesting. XML has the advantage over ASN.1 of a *fully bracketed syntax*. That is, begin-tags must always be matched by corresponding end-tags (or the begin-tag must be its own end-tag, as stated earlier.) The strengths of fully-bracketed syntax over braces are:
 - Content with fully bracketed syntax is somewhat easier for humans to follow than nested braces: while both forms of content require the use of indentation to assist readability, with braces one must mentally match the end-brace to the content preceding the start-brace.
 - In situations where there is a possibility of the content being erroneous (which can happen during transmission of content) fully-bracketed syntax is more robust for trapping errors. If an ASN.1 document has one opening or closing curly brace too few (or too many), it is very hard to pinpoint exactly where the problem lies. All you know, after reaching the end of a large document or signal stream, is that the braces don't match up. By contrast, in XML, there is no “universal” close-tag like a close-curly-brace: in the sample above, you have </PHONE>, </STUDENT>, etc. As a result, if there is a tag error, you know exactly what tag is missing (or unmatched), and it is possible to pinpoint the location of the error more precisely (and hence fix the error more easily).

Developer Note: In medical applications, the exchange format that is currently prevalent, HL7 version 2.x (which I will discuss later in a separate chapter) idiosyncratically uses vertical bars as field separators. A new draft version of HL, version 3, replaces this with XML syntax.

- In situations where the textual data to be exchanged is non-hierarchical, such as “flat” files with a non-nested row-and-column (rectangular) structure, a popular format for exchange is the *delimited* format. Here a field separator such as the tab or comma is used, and records are separated by symbols such as a newline. XML is more robust than these in transmission scenarios because if a tab/comma or newline is missing, it is hard to know, without inspection of the content by a knowledgeable human, exactly where the error is. Fully-bracketed syntax is less likely to fail here as well.

This is not to say that other errors in the XML content (such as a wrong digit in the phone number) could not occur, but the extra insurance offered by full bracketing can be significant. (Other techniques, such as the use of algorithms that process the entire content and generate a “summary” called a *message digest*, which is transmitted separately, tell you whether any error has occurred at all. Message digests are used to provide guarantees against electronic message tampering, in addition to the absence of errors.)

Developer Note: The extra verbosity/redundancy of XML compared to other formats is thus a tradeoff for greater assurance of content quality. It also allows greater ease and accuracy of electronic document processing. The fully-bracketed syntax idea was originally introduced in the programming language Ada, sponsored by the US Department of Defense (and is now incorporated into Visual Basic). As opposed to the curly braces of the C family of languages (which includes Java), the end of a block structure is indicated with a specific token (*end if*, *next*, *end function*, etc.). Thus, the very common family of semantic errors due to improper nesting tends to be caught at the exact position of the error.

In situations where XML content must be transmitted, this verbosity does not necessarily translate into increased network load. Knowledge of a specific set of tags for a given document allows compression techniques to be readily employed. Office 2007 files, for example, consist of XML files and associated data (e.g., images) compressed using the widely used ZIP algorithm.

- Consider XML as your first choice if for some reason you find it necessary to devise a means of representing information that must be exchanged between systems. Indeed, XML can be used for other purposes as well (such as specifying configurations) where it is desirable for all the information to be contained in a single file or document. Literally hundreds of XML-based markup languages have been devised for a variety of purposes, such as specifying:
 - Web services (essentially program-code libraries that are executable over the Web),
 - Algebraic equations,
 - Structural chemical formulas, and
 - Representation of graphics based on geometric shapes.

XML is also a natural fit for *configuration files*, which are used to customize a given installation of particular software. Web sites that utilize Microsoft ASP.NET technology, for example, use an XML file called `web.config` that contains information about database connection, default language settings and handling of errors and so on. These settings are customizable to the needs of an individual site.

3.2.1

XML Tool Base

Because of XML's software-friendliness, there are a large number of software tools that facilitate the creation and manipulation of XML. Over time these tools have become more powerful as well as significantly easier to use.

A standard language called XQuery³ lets you access and manipulate XML elements easily. All major database software vendors support XML as a special data type (just like strings and integers) as well as the use of XQuery or something like it – though they provide proprietary extensions to allow operations such as accessing the non-XML columns in the database schema.

3.2.2

XML Data Formats Are More Flexible

Another advantage of XML over textual formats for data interchange is that if, for some reason, the source format changes (with insertion of extra fields), the program that processes a text file also has to be changed, even though the extra fields may not be needed by the destination system. With XML, on the other hand, because each element continues to be sandwiched within its own tags, it is easier for the processing software to continue to pick out only the elements that it needs with much less modification. The program would need to work with the new schema definition to ensure that the document it receives is still valid, but that definition would be provided by the sender. The process of validating according to the definition, however, can be separated from that of pulling out the elements of interest: XQuery can be used for the latter task.

3.2.3

XML Is Embraced and Implemented by Large Organizations

Because of XML's importance, major relational DBMS vendors support XML as a special data type.

An XML column can be as large as a long-text column (typically, 4 GB maximum). However, as opposed to simple long-text, whose contents are essentially almost impossible to validate, a given XML column can be associated with a specific XML schema, so that at the time an XML document is imported into the database, the content is automatically checked for conformance to the schema.

To improve the efficiency of manipulating XML, the DBMSs support *indexing* of the XML content, so that for example, a query aimed at retrieving the content of all <PHONE> tags across all records can execute efficiently. (These indexes are different from those used for structured data, but have the same purpose.) DBMSs also provide facilities to convert relational data to XML format, and XML data corresponding to a "flat-file" structure into tabular format.

Even microcomputer databases support XML to some extent. Microsoft Access, for example, while not supporting the XML data type, can export the contents of a table into XML format, along with an accompanying XML schema specification, as shown in Figs. 3.2–3.4. Figure 3.4 shows part of the specification: you may notice that the syntax of the XML Schema definition language is itself based on XML.

Caveat: As in the case of full-text unstructured data, individual RDBMS vendor capabilities for XML vary widely and are minimally compatible with each other. This is yet another area where the ISO SQL committee was caught flat-footed: the ISO SQL 2003⁴ specification did too little, too late.

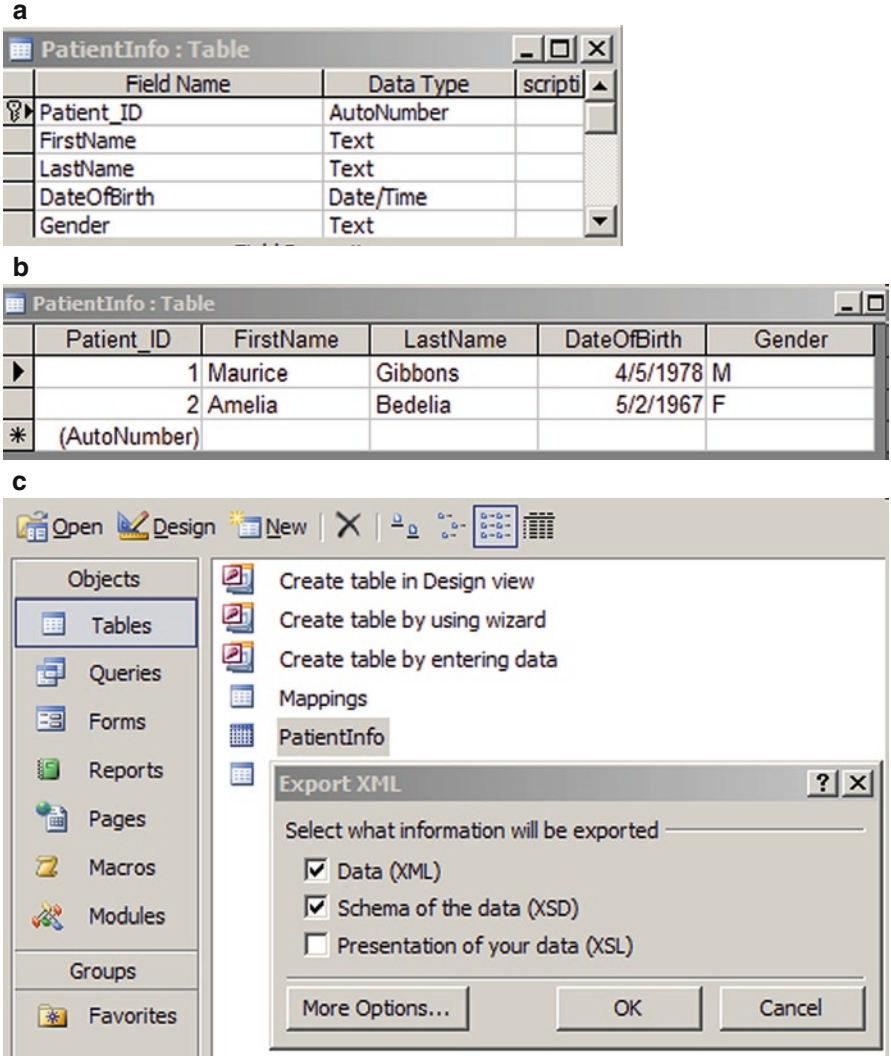


Fig. 3.2 (a) A table definition in Microsoft Access. (b) The same table with two rows of sample data. (c) Exporting the table definition and its contents into XML

3.3 Misconceptions About XML Use

No technology will solve every possible problem for you. There are numerous misconceptions about XML floating around that are not addressed sufficiently in the literature, and so I'll take the opportunity to do so: even the experienced developer may find it useful to skim

```

<?xml version="1.0" encoding="UTF-8" ?>
<= <dataroot
  xmlns:od="urn:schemas-microsoft-com:officedata"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="PatientInfo.xsd"
  generated="2010-04-05T16:45:44">
<!--PatientInfo.xsd is the accompanying schema file generated by MS-Access -->
  <= <PatientInfo>
    <Patient_ID>1</Patient_ID>
    <FirstName>Maurice</FirstName>
    <LastName>Gibbons</LastName>
    <DateOfBirth>1978-04-05T00:00:00</DateOfBirth>
    <Gender>M</Gender>
  </PatientInfo>
  <= <PatientInfo>
    <Patient_ID>2</Patient_ID>
    <FirstName>Amelia</FirstName>
    <LastName>Bedelia</LastName>
    <DateOfBirth>1967-05-02T00:00:00</DateOfBirth>
    <Gender>F</Gender>
  </PatientInfo>
</dataroot>

```

Fig. 3.3 Data from the table in Fig. 3.1 displayed in XML format

```

<xsd:element ref="PatientInfo" minOccurs="0" maxOccurs="unbounded" />
</xsd:sequence>
<xsd:attribute name="generated" type="xsd:dateTime" />
</xsd:complexType>
</xsd:element>
<xsd:element name="PatientInfo">
  <xsd:annotation>
    </xsd:appinfo>
    <od:index index-name="PrimaryKey" index-key="Patient_ID" primary="yes"
      unique="yes" clustered="no" />
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Patient_ID" minOccurs="1" od:jetType="autonumber"
        od:sqlType="int" od:autoUnique="yes" od:nullable="yes"
        type="xsd:int" />
      <xsd:element name="FirstName" minOccurs="0" od:jetType="text"
        od:sqlType="nvarchar">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="50">
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>

```

Fig. 3.4 Part of the accompanying schema file, PatientInfo.xsd. Only information on the table's primary key (Patient_ID) and the first two columns is displayed: the complete file is not shown for reasons of space. Many of the tags are specific to Microsoft Office, but their meaning can be inferred by comparing this with Fig. 3.2a

this section, because a technology as popular as XML is very easy to misapply (and often has been). Later, as we discuss concrete biomedical applications, we'll provide more detailed guidelines about what has worked and what hasn't.

The mere fact that a nascent technology or a markup language uses XML in some form or another does not make it useful for a particular purpose. The set of XML tags that are devised for a particular purpose must adequately circumscribe the problem that is intended to be addressed, and ideally address it elegantly, without more tags than are strictly necessary: few developers would care to learn a specification with hundreds of tags when only a few dozen would solve the problem. Far too many XML-based efforts have been poorly thought out in this respect, and died quick deaths after no one besides their authors bothered to use them (and very often, even the authors abandoned them after publishing a paper or two describing them).

3.3.1

Misconception: XML Is Easy for Humans to Work with and Understand

As stated above, the fact that XML itself has a simple *syntax* does not make a *particular XML specification* either readily comprehensible or easy to use. A case in point: those who don't believe me are welcome to take any Office 2007 document created by Word or Excel, rename the extension to .zip, and, after opening it and inspecting an individual .xml component in this bundle with a text editor, try to make sense of some of the content, which may be nested a dozen or more tag-levels deep. The Office 2007 XML format may be marginally more informative than in earlier versions of Office, which used a binary format and were not intended to be read by human, but it's as understandable as written Swahili or Turkish would be to most native speakers of English: while these languages use the Latin alphabet, it doesn't mean that the words will register in your brain.

Usability and understandability depends on how many tags that specification has, and how they are to be combined together. For our student-information sample document, for example, STUDENT elements are enclosed inside a top-level STUDENT_LIST element, and each of these in turn contain a single PHONELIST element that is permitted to be empty, and so on.

If, however, the tags and their combinations get very numerous, mere mortals can't be expected to remember, let alone learn, all the rules in the specification. Errors will almost always be introduced when trying to create a document based on such a specification by hand, or when trying to make significant manual edits. Microsoft Office's tags across all document types (Word, Excel etc.) number in the thousands.

Further, simply knowing that a tag exists, and that certain tags and attributes are associated with it, tells you nothing about what information the tag is supposed to convey: for that, you need to be intimately familiar with the Office software from both the power-user as well as the software developer perspectives. Even that knowledge may not suffice if the specification is not sufficiently well described in clear prose with lots of examples. At the time of this writing, Office Open XML Software Developer's Kit is actually described most clearly, not in documents from Microsoft's Website, but from the International

Standards Organization site (the Office Open XML specification, to Microsoft's credit, has been made freely available, so that anyone can create conformant documents). Microsoft's documentation for its Office Open XML Software Developer's Kit, by contrast, is completely bereft of either examples or explanation.

In general, a software application that understands the specification (and hence a given document's structure) is required for all but the most trivial specifications. For example, in the case of MS-Office, the application would be one of Word, Excel, PowerPoint, etc.: for an XML specification for geometric shapes one would ideally need to work with a drawing editor. Many vendors have often made the error of releasing an XML specification for a specific purpose *without* providing an accompanying software application to work with a conformant XML document. (Microsoft themselves made this mistake when releasing configuration specifications for their Source Code Control system, Microsoft Team Foundation Server.) This has the result that all but the most hardy or masochistic software developers are likely to simply give up on the specification.

3.3.2

Misconception: XML Allows the Semantics of Document Content to Be Adequately Described

New users of XML enthuse that XML provides *semantic* markup, but XML is only about *syntax*, as pointed out at <http://xml.coverpages.org/xmlAndSemantics.html>. It is true that the simple XML document of Fig. 3.1 is "self-describing", but that is a fortunate coincidence: the tags are few and describe concepts within our sphere of experience. For example, the tags are in English, the language of this book: if the tags used Hungarian or Japanese written in the Roman/Latin alphabet, they would be less understandable, though they would still be valid to an XSD that defined those tags. For that matter, one could use the tag "up" to mean "down", in the fashion of Humpty Dumpty in Lewis Carroll's *Alice in Wonderland* ("a word means what I choose it to mean...").

This is not an idle philosophical digression. The issue of meaning has several practical ramifications.

- In rapidly advancing scientific domains, different research groups may use different words to refer to the same concept, or the same word may be used in subtly different ways that are not obvious to a non-expert. An XSD may define partial semantics (in terms of valid values), but unfortunately the only way to describe meaning is through detailed definitions in prose.
- Certain tags that were devised for a given purpose may subsequently turn out to be inadequate because the meaning of a tag is too broad. One of the standards originally devised to annotate content made available on the Web (but which can also apply to any intellectual creation, as well as material such as museum catalogs) is the Dublin Core,⁵ a set of tags that includes Title, Creator, Identifier, Subject, Source, Publisher, Date and so on. While "Title" is not ambiguous, "Creator" has turned out to be. For vocal music, it does not distinguish between the roles of the lyricist (who wrote the words) versus the person who wrote the music. In a book of photos, the writer of the text may be different from the photographer. "Semantic Web" applications, which are intended to perform

electronic reasoning on Web content that has been annotated with tags such as those of the Dublin Core, would be understandably handicapped if the tags employed could not support representation of necessary nuances in meaning.

- The ubiquity of HTML has imposed a kind of linguistic hegemony on Web content. Because the technology was first developed by an Englishman, Tim Berners-Lee, the tags happen to be intuitive and mnemonic in English (e.g., for Bold, <P> for paragraph markers), and now everyone from China to Brazil has to learn those tags, however unintuitive non-English speakers may find them: fortunately, graphical HTML editors will shield both users and content-developers from the gory details of Web-page composition. The issues with XML, whose syntax was inspired by HTML, are similar: XML is not language-neutral, and therefore it is all the more important to provide detailed prose definitions as to what a particular tag implies.

This situation may change over time: if current trends in technology and world finance continue, those of us who expect to still be in business twenty years from now may find it prudent, like Wernher von Braun in Tom Lehrer's immortal lyrics, to learn Chinese. (Numerous Japanese terms, for example, have already made their way into the areas of manufacturing and management.)

To date, I am not familiar with any efforts to create a meta-specification for XML tags (in individual areas such as graphics or software design) that would be human-language independent. There is no technological reason, however, why this cannot be achieved. All that is needed is a translation table that converts tags from one human language to another.

Developer's Note: By contrast, programming languages are independent of the human-language syntax in which source code may be written – the International Standards Organization defines the keywords for several human-language implementations. Therefore a program developed in Russia for a Windows machine will run on a machine in the US. Note that this portability does not translate to source code: itself as many outsourcing operations have discovered, comments embedded in a program listing are typically mysterious when read outside the programmer's country of origin.

3.3.3

XML is a Misfit for Procedural Programming

One particularly misguided line of XML-related efforts was the attempt to use XML syntax as the basis of *procedural* programming languages, like C or Java. In practice, this implied little more than re-introducing practically all the standard programming-language keywords (e.g., *if*, *while*, *loop*) within angle-bracketed tags, overlaid with a syntax that

might be easy for machines to parse, but is sheer murder for programmers to write non-trivial code with. The result is best described by Dr. Samuel Johnson's analogy to a dog walking on its hind legs: it is not done well, and you are surprised to find it done at all. Mercifully, such efforts are increasingly dying accelerated deaths, and sites describing such approaches are now increasingly hard to find.

XML's use is reasonable, however, when used as the basis for a *declarative* programming language. In a declarative approach, the developer only specifies *what* is desired: a language interpreter figures out *how* to implement the developer's intentions. An example is HTML itself, which specifies how a Web page must appear and function. The interpreter here is a Web browser that takes care of details such as the processing of hyperlinks and the layout of pages whose content automatically rearranges, for example, when the end-user chooses to resize the page or adjust the text size. Similarly, Scalable Vector Graphics (SVG), used to specify drawings based on geometric shapes in Web pages, is very expressive. In both these cases, while it is possible to write HTML and SVG by hand, it is most productive to use special-purpose GUI editors: the only ones who really need to know the intricacies of the syntax of each today are the programmers who have to generate HTML or SVG dynamically from the contents of databases.

Even for declarative programming approaches, however, XML-based approaches are not guaranteed to be superior: machine-friendliness in terms of ease of parsing does not necessarily translate into ease of human use. The most widely used declarative language in the world today is SQL, which resembles English: it may be harder for software to parse, but the subset of SQL that is used 80% of the time has a relatively short learning curve, and most databases will let you compose it graphically by interacting with diagrams.

In both theory and practice, a SQL statement – specifically, the parse tree that is created prior to its execution – is expressible as an XML equivalent. Microsoft Access 2007 in fact converts graphically composed queries into an XML-based internal representation. However, it wisely hides this representation from the end-user.

Developer Note: The inventor of relational databases, E.F. Codd, a mathematician, originally conceived of an algebraic notation to specify database operations, but such notation turned out not to be particularly user-friendly, in addition to being a real challenge to compose on ordinary keyboards. The highly mathematical tone of Codd's original papers in fact contributed to a delay in acceptance of the relational model of data: it took the efforts of Chris Date, Codd's colleague at IBM, to translate Codd's ideas into simple English so that people could appreciate their elegance.

3.3.4

Misconception: XML Technologies Will Eventually Render Relational Databases Obsolete

While most high-end databases support XML as a data type, it must be used judiciously. The excellent text "Microsoft SQL Server 2008 T-SQL programming" provides some

excellent guidelines in the appropriate use of this data type. For example, it is *not a replacement for structured data that is a natural fit for straightforward relational modeling*, even for relationships that are strictly hierarchical. (An example of a strict hierarchy is multiple clinical events for a patient: in the business scenario, an example would be multiple line items for an invoice.) The reasons for this are:

- Data in any complicated domain (including, of course, biomedicine) tends to fall into multiple inter-related categories, and many of the relationships between these categories are *many-to-many*. For example, a scientific experiment may use a variety of reagents, but the same reagent may be used in a variety of experiments. XML is best at modeling relationships that are strictly hierarchical (i.e., *many-to-one*): one must model many-to-many relationships in a roundabout and unintuitive way. An RDBMS is more naturally suited to managing many-to-many data. XML allows you to represent many-to-many data using unique ID elements (analogous to surrogate primary keys) and cross-references to such IDs – so-called IDREFs – but this solution is relatively unwieldy.
- Even in strict hierarchies, some of the columns in the “many” data point elsewhere. For example, line items in an invoice refer to products, about which information is typically stored in a Products table. The relational approach to designing schemas here is standard and readily understandable: you gain nothing with XML.
- Complex validation checks (constraints) are generally far easier to specify on structured data than on XML content – especially in modern SQL implementations that are arbitrarily extensible with procedural code. Further, many tools will generate such checks through simple dialogs presented to power-users, and also execute far more efficiently.
- Designs based on XML are far more difficult to modify than based on structured data, where changes such as addition or removal of columns, column name or data type changes, and addition of constraints are mostly performed through GUIs.
- While XML content can be indexed, the indexes are nowhere as efficient as that for simple structured data such as integers, dates or short strings.
- XQuery, powerful as it is, is somewhat more difficult to use than SQL. This is not necessarily XQuery’s fault: XQuery is intended to solve a different problem, and hierarchical XML structures are more complex than the flat row-and-column structures (tables) that SQL operates on. To use XQuery productively, you must know the exact structure of the XML you are operating on.

To take a concrete example: the `Student_List` XML sample at the beginning of the chapter would be satisfactory for exchanging information between systems, but would make little sense in terms of an actual student database: a table of Students, Phone Numbers, etc. would be more appropriate. Dejan Sarka, author of the XML chapter in the Ben-Gan SQL Server text cited earlier, points out that in cases where simple structured data is to be captured, use of XML for long-term data storage of significant data volumes serves as not much more than rebottled old wine: while some developers may get to show off their knowledge of their latest toy, the customer’s needs are ill-served, and the system eventually becomes difficult to impossible to maintain over time.

Examples of the *legitimate* use of XML columns are:

- You need to store hundreds or thousands of static HTML files (the contents of Web sites) or Microsoft Office documents (Word files, Excel worksheets, Powerpoint presentations or Visio drawings) whose contents often need to be searched electronically (e.g., locate all drawings containing a particular symbol, or all Word documents containing a particular phrase). There is no simple way to represent an HTML or Word document's content (e.g., font and style changes, embedded graphics, etc.) in a relational design, and so XML is an obvious candidate. (For HTML, one assumes that the documents have been created by tools that generate "strict" xHTML output mentioned earlier.)

When stored as XML and indexed, such content can be searched efficiently using XQuery syntax, provided that the developer learns about the Office XML document structure. By contrast, the use of binary formats (e.g., .DOC, .XLS), while not preventing query, is much less efficient, because queries have to be implemented as programming code that must operate on one document at a time.

Numerous other software vendors beside Microsoft make their formerly binary-only files available as XML files. For example, Autodesk, creator of the computer-assisted-design software AutoCAD, supports an XML version of their design files, as do several document-management-system vendors. In the area of chemistry, XML-based formats are now standard for depicting chemical structures.

- Various type of XML documents that serve as templates may also be managed within a database. For example, XML Schema definitions, discussed earlier, are themselves XML documents.
- Developers within an organization may develop custom XML dialects for specific purposes: we discuss an application in biomedicine, natural language processing of medical text, shortly.

The issue of validation is an important one. The tools that are built into DBMSs will check that XML content that is attempted to be imported into a database conforms to the XML Schema associated with the column. However, Costello⁶ states the issue best: XML Schemas, while powerful, are not all-powerful. Biron⁷ reports the experiences of an HL7 working group in developing XML schemas to define and validate HL7 version 3 messages: his summary message is that "HL7's experiences with designing schemas that work across a broad array of tools has been extremely disheartening". Biron's description includes phrases like "nightmare" and "trying to escape from a labyrinth".

Part of the problem may be that the HL7 model itself is not all that simple, as we'll see in a future chapter. However, XML schema technology alone does not suffice to define complex constraints on the data. Means of augmenting validation includes the use of XML-based technologies such as Schematron,⁸ which is open-source and has recently become an ISO standard, and the use of conventional procedural programming languages that are XML-aware, such as the latest versions of the .NET languages.

3.4

Metadata-Related Uses of XML in Biomedicine

While I'm dependent on XML in more ways than I care to enumerate, there are certain uses that I find inappropriate, and I'll mention them at the outset. XML is not a very effective means of storing significant amounts of diverse technical metadata that must be maintained over time by a team of individuals. While it has been used for such purposes – one vendor uses it as the internal format for maintaining a database of “business rules”, which I will discuss later – I would prefer to use an RDBMS for such a purpose. Ease of maintaining the consistency of such content is paramount, and RDBMSs are a mature technology with mechanisms such as referential integrity and transactions: they have been designed to scale well in multi-user scenarios and recover well from hardware errors. The “pure” XML database engines have, as of this writing, a lot of catching up to do in this regard: achieving the equivalent end-result is simply a lot more work.

I now provide a synoptic description of metadata-related uses of XML.

3.4.1

Biomedical Data Modeling

One lesson that has been learned is that having developers work directly with XML to model data and metadata, or to specify software system architectures, is not very productive. One well-known misstep (which was eventually corrected) occurred with the Microarray Gene Expression Data (MGED) Group,⁹ which initially attempted to specify the information that microarray experiments should capture through an XML specification. There are several challenges in applying XML to the data modeling task.

Descriptions of software artifacts, and especially descriptions of data models, tend to be *highly visual*: Peter Chen's devising of the Entity-Relationship (ER) diagram¹⁰ in the 1970s was a design breakthrough in terms of providing a visual representation quickly understandable by non-technical persons: we have used ER diagrams in several places in this book. While software tools will render an XML Schema definition visually, the data-modeling aspects that need to be represented in an ER diagram don't translate well. This is because the display metaphors that are suitable for XML (highly nested structures) are quite different from those applicable to data models.

The Unified Modeling Language (UML),¹¹ originally designed at Rational Software (now part of IBM), is specifically intended to describe various aspects of a system through a variety of diagrams. UML is now something of a *de facto* standard – in part because Rational Software made its definition freely available, turning it over to the Object Management Group. Data modeling is done in UML using Class diagrams, which provide a superset of the functionality of ER diagrams. (The MGED group fortunately corrected itself and eventually employed UML for its specification: the XML specification, which is used to *interchange* data, was in turn *derived* from the UML.)

Developer Note: In graph theory terms, a relational schema is a *general directed graph*, while an XML schema is a *tree*, a specially constrained directed graph, where element has a single parent, so that ultimately everything descends from a single ancestor.

3.4.1.1

Use of XML for Data Modeling

This is not to say that XML technology itself is completely inapplicable to data models: it's just that it should be employed not by humans but by software tools, i.e., under the hood. UML actually has an XML equivalent, called XMI (XML Metadata Interchange).¹² XMI was originally devised so that the UML tools of different vendors could exchange information with each other. XMI content is intended to be generated and interpreted by software tools, but is complex enough that it must operate under the hood: all that the user sees and works with are diagrams. In other words, XMI is a bit like assembly-language; the overwhelming majority of software developers need not care about it.

XMI does, however, provide an immediate benefit. Diagrams developed with UML can be stored in a database in XMI format, so that they can be queried with the database's XML tools. While XMI is not particularly easy to work with, because of its formidable complexity, it is occasionally rewarding, for those brave souls prepared to do this, to be able to query UML diagrams in ways not anticipated by the UML vendor.

Further, software pipelines, often designed by third parties, can also transform XMI into actual implementations such as relational schemas. While such tools could conceivably operate directly on a particular product such as IBM Rational Rose, the tool vendors can be more productive if they work with XMI, because they can more readily port their tools to multiple products: they don't have to worry about dealing with multiple, proprietary binary formats. The individual XMI dialects of different vendors are about 90% similar, while binary formats would be expected to be almost 100% *dissimilar*.

Developer Note: The ability to exchange UML models between software packages is intended to avoid vendor lock-in: *in principle*, a customer of one vendor can move to a rival, superior offering, provided all the information in the old format can be imported into the new system. Unfortunately, *in practice*, individual vendors have been free to beef up their own flavor of UML with proprietary enhancements, and levels of compliance with the UML standard were not specified in detail until recently. Consequently, even for those vendors who use XMI each vendor's flavor of XMI is different from the others. As a result, the promise of UML software-tool freedom of choice is still remote.

3.4.2

Annotation of Reference Content

XML can be used for the storing of *modest* amounts of descriptive metadata, particularly when it is directly associated with the content (typically an individual document) that is being described. Such content serves as annotation. Such uses of XML are mainly in the area of biomedical library science, and have been based on the Dublin Core descriptors, described earlier. I emphasize that the Dublin Core is considered a *minimal* set of descriptors – there are only fifteen tags, which is why the word “core” is used. If its use has been found to be limiting for the area of music, as mentioned before, one could imagine that in the specialized area of biomedicine, it would be even more so. To deal with the limitations, individual groups have created their own tags, but these have not yet attained the level of “standards”.

3.4.3

XML for Data Interchange: Guidelines

3.4.3.1

You Don't Have to Use XML

The fact that XML is a robust data interchange format (because of features like fully-bracketed syntax) does not *mandate* its use for exchange between two systems. People were exchanging data, after all, long before XML was invented. XML's power also comes with some accompanying complexity, and in many situations, its use may be overkill.

If the teams that maintain the respective systems collaborate closely with each other, and the possibility of data loss during exchange is minimal (and data integrity can be verified with message digests) they can use simpler exchange formats such as tab-delimited. XML makes more sense as an interchange standard only when the possibility of such collaboration is remote, and/or the individual systems may not care about all elements in the exchange standard. In such a case, the XML schema definition for that standard will have to specify which elements are mandatory and which are optional.

As I've stated earlier, it's typically a very bad idea to define an exchange format for a brand-new application area straight away starting with XML. The first step is to define, through multiple iterations of a consensus process, what data elements are commonly used by major players in this area, which elements are considered essential and which are optional. The consensus process is facilitated by a graphical data model, using tools such as ER diagrams or UML Class diagrams, which are relatively easy to understand and review. Finally, once everyone has agreed on the names and semantics of the elements in the model (often different players refer to the same functional element using different names), an XML Schema can be derived from the diagrammed data model: most of today's software-design tools will make this step almost automatic.

3.4.3.2

If You Use XML, You Can Do Things the Easy Way or the Hard Way

In any non-trivial data model, several types of data need to be exchanged, and these data are often loosely inter-related. In a business application, for example, information about product suppliers is quite different from information about products themselves.

Relational database schemas (which are closest to the data-model diagrams just described) correspondingly store information in numerous tables. It would be folly to devise a single XML schema that attempts to represent all the data: this would translate into one giant XML document in which all the information would have to be placed before exchange. Some drawbacks to this all-in-one approach are:

- Today's relational DBMSs do a lot in terms of facilitating easy XML-formatted data export, but combining information from a dozen tables into one giant document requires the writing of fairly elaborate programs. One cannot do this with point-and-click or with a three-line query written in SQL. Combining data generates busywork, which is only invited by contract software developers who are being paid by the hour.
- Even if such XML were created, it would be nested numerous levels deep; after being sent to its destination, it will again need to be imported into some kind of relational DBMS. Now the laboriously composed XML will need to be taken apart into the original dozen or so tables with a structure similar to those of the originating system. Again, only contract programmers look forward to such a chore.
- If the data model changes because new elements are added (this happens all the time in biomedicine, because science advances), the interchange model also changes. As a result, the painstakingly composed XML-document generation programs require repeated cycles of major surgery. By contrast, if you were using separate delimited files that corresponded to separate tables, changes to one file would not affect the others.
- It is much better to create *multiple* XML schemas corresponding to individual categories of data. This way, each schema-conformant document has a relatively simple structure that can be created with no programming or with relatively simple SQL scripts, thus reducing the programming as well as maintenance overhead. Each schema doesn't *have* to correspond to a single relational table: for example, one may reasonably combine invoice-header and invoice-line-item information in a single schema and document, even though RDBMSs typically represent this data in at least two tables. Here, the resulting level of nesting is only one level deep – line-items within invoices within a top-level enclosing element like <INVOICE_LIST>. A modest level of nesting is not too hard to follow.

3.4.4

XML for Data Export

XML is a reasonable format to use when a reference database with a very large user-base needs to enable users to save data (e.g., query results) in a format that can subsequently be manipulated programmatically. Examples of such use are seen with most of the databases

maintained by the US National Center for Biotechnology Information (NCBI), which will let you export the results of a search – e.g., of PubMed, Genbank or Online Mendelian Inheritance in Man (OMIM) into an XML format.

The requirements of a data export format are somewhat similar to those of an interchange standard between non-collaborators. That is, the internal data model used by the reference database is not public, because it may change as knowledge advances. Therefore the database’s developers have no choice but to create a single XML export file rather than multiple separate ones, making the task of the individuals who have to extract information from it correspondingly more challenging.

Still, such a format is a Godsend compared to efforts which simply displayed formatted text on the screen. In the past, data-extraction programs had to use a highly fragile technique called “screen/report scraping”, which relied on certain fields being present in certain positions in the screen/output. Unfortunately, these were not guaranteed to stay fixed with newer releases of the database or its front-end software – even a shift of one character to the right or left would cause the extraction program to break.

NCBI implemented structured formats in part due to the needs of numerous groups who were putting their data to uses that NCBI could not have anticipated. At one time, NCBI were strong advocates of ASN.1, and still support it, for the benefit of legacy programs: now their XML format is much more popular with programmers.

3.4.5

Medical Natural-Language Processing

XML columns are often used to store the output of medical natural-language processing. If concepts in a controlled clinical vocabulary – I will discuss this topic in a subsequent chapter – have been identified in medical text such as surgical notes or discharge summaries, then XML-based markup can enclose individual recognized phrases with tags that are labeled with the concept IDs. For example, the original text segment:

...The patient complains of throbbing headache...

may be marked up, after concept recognition using the SNOMED CT vocabulary, thus:

...The patient complains of <Concept Snomed_ID= “162308004”>throbbing headache</Concept>...

Such markup has been used for a variety of purposes. The nature and extent of the markup depends on the NLP approach that is being used. Many NLP applications work as a pipeline: that is, they consist of several specialized software modules that operate sequentially. For example, the first software module may partition a clinical document into sections (e.g., identifying information for patient and physician, chief complaint, past history, etc.). The next module may segment the content into individual sentences, the third may identify noun phrases based on sentence structure, and the fourth may identify individual biomedical concepts embedded in those phrases. Finally, additional modules may identify negation of concepts (e.g., a clinical finding or diagnosis was stated as being absent) or their temporal aspects (e.g., whether they occurred in the past or apply to the present encounter).

Starting initially with the raw text, the textual content is progressively transformed; the output of the previous module becomes the input of the next module in the pipeline. At each stage of transformation, XML markup is added progressively to the output, where the tags or attributes are relevant to a given module's purpose. The markup serves the role of *descriptive metadata*, because it adds value to the original document in the form of embedded annotations.

There are no "standard" XML-based NLP formats as yet, so the output that is created depends on the individual developer (if the NLP module is developed locally) or the software package, if one has been used.

3.5 Conclusions

XML is a conceptually simple yet versatile markup format that has a vast number of applications. Because of its range of applicability, there are numerous pitfalls in its use, and some areas are a better fit than others. Even when it fits the problem, certain approaches are superior and/or simpler.

A side-effect of the ubiquity of XML is that the tools for working with XML are getting progressively more powerful as well as easier to use. The first-generation tools took a lot of effort to accomplish simple tasks. Modern tools – notably Microsoft LINQ (Language Independent Query), which lets you manipulate several data structures (including XML) using a powerful SQL-like syntax, and the XML programming support in the latest version of Microsoft .NET – dramatically improve productivity.

References

1. World Wide Web Consortium. XML schema. 2001 [cited Feb 20, 2002]; Available from: <http://www.w3.org/XML/Schema>.
2. Larmouth J. The use of ASN.1 in the specification of e-business standards - (Oct 2003). 2003 [cited 10/2/10]; Available from: <http://www.asn1.org/paper/index.htm>.
3. World Wide Web Consortium. XQuery. 2010 [cited Feb 20, 2010]; Available from: <http://www.w3.org/XML/Query>.
4. Whitmarsh Corporation. SQL 2003: draft standard. 2002 [cited 10/1/10]; Available from: http://www.wisecorp.com/sql_2003_standard.zip.
5. Dublin Core Metadata Initiative. Dublin Core Metadata element set, version 1.1: reference description. 2002 [cited 2002 Feb 23, 2002]; Available from: <http://dublincore.org/documents/dces/>.
6. Costello R. Extending XML schemas: a collectively developed set of schema design guidelines. 2010 [cited 10/2/2010]; Available from: <http://www.xfront.com/ExtendingSchemas.html>.
7. Biron P. Health Level Seven: XML schema user experience report. 2005 [cited 9/10/10]; Available from: <http://www.w3.org/2005/05/25-schema/HL7.html>.

8. Schematron.com. Schematron: a language for making assertions about patterns found in XML documents. 2010.
9. Microarray Gene Expression Data (MGED) Group. MGED Home Page. 2002 [cited 7/8/02]; Available from: <http://www.mged.org/index.html>.
10. Chen PP. The entity-relationship model: toward a unified view of data. *Assoc Comput Mach Trans Database Syst*. 1976;1:9-36.
11. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*. Reading: Addison-Wesley; 1998.
12. Object Management Group. XML Metadata Interchange. 2002 [cited Feb 20, 2002 v. 1.02]; Available from: <http://www.omg.org/technology/documents/formal/xmi.htm>.

4.1 Introduction

Electronic medical record (EMR) systems illustrate all of the challenges of designing and maintaining large systems. In complexity, their database designs rival those of the Enterprise Resource Planning (ERP) applications. For the large institutional EMRs, the similarity extends to price as well as to the level of institutional commitment required to change your existing processes. When you invest in such a system, you are betting the bank: it's extremely expensive to change your mind, and deployment failures from inadequate customer preparedness can be spectacular. Consider the following issues:

- Like ERP applications, EMR systems need to interface with a wide variety of third-party software, and they may even interface with ERP applications for areas such as accounting and inventory. However, in terms of database design, their complexity is of a different type. Specifically, an EMR's customers will demand the need to capture new kinds of data that may not have existed at the time the EMR was purchased. We consider a couple of these situations.
- As the science of medicine advances, new clinical approaches may be devised to treat common conditions, that involve specific diagnostic workups, and specific treatment options may be employed depending on the results of these workups. A particular diagnostic workup involves recording of positive findings in the patient's history, clinical examination or investigations as well as "significant negative" findings. (Significant negatives are findings that are looked for because they are important to a particular condition. Even when found to be absent, their absence is still documented because of an impact on decision-making. For example, in a workup of suspected Alzheimer's disease, it is important to document that a particular patient does *not* have psychological changes such as depression or delusions.) In addition, specific parameters need to be monitored to determine clinical progress after therapeutic intervention.
- A new infectious agent (e.g., H1N1 strain of influenza) emerges and new lab tests may be developed to detect it.
- New therapies (including new medications) may be available to treat existing (or new) conditions. The therapies may involve, not just giving individual medications in isolation, but using groups of medications in a particular dosage and chronological schedule: the latter approach is commonly used for treatment of cancers.

To handle these situations, one must be able to define new disease conditions, clinical findings, lab tests, and medications as well as new groupings of existing/new findings and therapeutic measures. Finally, one must be able to associate all of these with the individual patients to whom they have been applied. All of these changes need to be made to the system without having to involve the vendor, and sometimes the changes need to be made within hours or days. Especially rapid response is required for public-health emergencies such as those that accompany natural or man-made disasters.

The requisite flexibility is achieved through the use of two design approaches:

1. A means of representing highly variable clinical data that faithfully reflects its *sparseness*: that is, while hundreds of thousands of past history items, clinical findings, diagnoses, and investigations (including laboratory tests) exist across all of medicine, only a very modest subset of these will apply to a given patient. One technique here is to use extensible, multi-dimensional hash tables that can be stored on disk; another technique employs the Entity-Attribute-Value (EAV) data model. I will discuss both of these shortly, describing their respective strengths and limitations.
2. The use of a large sub-schema with numerous tables containing metadata. The metadata is used to support the presentation, capture, interpretation and export of the clinical data: without the metadata, the clinical data would essentially be unusable.

4.2

The Protocol

The management of a patient with a single known, reasonably common, disease condition is typically based on a *protocol*, which refers to a standardized means of information gathering and decision-making based on the information gathered. The parameters that need to be recorded in this condition have been worked out by experience over years or decades. For common conditions, protocols have been in use for more than a hundred and fifty years.

Even when existing only as paper forms rather than electronically, the use of protocols not only minimizes the chance of accidentally omitting to gather essential data, but can serve as a means of training care providers: with repeated exposure to the printed protocol, the provider internalizes/memorizes it and is able to function effectively even when the protocol is unavailable (e.g., in emergencies in the field). An electronic protocol, of course, allows data to be captured in a structured form, with a large number of discrete fields: such structure facilitates future data analysis.

As knowledge advances, the number of protocol expands. Less than a decade ago, protocol for AIDS did not exist, but now AIDS is complex enough to have a whole bunch of sub-protocols hanging off it. Further, a given protocol may change: new parameters may be added as knowledge of the condition advances, or certain parameters that were thought to be important may be eliminated as they are later discovered to be non-relevant, or in the case of investigations, not useful.

Protocols are not a solution to all problems. Their use breaks down in the following situations:

- To use the protocol-based approach effectively, you must know what condition the patient has, and ensure that there is a protocol available for that condition. In emergency medicine, the nature of the patient's condition is often unknown, and the variety of conditions that the patient can present with is vast. Consequently, most of the data on the patient must be gathered in the only format that is flexible enough to capture any information that the patient volunteers, or is discovered during the examination – that is, (unstructured) narrative text.
- They cannot be applied for a completely new disease condition. For example, in the seventies, the illness described as “Toxic Shock Syndrome” was discovered predominantly in young white women. After several false leads, it was found that all these women happened to be menstruating while they came down with the illness, and they were eventually found to be using a particular brand of hyper-absorbent tampon whose prolonged use was found to cause increased vaginal growth of a particular bacterium: this bacterium, a strain of the very common *Staphylococcus aureus*, manufactured a toxin that was responsible for the symptoms of the illness.
- The presence of one disease does not preclude the presence of another. With multiple, simultaneous illnesses in the same patient, multiple protocols may need to be employed on the same patient simultaneously.
- In situations where sophisticated electronic support may not be available, use of electronic versions of protocols is rarely feasible.

In primary care or emergency medicine, even if electronic protocols were available, the conditions that a given patient can present with are far too diverse to allow convenient selection of a specific protocol, especially when the clinician is still in the process of discovering what exactly (or approximately) is wrong with the patient. Any protocols that may be employed are based strictly on human (care provider) memory.

For all of the above reasons, narrative text – of the long-text data type – is an important part of the medical record. In addition, it is the only way to record nuances of information (e.g., patient-volunteered information) that the structured fields in the electronic protocol (which belong to the numeric, enumerated, date/time or short text data types) do not allow. Therefore, most structured-data-capture user interfaces in EMR software have “comment” fields where narrative text can be recorded.

4.3

Representing Structured Data: The Challenge

Representing the data storage for a single clinical protocol would seem to be intuitively obvious: we would create a table with one field or each parameter of interest, with one row for each patient where it applied. Strictly speaking, rather than one row per patient,

we would have one row per *clinical event*, where an event is a combination of a patient ID and the date/time of the clinical encounter: this is because a patient may present with the condition (especially a chronic one) more than once.

EMRs typically use UIDs to denote clinical events, which are recorded in specially designated tables, though the clinical event can mean different things in different circumstances. In the ambulatory care setting, for example, clinical events would correspond to patient visits. However, in the laboratory test setting, there may be multiple time-stamps associated with an individual test result: the date/time when the patient sample was collected, when the test was performed, and when the provider who requested the test was notified.

However, a design that is convenient for a single disease is not workable for a design that must serve an entire hospital with multiple specialties. Such attempts have been made, and have failed spectacularly. The following reasons illustrate the design problem's difficulty.

- Across all branches of medicine, the protocols potentially range in the thousands, requiring that many tables be created. This makes it challenging to answer the most common query “tell me what’s wrong with a particular patient”: a vast number of tables would have to be searched sequentially to determine if that patient’s data is present. (Remember that the vast majority of patients have only one or two things wrong with them: this is what we mean by sparseness.) New protocols require creating new tables, as well as building user interfaces to those tables.
- As previously stated, the parameters that are recorded across all protocols range in the hundreds of thousands. Further, certain common clinical parameters are important for multiple clinical conditions (and hence multiple protocols). Measurement of blood hemoglobin, for example, is important in several conditions: anemia due to iron deficiency or worm infestation, blood loss following trauma, as well as monitoring of cancer chemotherapy with drugs that can adversely affect the functioning of the bone marrow (where the red blood cells that contain hemoglobin are manufactured). Such parameters would have to be replicated in multiple places: the common task of simply locating the most recent value of that parameter would be a challenge to carry out. Defining a parameter takes significant effort. Apart from its data type, one must specify a host of validation tests for it, such as maximum and minimum permissible values, as well as validation across a set of parameters: for example, the sum of the components of the differential white blood cell count must sum to exactly 100, since the individual components represent percentages. Once a parameter is defined in detail, we would like to reuse its definition across all the protocols where it is relevant.
- From the above account, it may seem that one could compose a protocol simply by putting a set of parameters of interest together, and presenting them to the user in a logical order that is determined by the protocol (e.g., history-related parameters precede clinical-exam-related ones). However, there are also *intermediate groupings* of parameters that must be modeled. Many lab tests, for example, are performed as part of a standard panel – e.g., hematology, serum electrolytes, and renal function tests.
- Finally, protocols themselves are not applied in all aspects of medical data capture, as stated earlier: the more uncertain the status of the patient and the more variable the information that needs recording, the more often one must resort to narrative text.

There are two steps required to address the above problems.

1. Devise a means of recording sparse patient data that does not require creation of new tables as new parameters or new protocols are defined, which still stores concisely the positive or significant negative parameters that have been recorded for a given patient. A value of a given parameter is stored just once, irrespective of how many protocols it applies to.
2. Separate the definitions of parameters and the protocols where they are used as building blocks from the patient data. The definitions can then serve as templates for presentation, entry and interactive validation of the data.

This chapter will deal primarily with how structured data is captured in the EMR. Subsequent chapters will address how the protocol itself is represented, as well as introduce methods that attempt to make the unstructured narrative-text data more useful.

4.4
Storing Sparse Data

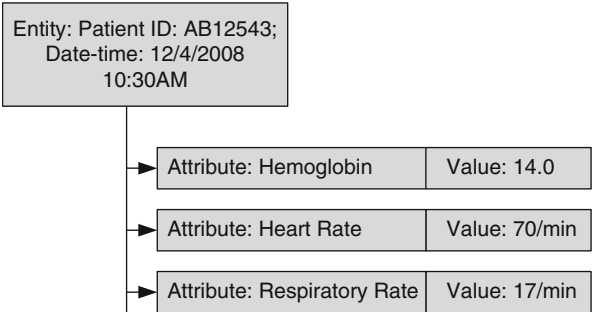
Approaches currently used widely for storing sparse data in EMRs utilize *key-value pairs*, which I now define with a simple EMR-related example. To record the value of a clinical parameter for a given patient at a point in time, the key is a combination of the clinical event and the parameter of interest: the value is the value of the parameter. For example:

Key: <Patient ID: AB12543; Date-time: 12/4/2008 10:30 AM; Parameter: Blood Hemoglobin>

Value: <12 gm/dl>

Key-value pairs are a general means of representing almost any information, and their use, which is hardly limited to biomedicine, goes back to the world of artificial intelligence research in the 1950s. It can be seen that the key is a combination of two logical components: the thing being described (the *object* or *entity*: here, the clinical event) and a particular aspect of the thing being described (the parameter or *attribute*). There are several alternative ways to physically implement this conceptual design: for example, one can have a set of attribute-value pairs that are linked to the entity that they refer to, as illustrated in Fig. 4.1.

Fig. 4.1 Attribute-Value pairs associated with an “Event”, the Clinical Event (a combination of Patient ID and a time-stamp representing when the Attributes were measured in the patient). One can record as many pairs as there are findings (positive or negative) of interest for that event



With the key-value-pair approach, we record only what we know about a given patient at a point in time: that is, the positive and significant-negative findings. This approach is radically different from maintaining thousands of columns for what a patient might have, but whose values are either non-relevant or unknown.

4.4.1

Speeding up Search of Keys: Indexes

In production databases, where the number of key-value pairs runs into the millions, it is important to devise ways to store such data on disk in a way that facilitates rapid search for individual keys (or a set of keys). In the case of the EMR, the commonest type of query is “Show me all about the patient XYZ, with the findings ordered with most recent findings first; optionally, limit the display of findings to those within a specific range of dates, such as the last two visits, and specialty.” The standard technology used to facilitate fast key search is based on ancillary disk-based structures called *indexes*. The use of indexes is ubiquitous in database technology and not limited to key-value pairs. I will provide references to the very well-written Wikipedia entries on related topics for those who would to know more. I will now summarize index implementations.

- The most common implementation of indexes, which uses a standard software technology called a B+–tree,¹ allows the following operations to occur with reasonable efficiency:
 - Locating a single key, after which one can fetch the value, alter the value, or delete the key. The time required to locate a key is proportional to $\log_2$ (number of keys). This means that locating a given key from among a million keys takes only 5 times as long as locating a key from a collection of 16 keys:
 $\log_2(1,000,000) \approx 20$, $\log_2(16) = 4$.
 - Sorting the keys (e.g., ordering events most-recent first), and therefore retrieving a range of keys (e.g., events within a particular date range).
 - Inserting new key-value pairs.
- There are alternative indexes which offer different benefits compared to B+ trees.
 - For example, *hash indexes*² allow very fast location of a single key (in constant time, irrespective of collection size, by transforming the key into a number that corresponds to the physical position of the key-value pair in storage), using a fast mathematical operation called a hashing function (see the Wikipedia entry for more details). However, they support key-range searches relatively poorly.
 - *Bitmap indexes*³ are useful when the number of unique keys to be searched is relatively few in comparison to the collection size (e.g., in an automobile database, while there are millions of cars, there are much fewer makes/models of car, years of manufacture, color and state of registration). They take up much less storage space than B+ trees and allow rapid identification of a set of records based on combinations of categories (e.g., find all 2005 maroon Ford Explorers with Arizona license plates).

To illustrate how a bitmap index works, consider a national motor vehicle license database (the license table would be expected to contain around 250 million records). Consider a field such as State of registration, which can take about 60 unique values in the USA (if we include non-state Federal territories as well). We create one array of bits (1s or 0s) for each unique value. Each array has one bit for each physical record: the position in the array corresponds to the record number. A bit value of 1 means that the value is present for that record: 0 means it is absent. Thus, if the Arizona bitmap had a 1 in position 14, it means that the motor vehicle for the 14th physical record is registered in Arizona. For a given array, the majority of bits would be zero, and remember that 8 bits fit into one byte: consequently, bitmaps can be stored on disk in a highly compressed form.

If we had similar bitmaps for vehicle manufacturer, to find Volkswagens registered in Arizona, we would simply overlap the Arizona bitmap with the Volkswagen bitmap, and find all records where both corresponding bits were 1. This operation, called bit-ANDing, is performed very rapidly by computer hardware. (See Fig. 4.2.) Some compression techniques allow bit operations to work directly on compressed indexes without needing to uncompress them first.



Fig. 4.2 Bitmap Indexes. The *top two arrays (bitmaps)* record information in a vehicle database for the State of Registration (*Arizona*) and the Manufacturer (*Volkswagen*) respectively. There are as many cells as there are records: a 1 indicates that the value is present, 0 means it is absent. In the first bitmap, the cars in records 4,8 and 11 are registered in Arizona. The third bitmap is computed on the fly if we want to find out all Volkswagens registered in Arizona. The two bitmaps are merged with an AND operation – a given cell takes the value 1 only if both corresponding cells have the value 1: otherwise, it is zero. Here, records 4 and 9 match both characteristics of interest

4.4.2
Storing Key-Value Pair Data

There are several production approaches for key-value pair storage, of which two are widely employed.

4.4.2.1
Hierarchical Key-Value Pairs

The first approach, used by software that has roots in the M (formerly MUMPS) programming language, uses a hierarchical structure, where a given value in a key-value pair can itself be a collection of key-value pairs, with this structure being nested arbitrarily to any level of complexity. (In M, this structure is called a “global”: this term simply refers to a variable of global scope whose value is stored on disk, and therefore persists between program runs. The M runtime environment uses clever caching algorithms to load some of the global data into RAM for faster access.) The way the data is organized is as follows:

- For EMR data, we have a collection of patients. Linked to a given patient is a set of clinical events for that patient (i.e., a sequence of date-times). Linked to each event is a collection of parameter-value pairs.
- At each level, the keys in a collection are organized for fast access using an index.

Figure 4.3 illustrates a hierarchical structure. This design is extremely fast for locating very specific data on a single patient: the software simply follows the links, using indexes to locate particular keys or key-ranges. This operation turns out to be important in systems that support point-of-care operations, at both the ambulatory and in-patient level. (In database parlance, hierarchical key-value pairs are very efficient for transaction-oriented EMR applications.)

However, this hierarchical structure is less well suited to answer questions such as “Show me all patients who have a particular clinical finding, or a particular range of values for a clinical finding – such as patients with a blood hemoglobin value less than 7 g/dl”, because

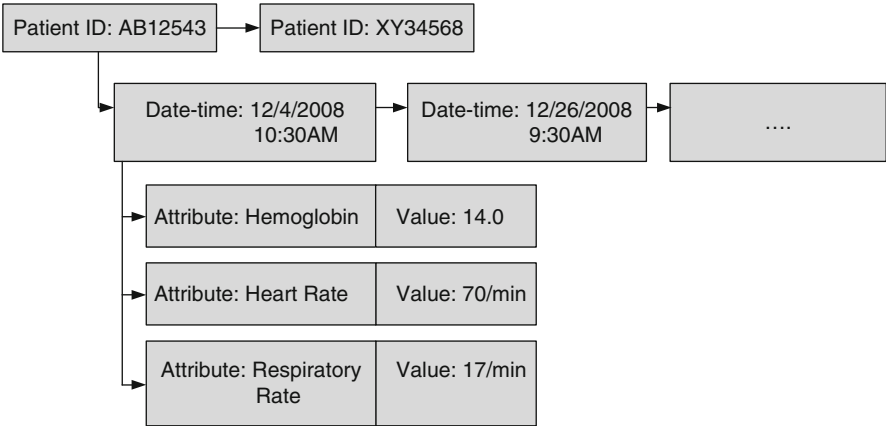


Fig. 4.3 Hierarchical Key-Value pair structures. At the *top level*, we have a list of patients. For a given patient, we have a list of clinical events. For a given event, we have a list of Parameter-Value pairs. Notice that in this structure, the only way to get to a particular Parameter-Value pair is via the corresponding event, and therefore from the corresponding patient. Because of this navigational design, one cannot rapidly determine, for example, all Hemoglobin values less than 10 mg/dl, and then identify the patients and clinical events that these values refer to

there are no indexes to speed up access to individual attributes (or values of attributes). In other words, the attribute information is sub-ordinate to the event information, which is in turn sub-ordinate to the patient information.

4.4.3

Entity-Attribute-Value Tables

The second approach uses a relational database (RDB) engine. This approach, the *Entity-Attribute-Value* (EAV) data model,⁴ consists of tables with at least three columns: the entity, the attribute and the value. RDB designs store data with minimal redundancy: therefore, for the Entity column, the UID of the clinical event is taken from a clinical events table that contains patient UID and date-time stamps. For the Attribute column, one uses the UID of a clinical parameter, which is taken from a Parameters table (a metadata table) that contains detailed information on each parameter: name, description, data type, information for presentation and validation, and normal-range information for those parameters that happen to be laboratory tests. (In keeping with RDB design principles again, all this information cannot always be stored in a single table, and so multiple tables are typically used: normal ranges, for example, depend on multiple factors, such as gender, age, as well as physiological states such as pregnancy.)

RDBs typically require a column to have a specific data type – e.g., integer, decimal number, text, date. While the entity and the attribute are long integer unique identifiers (UIDs), this becomes an issue for the Value column. Four approaches have been used here:

1. Every value is stored as text, but the Parameters table records what the true data type of a given attribute is. Data that is intrinsically a number or a date is converted on-the-fly. In the previous chapter, I have vigorously deprecated this approach, and provided reasons for the same.
2. The table contains multiple value columns, one for each data type. Only one of them has data: the other columns are empty. An additional column (the discriminator) records which of the value columns actually holds the data. While this approach is not as bad as the previous one, I don't recommend it: it makes bulk queries of data somewhat unwieldy because you are constantly consulting the discriminator to determine how to handle a given row. Also, the discriminator is less useful than one might think: most of the time, you will need to consult the Parameters table anyway in order to determine how to present a given parameter. (For example, if the variable is of the Enumerated data type, as discussed in the previous chapter, you will present it as a pull-down list that shows the descriptions, rather than presenting the internal numeric code.) Further, if you want to index the values, you will have to create one index for each value column, and since in a given row all value columns except one are null, the majority of entries in each index will be for null values.
3. Microsoft SQL Server (but not other database engines) supports a data type called the *Variant*, which is a kind of general-purpose data type: a variant can hold almost any data other than an image, and the data type will change based on what it assigned to it. While this seems highly flexible, this flexibility comes at the cost of computational

efficiency: adding two integer-variants, for example, involves converting each variant into an integer, and converting the result back into a variant.

Variants are one of those features that are design traps: the database designer is essentially allowed to postpone serious thought about design decisions, and I can't think of a single circumstance that justifies their use. The use of Variants – the only data type in the VBScript programming language – leads to strange runtime errors if the Variant's value happens to be of a data type other than what was intended.

Aside: VBScript is the only version of Basic that Microsoft gives away freely – VBScript interpreters are built into Internet Explorer as well as Windows itself, and you can implement your own version without paying Microsoft a dime in royalties, as long as you acknowledge their trademark. This is why it has been adopted by several vendors – e.g., Cache, discussed subsequently in this chapter. However, for many developers, VBScript is worth exactly what you pay for it. It may be easy to learn, but as a serious language, it has proved so unpopular with programmers – it has been called the castrated, lobotomized version of the full Visual Basic – that Microsoft now de-emphasizes it; there have been no enhancements to the language after 2000.

4. The approach that I recommend is the use of *separate EAV tables for each data type*. The Parameters table referred to earlier contains information on the data type of a given parameter, so that you know which EAV table contains a given parameter's data. You can have EAV tables for integers, decimal numbers, short text, long text, dates, strings and binary data such as images. The advantage of this approach is that one can use indexes on the value, so that searches for values in a particular range can run rapidly. (Note that in approach 1, an index is essentially useless, for reasons stated in the previous chapter.)

Note that when you use data-type-specific EAV tables, *you must designate each table's value column with the largest size allowed for that data type in your DBMS*. Thus, for integers, you must use long integers; for variable-length strings, the maximum length that can be indexed (typically, 255 characters), and for decimal numbers, the maximum width and precision allowed. (You could, if you wished to complicate matters, designate separate tables for bytes versus short integers versus long integers – these representations using 8, 16 and 32 bits respectively – but on today's storage devices, the miniscule amounts of disk space saved are not worth the extra housekeeping effort: remember that each row still has to accommodate entity and attribute columns, so your savings are not as dramatic as you may hope.) If you actually wish to restrict the values that are entered – e.g., integers between 1 and 10, or strings less than 20 characters, such constraints must be specified in metadata: this is the focus of the next chapter.

The conceptual design of an entity-attribute-value sub-schema is shown in Fig. 4.4.

As you will see in later chapters, this design has been greatly simplified for purposes of illustration – for example, it shows only a single “type-less” EAV table. For now, I won't jump the gun and overwhelm you with details.

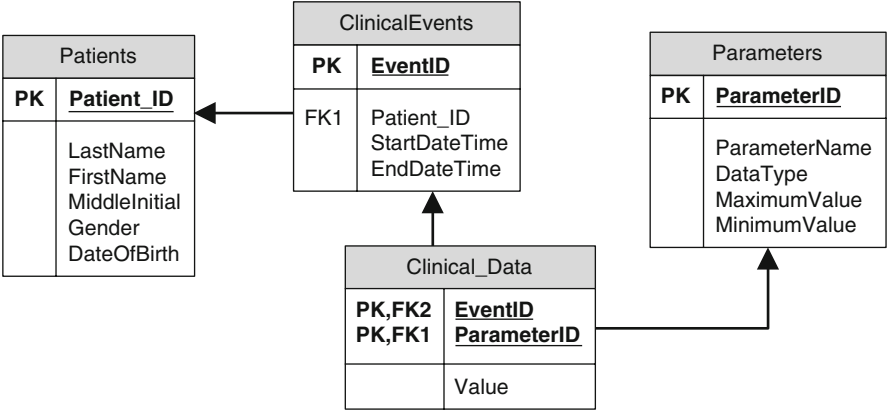


Fig. 4.4 The Entity-Attribute-Value design in a relational database. “PK”=Primary Key (the field/s used to identify a given record in a table uniquely). FK=Foreign key – A field whose value is borrowed from a primary-key value of the table that the arrow points to.

The Patients table contains demographics information on all patients: the ClinicalEvents has an artificial unique-identifier primary key, the Event ID, which corresponds to a given Patient-Timestamp combination. Some events, which represent time intervals (such as an intravenous infusion), have two time-stamps, representing the time the event started and ended. Most events, however, represent single points in time (such as a blood collection or a clinical examination, where start and end-points are close enough not to matter.) For such events, only the Start-timestamp is used; the end time-stamp is left empty.

The Parameters table records detailed information on each clinical parameter of interest: as for the Patients table, only a few of the fields have been shown.

The Clinical_Data table is the Entity-Attribute-Value table: Entity=EventID, Attribute=Parameter ID. In practice, because the Value column must have a fixed data type, there are multiple EAV tables, one for each data type (e.g., integers, decimal numbers, enumerated, text, etc.)

Aside: MUMPS deals with the data type issue by being a type-less language: everything is stored as a string, but based on usage, a string can be treated as a number. This can lead to the same kinds of errors that are encountered with variants. Further, compared to modern programming languages, MUMPS has very limited support for dates.

4.4.4
Using EAV Modeling Judiciously: Caveats

Key-value pairs and the EAV data model, as I’ve stated earlier, are a very general means of representing almost any kind of data. That does not mean that they are *appropriate* for all kinds of data. Note that a single key-value pair, or a single row in an EAV table, contains a single fact about a given entity. To store twenty facts about an entity, you need twenty rows.

If, on the other hand, you have several thousand entities that are *homogeneous*, i.e., described by the same, limited set of attributes, then it is simplest to create one *column* for each attribute, so that a single *row* of a table contains all the information that you would want to record on that entity. This approach to modeling data (*column modeling*) is the conventional as well as intuitive approach to modeling homogeneous data: column modeling is what is typically taught in basic Database courses.

By contrast, EAV represents a form of *row modeling*, because individual facts are modeled in rows. The line items in a supermarket receipt are an example of row modeling: each line contains information only on an item that you have purchased: the receipt does not contain data on all the thousands of items in the store that you could have purchased but did not.

I will now describe aspects to consider when implementing EAV designs.

4.4.4.1

Complex Boolean Query on EAV Data Is Not Particularly Efficient

Relational databases query data easily on combination of criteria involving multiple attributes when all the attributes refer to columns of a single table (e.g., in a table of persons, finding all females above 45 or males above 60 currently living in New Haven, CT). If the same information was stored as key-value pairs, one would have to perform the same query using set intersection to implement the Boolean AND operation and set union to implement the OR operation: both of these are more hardware intensive than if we were operating on columns of a single table.

In the single-table scenario, the DBMS optimizer would first identify a subset of data using columns that happened to be indexed: the resulting subset would then be scanned in a single pass to identify rows matching the remaining criteria. To achieve the equivalent with EAV data would involve the following steps:

1. Find all rows where attribute="gender" and value="male". (i.e., find all males).
2. Find all females.
3. Find persons above 45.
4. Find persons above 60.
5. Find persons living in Connecticut.
6. Find all persons living in New Haven.
7. Combine sets 1 and 2 using set intersection.
8. Combine sets 3 and 4 using set intersection.
9. Combine sets 7 and 8 using set union (for the OR condition).
10. Combine sets 5 and 6 with intersection.
11. Combine sets 9 and 10 with intersection.

One could combine certain operations into a single step (e.g., 10 and 11 could be merged), and one could take advantage of the fact that set 6 will always be a subset of set 5 (so that step 5 could be eliminated entirely). But in any case, the process will be more involved. Query generators that understand EAV data can automate generation of the SQL that creates all of the steps, but the resulting SQL is much more complex and consequently takes

much longer (though not prohibitively so) than the straightforward SQL that operated on a single column-modeled table.

In summary, you don't use EAV unnecessarily: *you must use it only for data (such as clinical parameters) where attributes are sparse.*

4.4.4.2

Implementing a NOT Operation Can Be Expensive with Row-Modeled Data

A standard method of recording diagnoses in electronic medical records is to record *diagnosis codes* against a patient. Such a table has three columns: Patient ID, Diagnosis code, Date of diagnosis. A patient with three diseases would have three rows of data. Since there are several thousand known diagnoses, the *absence* of a given diagnosis is not recorded explicitly: it must be *computed*. Suppose we wanted to identify patients *without* hypertension. We must first get the complete set of patients, then find those with hypertension, and finally subtract the latter from the former using *set difference*.

The number of rows returned is unpredictable: all we can say is that in the worst case—if the diagnosis is very rare—almost all patients in the database may be returned. In fact, it is doubtful that simple NOT operations would even be permitted *by themselves*: such queries would indicate an extremely naive user.

However, NOT operations combined with an AND (e.g., persons living in New Haven *and not* having hypertension) could be more selective: one could first identify the former, and using a sub-query, *eliminate* those with hypertension. We label this selective set difference the AND_NOT operation. As we will see in a later chapter on query of EAV data, rearrangement of an expression (NOT x and y) into the form (x AND_NOT y) is a useful query optimization strategy.

4.4.4.3

Row-Modeled Data Is Not Easily Analyzed in Its Raw Form

One problem with data represented in row-modeled format is that they are not very suitable for analysis: most statistical and graphical tools expect data in column-modeled format. Converting row-modeled data to column-modeled data is a straightforward operation (if compute-intensive for large data volumes), and is called *pivoting*: users of Microsoft Excel are familiar with this concept, but it is illustrated in Fig. 4.5 below, which illustrates the end-result for a single patient. The less common, inverse, operation is called un-pivoting or de-pivoting.

4.4.4.4

The Infrastructure for EAV Systems Is Not for the Faint-Hearted

More important, the use of the EAV data model *mandates a very robust metadata infrastructure* to your software system. If your data is column-modeled, data-type checks prevent entering of, for example, invalid dates. Range and other types of validation checks

Patient ID	Time Stamp	Attribute	Value
XYZ	1/2/08	Sodium	130
XYZ	1/2/08	Potassium	5.2
XYZ	1/2/08	Calcium	9.2
XYZ	1/2/08	Hemoglobin	12.6
....			

Patient ID	Time Stamp	Sodium	Potassium	Calcium	Hemoglobin
XYZ	1/2/08	130	5.2	9.2	12.6
....					

Fig. 4.5 Pivoting of EAV data. The *top table* shows a snapshot of patient data in EAV structure (This is a view that combines the Form_Data, Group_Data and Parameter_Data tables of Fig. 4.4.) In the bottom table, the attributes – Sodium, Potassium, etc., – have been converted into individual columns. The individual data values are joined on a single patient/time-stamp pair

on columns or sets of columns are also readily defined, often using very straightforward SQL. However, it is much trickier to prevent entry of erroneous data if an EAV approach is being used, because the database layer of your software cannot perform any checks. Preventing such errors requires a layer of software that constantly consults the metadata (such as the Parameters table) to check every operation that the system performs as a consequence of the user’s actions.

Having to consult metadata constantly seems more inefficient than it really is. The meta-data is much less voluminous than the data, and on modern CPUs with abundant RAM, is readily cached in memory. (Certain DBMSs, notably Oracle, allow you to specify, at table creation time, the extent to which the table’s contents and indexes should be cached in RAM: other engines such as MS SQL Server let you achieve this performance tuning in a more roundabout way.)

Further, most of the operations that are error-prone are interactive user actions on data, and on modern CPUs, the time waiting for the user is an eternity compared to the time required to perform database lookups on cached metadata. The challenge is in re-inventing the RDB wheel – writing program code to perform all the operations that the database engine performs with no (or minimal) intervention on the developer’s part.

Finally, it is important to note that using the EAV data model in a large scale system calls for a level of software-engineering discipline that is much higher than if you were to use a column-modeled approach. The metadata, which I will describe in detail in the next chapters, not only serve the purpose of documentation (of individual parameters, for example), but are essential to be able to interpret the data. For example, an attribute UID is meaningless to the user, but the parameter’s description is meaningful. For values of the enumerated data type, the numeric codes are meaningless, but the textual descriptions associated with each code convey meaning. It is important to ensure that the metadata and data are completely synchronized with each other: changes to the metadata must be reflected instantly in the data.

4.4.4.5

Pure EAV Designs Are Unrealistic: Use Only as Directed

I have stated that EAV designs involve re-inventing the wheel with respect to maintaining data integrity. In the case of the highly heterogeneous and variable data that one encounters in the context of the clinical data in an EMR, such wheel reinvention is unavoidable. However, you do not want to go through this effort needlessly. It is important to note that even in EMR systems that use the EAV data model, the EAV approach is mostly employed *only* for the clinical data. If you were to inspect the definitions of tables in such a system, you would find the overwhelming majority of them using the conventional, column-modeled approach. In addition to basic Patient Demographics, which has been illustrated earlier, Visit information, and many tables in the Billing sub-schema, almost all of the Metadata tables are conventional. This is because the attributes (columns) in these tables are *homogeneous* – there is a fixed, predictable number of them for a given table – and they are *dense* – that is, they are mostly filled in with data rather than empty. In other words, in the real world, so-called “EAV” schemas are actually “mixed”: the database designers use both conventional and EAV modeling as appropriate for different categories of data.

Further, even if some data is naturally row-modeled – such as purchase-order line items – that *does* not make it a candidate for EAV modeling. EAV modeling is a *generalization* of row-modeling that is useful for highly *variable* attributes. On the other hand, the attributes that constitute the details of a line item – purchase order ID, product ID, unit price, number of units purchased, discount, total purchase value (in currency units) – are *fixed*. Such data is best modeled as a special-purpose table (LineItems) where these attributes become columns: the additional layer of indirection with an EAV design affords zero gains and only yields massive inefficiency, especially considering that the database can eventually accumulate millions of such rows.

4.4.4.6

Naturally Row-Modeled Data in Biomedicine: Genotyping

In the area of bioinformatics, *genotyping data* is an example of naturally row-modeled data. Humans inherit their genetic material, DNA, from two parents, so that every segment of DNA that one would care to characterize exists in two copies, maternal and paternal. (The exception is that for males, the X chromosome comes from the mother, and the Y from the father.) The role of variations in genetic segments – as the cause of differences in appearance or disease susceptibility – has been known for a long time.

While a segment can have multiple variations in DNA sequence that are of clinical significance – the total number runs into the millions across all segments – an individual human can have at most two variants per segment (which may be identical if the mother and father had the same variant). The term “genotype” refers to the genetic constitution of the individual. A table that stores genotyping data for humans has the structure: Patient ID, genetic segment name (or ID), variant 1, variant 2. The genetic segment is the “attribute”, and the two variants represent the “values”.

4.5

Aside for Software Developers: Should You Use EAV Tables or Hierarchical Key-Value Pairs?

The non-technically inclined may wish to skip this sub-section, but I have included it because in certain circles this topic is still the focus of religious wars and misinformation.

Both approaches share an underlying principle – that key-value pairs represent units of information – but use very different designs. It is important to provide some historical perspective. The hierarchical key-value-pair approach, as I’ve stated earlier, was originally implemented in systems built using MUMPS. MUMPS (for Massachusetts Utility Multi-Programming System) was devised in the 1960s in Massachusetts General Hospital, Boston at a time when computer RAM was measured in kilobytes. At the time, it had several innovative ideas, notably the ability to perform indexed disk-based data access with smart memory-caching, with almost no effort on the developer’s part – you merely defined a “global”, and the runtime environment took care of everything. Further, its implementation as an interpreted language made software development considerably faster than the alternative for business-style data processing, COBOL.

The most notable MUMPS success, in terms of spread of deployment, was as the foundation for the EMR, the Decentralized Hospital Computer Program (DHCP, later renamed to Vista). Coupled with its employment for other EMRs (it was also the original basis for the Epic Systems EMR, one of today’s commercial-EMR market leaders) these successes led to the erroneous belief, in some circles, that MUMPS was especially suitable for medical applications, almost a predictor of success in this area.

4.5.1

MUMPS: Myth and Reality

In reality, MUMPS as a programming language has several design limitations that make its use today for large-scale implementations developed by a team of programmers challenging, to put it mildly. The language has not evolved significantly over the last 30 years. Today’s Java, C# or Visual Basic programmer would find working with MUMPS not much pleasanter than being afflicted with the childhood viral disease that shares its somewhat unfortunate name. The MUMPS programming language resembles the original 1960s BASIC, but has some idiosyncrasies that may have made sense in the 1960s but seem archaic, or only make programming code hard to read today. For example:

- Most of the built-in reserved words can be abbreviated to single characters; the number of spaces between tokens is significant. (See the Wikipedia entry on MUMPS⁵ for an example of code.) This may have made sense in the days when RAM was measured in kilobytes: today, it merely impairs code readability by forcing a mental translation step.
- The language’s limited structured-programming constructs make the liberal use of GO TO statements mandatory. Excessive use of the GO TO statement has been considered

harmful ever since the computer scientist Edgser Dijkstra's pointed out its problems in a famous 1968 Letter to the Editor.⁶ Specifically, it makes programs significantly harder to understand and debug, and therefore more error-prone. Some modern languages such as Java have in fact eliminated GO TO from the language, introducing alternative, safer constructs that make its use unnecessary.

- Like the original FORTRAN and BASIC, MUMPS does not require variables to be declared in advance of their first use. For small programs, this is "convenient" in saving a few lines of typing: for programs of non-trivial size, it makes the very common class of programming errors due to misspelled variables (e.g., `Clinical_Endpoint` when `Clinical_End_Point` was the symbol intended) much harder to catch. Rather than being flagged as an error at code-writing/compile time, a misspelled variable is simply treated as a new variable, resulting in subtle errors that require extensive debugging to trap. One of the costliest software bugs ever, the failure of the Mariner-1 unmanned fly-by of Venus, falls into this class of errors: this bug has been referred to as "the most expensive hyphen in history".¹⁰
- Finally, in a modern programming language, what the core language does or doesn't do for you is less important than the richness of the subroutine libraries that accompany it. The popularity of Java and the .NET languages is in large part because the vast number of class libraries available (built-in as well as third-party open-source or commercial) save the programmer the trouble of reinventing any number of wheels. M has fallen greatly behind, and the gap widens with each passing year. While the M Technology FAQ⁷ would have you believe that the language is still vital, many companies appear to have seen the writing on the wall: an Israeli software company, CAV Systems, even offers software called JUMPS that converts "legacy" (their phrase, not mine) MUMPS code to Java with calls to an RDBMS.⁸ Like Douglas MacArthur's old soldiers, MUMPS programmers, if not dying, are fading away. The fact that the MUMPS FAQ was last updated in 1999 says it all.

The above EMR successes are mostly due to the implementers, who were remarkably talented and driven persons who understood the medical domain thoroughly: arguably, such individuals might have designed good systems no matter what toolset they were using. Excellent software has been built in the past using assembly language – the original 1981 Turbo Pascal compiler, written by the renowned Anders Hejlsberg, the creator of C#, springs to mind – even though this would hardly be considered the tool of choice today. While the choice of programming language/environment can definitely make a difference in productivity – a poor toolset can reduce the output of a talented team – the world's best toolset cannot salvage a bad system design, or make a significant difference to an undisciplined or mediocre team that does not understand the problem domain, or one that lacks effective leadership or team cohesion.

Further, the extent and nature of the early MUMPS successes must be qualified: the aspects of MUMPS that facilitated relatively rapid development of the VAMC EMR have now made the code base considerably more difficult to maintain: the VAMC is in the process of overhauling the code-base to use a Java/RDB architecture, and the process has been a slow and painful one.

4.5.2

Hierarchical Key-Value Pairs as a DBMS Foundation: Caché

To be fair, hierarchical key-value pair storage is quite a neat idea – though it lacks the genera-purpose versatility of a relational database engine – and MUMPS’ design principle of making such storage accessible to the programmer with minimal effort was highly innovative at the time when it was introduced. InterSystems Caché⁹ is a transaction-oriented database engine based on this approach, which is currently the engine used for most applications originally written using MUMPS (including the VAMC EMR and Epic). To make the use of Caché palatable to today’s programmers, however, InterSystems provides both a SQL-like syntax to allow definition of tables (which can be hierarchically nested within other tables, because under the hood they are implemented as key-value pairs) as well as a dialect of VBScript for traditional programming tasks. Further, they make the data available to external programs (such as spreadsheets and analytical packages) through an Open Database Connectivity (ODBC) driver. The Caché engine also allows the use of structures such as bitmap indexes.

While Caché is a well-designed product, you have to take their marketing blurb (which includes the phrase “post-relational”) with more than the customary pinch of salt. InterSystems’ advertising literature styles Caché as an “Object DBMS”, a term so ill-defined that it has been used by all types of products, including repackaged 1970s-generation pre-relational DBMSs that have since perished. However, if by “Object” DBMS one means a system that lets developers define their own data types as well as operations specific to them (something that the big three – Oracle, Microsoft and IBM – now do with the current generation of their relational DBMS offerings – then Caché doesn’t meet that definition.

Also, Caché calls itself the “world’s fastest database” – but it is more accurate to state that it is very fast only for those operations that hierarchical data structures are known to be very efficient at – such as getting the data for a single patient, an operation that exploits the data’s hierarchical nature. We’ve already stated that operations that are important in research applications, which involve query on the basis of attribute, perform poorly. EMR vendors such as Epic realize this fully: they provide utilities to export Caché data into text files that can be imported into external systems such as data warehouses, which typically utilize a relational or multi-dimensional database engine.

4.5.3

Summary

The bottom line: whether to use EAV with relational databases or hierarchical key-value-pair storage depends on several factors, such as what technology you are most comfortable with (most programmers today have been weaned on relational designs) and the presence of legacy code that must be converted. Whether a given EMR uses one approach or the other should not be a deciding factor in its purchase: vendors such as InterSystems have done their best to make their products inter-operate with other software tools.

It is certainly true that hierarchical key-value-pair storage is effective only for transaction-oriented systems, so you should be on your guard if any vendor tries to persuade you that this approach is good for systems that require ad hoc query capability, such as data warehouses.

References

1. Wikipedia. B+ tree. 2010 [cited 3/2/2010]; Available from: http://en.wikipedia.org/wiki/B%2B_tree.
2. Wikipedia. Hash table. 2010 [cited 3/2/2010]; Available from: http://en.wikipedia.org/wiki/hash_table.
3. Wikipedia. Bitmap index. 2010 [cited 3/2/2010]; Available from: http://en.wikipedia.org/wiki/Bitmap_index.
4. Dinu V, Nadkarni P. Guidelines for the effective use of entity-attribute-value modeling for biomedical databases. *Int J Med Inform*. 2007;76(11–12):769-779.
5. Wikipedia. MUMPS programming language. 2010 [cited 4/10/10]; Available from: en.wikipedia.org/wiki/MUMPS.
6. Dijkstra EW. Go to statement considered harmful. *Commun ACM*. 1968;11(3):147-148.
7. www.faqs.org. M technology: answers to frequently asked questions. 1999 [cited 10/1/10]; Available from: <http://www.faqs.org/faqs/m-technology-faq/>
8. CAV Systems Ltd. JUMPS: Java/RDBMS from MUMPS. 2008 [cited 11/1/10]; Available from: <http://www.mumps-migration.com/>
9. Kirsten W, Ihringer M, Kuhn M, Rohrig B. *Object-oriented application development using the caché postrelational database*. Berlin: Springer; 2003.
10. Wikipedia. Mariner 1. 2010 [cited 10/3/10]; Available from: http://en.wikipedia.org/wiki/Mariner_1

5.1 Introduction

In the days prior to computerization, paper was the primary means of recording the details of an encounter of a patient with a care provider. Even after the advent of the EMR, paper is still with us: one important function of the EMR is to generate hard-copy intended for the patient or for third parties without EMR access. The traditional paper record for a given patient (placed in a folder) was arranged in a manner designed to facilitate access to information on multiple encounters. Thus, each patient's information was sorted by date (most recent first), and within date by a means of organization represented by the acronym SOAP¹ – Subjective findings (e.g., presenting complaint and symptoms related to it, past history), Objective findings (e.g., results of physical examination), Assessment details (e.g., lab test results) and Plan of action (i.e., orders).

In environments where patients can present with extremely diverse complaints – such as primary care or emergency medicine – the least (electronically) sophisticated implementation of SOAP essentially involves using the EMR as a glorified word-processor, where the clinician records information as unstructured, narrative text. From the perspective of successful patient outcome, such an approach demands a significant level of knowledge and competence on the part of the care provider, who must recall and document the findings that are relevant to a workup of a given complaint: even the absence of certain findings must be recorded.

To recall what I stated in the earlier chapter on representing structured clinical data, *protocols* – standard means of patient management, including information gathering – are employed for common clinical conditions. One major advantage of protocols is that their systematic use minimizes variances in patient outcomes due to the non-uniform distribution of caregiver experience and expertise. When protocols are employed, the majority of the information that is captured is structured.

We now consider the requirements of a user interface intended to facilitate protocol-based data capture. The two most important requirements are speed and ease of use. Clinicians, who are not always facile with the keyboard, already complain about the significant additional burden imposed on their time by having to enter prose into an EMR. If we are going to capture structured information – dates, numbers, enumerated values – that can be analyzed later, we want to make sure that data capture is *faster*, not slower, than for narrative text.

At the outset, I want to emphasize the reasons why the highly simplified version of the Entity-Attribute-Value (EAV) Data Model discussed in the previous chapter (which considers only how the data is *stored*) does not suggest the basis of a user interface.

5.1.1

Pitfalls of a User Interface Metaphor That Directly Reflects EAV Storage

I've stated earlier that there are hundreds of thousands of possible clinical findings across all branches of medicine. Consider a hypothetical user interface intended for structured data capture that corresponded directly to the EAV representation. In such a user interface, a clinician would have to first specify an attribute/finding of interest using a keyword search. Search would possibly be assisted through a Google-like type-ahead feature where entering a partial phrase showed a list of all matching phrases. After selecting the desired attribute, the value corresponding to the attribute would be entered. These two steps would be repeated for each parameter of interest.

Even a limited experiment would demonstrate that such an interface is unusable:

- It does not solve the problem of clinician expertise variance: the interface still requires a clinician to *remember* the parameters that are relevant to a particular clinical condition, in order to search for them. An electronic protocol should be capable of being employed by less knowledgeable individuals such as medical students or other trainees with modest effort.
- Medicine is rife with synonyms, which complicates locating a parameter of interest. The EMR must also store all possible synonyms of each parameter and allow search by these. Alternatively, the clinician is required to specify multiple synonyms manually, and even then, may often fail to find the parameter of interest in a reasonable time.

Consequently, in production scenarios where tens of thousands of attributes exist, the clinician-user would spend far more time locating the parameter of interest than in entering the corresponding value. Data entry would become a distraction from the primary task of recalling and recording relevant findings comprehensively. Conscientious clinicians might find that the only way to forestall this eventuality is to write information down on paper first before attempting to enter it into the EMR, which increases the data-entry burden even more.

The above hypothetical scenario is a kind of *reduction ad absurdum*: to the best of my knowledge, it has never occurred in real life with a production-mode EMR. (Similar situations have, however, occurred with some clinical order-entry systems in the 1980s, where the mandated use of dysfunctional software provoked medical residents to go on strike.) The basic EAV model must therefore be augmented by metadata that is concerned with the *presentation* of logically related sets of attributes to the user.

5.1.2

Representing the Protocol

For structured, protocol-based data capture, the requirements of speed, ease of use, and accuracy can most readily be met by a “fill in the blanks” user interface that is the electronic analog of the paper protocol form. The individual parameters applicable to a given condition – e.g., a heart attack – are already laid out as fields in SOAP sequence, preferably accompanied by help for each field (either as part of the form, or accessible in a context-sensitive manner).

Therefore, an important aspect of electronic protocols representation is the definitions of relevant clinical parameters for each protocol, and the electronic data-entry/browsing forms that present parameters in a logical sequence. The challenge is how to transmute the data presented to the user in such forms to and from the relatively non-intuitive internal representation of the EAV data model. This chapter will describe how these are represented as metadata, and how such transmutation can be implemented.

Sophisticated protocols may also have complex computations that execute when particular conditions that may apply to a patient are met – for example, when certain laboratory parameters exceed predetermined thresholds. We discuss the latter issue in the next chapter where we consider representation of executable knowledge. In Chapter 14, I’ll show that the design principles described here also apply to *clinical research information systems* (CRISs), though with some significant differences, due to the somewhat different functional requirements required in research scenarios.

5.2

Building the User Interface

The major software effort in implementing protocols goes into building of a user interface. There are two ways to go about this: programming interfaces using the GUI-building tools available to you (e.g., Web-page designers), or building a framework to generate most (or all) of the interface. We consider each approach in turn.

5.2.1

Hand-Coding of Interfaces

This approach involves sketching out the requisite functionality of the various forms that will be required, programming the user interface from first principles, and testing each form to ensure that the requirements are met. The hand-coding approach can only be recommended if you are part of a software team that is being paid by the hour, irrespective of the quality of work that you do. My reasons for deprecating this approach are stated below.

1. User-interface specifications change quite often in biomedicine because knowledge advances frequently. More important, as stated earlier, the number of forms required of a complete application run into the hundreds, and user interfaces to new data are often required to be developed within a day or two: the luxury of being able to program an interface to a specification at a relatively leisurely pace rarely exists.
2. The approach is fragile. Coding a user interface to a specification and testing the same on a form-by-form basis is repetitious, intellectual grunt-work that unfortunately also happens to be error-prone. The task of form-development relies entirely on human diligence, and so opportunities to learn from previous experience do not exist or are largely squandered.
3. Irrespective of whether the underlying data is stored in entity-attribute-value (i.e., row-modeled) structure, or in traditional columnar (one column-per-parameter) structure, the electronic form must essentially provide the illusion of column-modeling. In other words, the end-user should not know or care what attribute-value data representation means. Only through automation is it possibly to implement such software sleight-of-hand robustly.

5.2.2

Automatic Interface Generation

The alternative approach to hand-coding is to use the specification itself as the basis for automatically generating a functioning electronic form with embedded code within it. In other words, the specification must be *computable*: for the purposes of this book, the specification must be recorded using multiple related tables in a relational schema. *The set of tables that store the specification constitute a metadata subschema.*

The metadata constitutes one of the critical components of an interface-generation framework, whose high-level architecture is illustrated in Fig. 5.1: the legend accompanying this figure describes what the other components are and how the components work together.

Building a framework is initially much harder to do, but the subsequent gains in productivity can be dramatic. The gains come about mainly because, as the number of executable components to be developed grows, the time taken to develop the framework is more than amortized by automatic form generation, and in part because the computable specification can be developed by a “power user” who works with clinicians. Such a power user has some understanding of the field of medicine (though not necessarily to the same level of depth as a clinician), and is typically not a full-fledged software developer. Such individuals, typically with Bachelors’ or Masters’ degrees in life science fields such as public health or nursing, are almost always much less expensive than either clinicians or skilled programmers.

If the framework is made powerful and flexible enough, even professional software developers will find it faster and more convenient to specify forms by creating metadata through a GUI and then generating forms instead of programming forms from scratch. This approach exemplifies “creative laziness”, a point I emphasized in Chap. 1. Our experience is that the efforts of skilled software developers are best focused on enhancing the

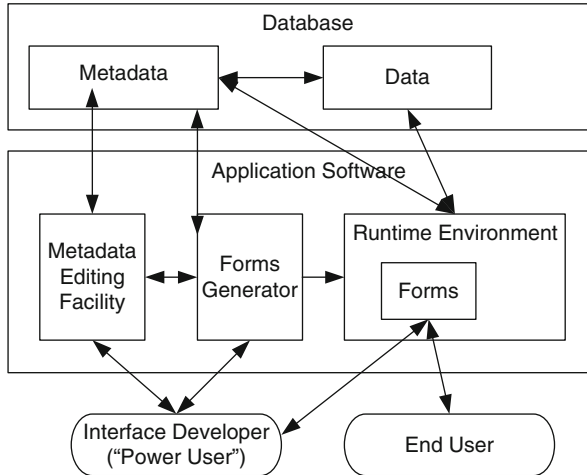


Fig. 5.1 The architecture of a framework to support metadata-based interfaces. The framework consists of a *database layer* (metadata, and data that references the metadata elements) and an *application software layer*. The latter consists of three GUI-based components: a *metadata editing facility*, a *forms generator*, and a *runtime environment*. Power-user-interface developers (who know the biomedical domain, and are not necessarily programmers) create and modify metadata, and use it to generate forms (typically Web pages containing embedded code). The forms are copied to the runtime environment, which contains, among other things, a subroutine library invoked by the form. End-users interact only with the runtime, and are unaware of the first two software components. *Not shown*: The database layer consists of a subcomponent (fictitious protocols with fictitious patients) that is used only for testing the forms before deploying them

framework – specifically adding more features based on user requests, implementing the library routines necessary to support these features, as well as making those features easy to specify and harder to specify incorrectly. The last is achieved by performing extensive consistency checks and providing informative error diagnostics – a point I will address later.

5.3

Metadata for Form-Generation Frameworks: Requirements

Effective metadata design is all about devising computable structures that represent a general, reusable solution to a problem. For this purpose, it is important to consider some of the basic requirements of robust user interfaces. Specifically, the interface (and hence the metadata that is used to specify the interface) must support the following kinds of functionality:

- Interactive, real-time validation of the data that the user enters, to *trap* erroneous input;
- Facilitation of rapid and accurate data entry that *minimizes the likelihood* of making mistakes.

I will first address the various requirements, and later describe an example schema (based on TrialDB,² the clinical study data management system developed by the author's group) that captures these requirements. The original paper describing TrialDB, initially called ACT/DB, is.³

5.3.1

Ensuring Validity of Data Input

Input accuracy can be ensured in the following ways:

1. *Limiting what the user can enter by presenting only available choices.* Parameters based on the enumerated data type are presented to the user either as a pull-down list or a set of radio buttons. This is both an ergonomic aid as well as a means of ensuring data accuracy for enumerated parameters.

The choice between pull-downs versus radio buttons depends on the circumstance: radio buttons take up more screen real-estate, but are faster to operate with the mouse, because the pull-down step is omitted. For keyboard-oriented users, however, pull-downs are faster because the user can tab into the field and type in the first letter to go quickly to the appropriate choice. Most non-Web interfaces support incremental type-ahead, but with Web browsers type-ahead does not work: you have to keep typing the same first letter to go to the next choice.

2. *Validation at the individual field level* includes data-type based checks for dates and numbers, range checking, regular-expression checks for text, spelling check, and mandatory-field check (blank values not permitted). For range checks, it is important to distinguish between a range that defines the valid values (i.e., a value outside this range is impossible), and a range that defines *reasonable* values, where a value outside this range is *suspicious* and should be warned against, but which may still be accepted if the person entering the data overrides the warning.

Parameters of the text data type are validated as follows:

- *Regular expression* checks ensure that the text matches a particular pattern and/or length (e.g., phone or social security numbers, or letters from a particular set of characters, such as DNA sequences). For example, the pattern nnn-nn-nnnn, where n is a digit and “-” is the hyphen, defines a valid social security number. We have simplified things in this example, because there is a special syntax used to define regular-expression patterns, which are very versatile and powerful. Jeffrey Friedl's book, “Mastering Regular Expressions” (O'Reilly Media) is an excellent text. Regular expressions require modest effort to learn, but comprise an essential part of the software developer's toolkit: almost all modern programming languages support their use, and for those who find them too hard to remember, there are several websites that offer “cheat sheets” for commonly used patterns.
- Narrative text can be checked with interactive spelling checkers, which are often freely available and work on Web pages: the better plug-in tools, such as Google Toolbar, become more useful the longer they are used, because they allow an individual user to customize one's spelling dictionary.

3. *Computed values*: the values of certain items are not allowed to be edited, but automatically calculated through a formula based on other parameters in the form whose values have been previously filled in. I will later demonstrate an example of body surface area, which is a function of a person's height and weight: body surface area is used to compute the dose of a variety of anti-cancer drugs.
4. *Cross-field validation*: Here, the developer specifies one or more conditions that must be true of the captured data. A well-known example in laboratory medicine is that the sum of the components of the differential white blood cell count must be equal to 100 (since they are all percentages). Under many circumstances, it is necessary to support validation checks as well as computations across the entire database, not just the current form.
 - For example, the reasonableness of certain values (such as height and weight) depends on the patient's age, which in turn is computed by calculating the difference between the date the value was measured and the patient's date of birth. Date of birth is stored against the patient's demographics data, and one should not force the user to enter it again in the current form just to enable the computation.
 - An example of a cross-field validation check that requires checking another part of the database is a rule that states that a particular date must always be greater than the date the patient was first seen. The latter in turn must be obtained from the history of clinical events associated with the patient, which is again stored in a separate part of the database schema.

Alerts are closely related to validation in that a similar mechanism can be used in some circumstances. In an alert, when a condition related to the patient's data is true, e.g., a value of a laboratory parameter crosses a critical threshold, the system performs an action – at the least, a message to the user. (The triggering condition, however, may be much more general than something activated by data entry.) I'll look at alerts in greater detail in the next chapter.

5.3.2

Facilitating Ergonomic Operation

The techniques described here speed up data entry, and also prevent the user from taking actions that might result in a mistake.

- The use of *default values* for certain fields can speed data entry by saving the user the effort of specifying what may be the commonest response. For example, if the user has to enter geographical location, the default value of state and country would be that where the institution is located.
- *Skip logic* is employed when a particular response to a given question (e.g., an answer of Yes to a question about past history of cardiovascular disease) causes certain subsequent questions which would ask for more details about this condition to be disabled or to become invisible if the user responds to the initial question with a No, because they are now inapplicable. See Fig. 5.2 for an example.

Date of Interview:	Mar-15-2004
Interviewer	Kevin R. Choi
History of hypertension?	1 : Yes
Hypertension (hist), description	Patient has a history of hypertension with 180/10 BP
History of Cardiac Disorder ?	1 : Yes
Cardiac disorder (hist), description	Arrythmia, premature ventricular beats
History of Bronchopulmonary disease?	0 : No
Bronchopulm disease (hist), description	

Fig. 5.2 Part of an electronic form, illustrating pull-down lists and skip logic. The “description” field following the previous “Yes/No” question is disabled (and grayed out) if the user responds with a “No”

- In most cases, when a skip occurs, it is desirable to *disable* the dependent field/s by making them non-editable (and optionally specifying a value that an individual dependent field can take when it becomes non-editable), while preserving the layout of the form, so that the captions of the dependent fields are undisturbed. Providing visual feedback by graying the background of the dependent field is a common means of providing user feedback.
- In some cases, however, when a large number of fields (e.g., an entire section) must be disabled after a skip, it may be preferable to render the entire section *invisible*. On Web pages, this means that the form shrinks: i.e., the invisible section does not take up any blank space. The approach of utilizing visibility is best employed in *reverse*, in combination with default values. That is, rather than display a lengthy form whose contents shrink based on the user’s responses, the initial value of the independent field is one that causes the dependent section to stay *hidden*, so that the form appears brief: changing the value then makes a previously hidden section appear. This creates the (correct) psychological perception that the user’s data-entry load will typically be small.
- *Optional Fields*: These often occur in the CRIS scenario, especially with laboratory test panels. The same form is used by multiple groups of users in different clinical studies, but each group wishes to work with only a subset of the parameters on the form, because the other parameters are not relevant to the research theme of their study. Non-relevant parameters are therefore required to be hidden.

The hiding mechanism is similar to that employed for skip logic, except that the parameters that are shown depend on the current study that a user is working with, and the visibility of the parameter fields does not change dynamically. The metadata required to support the optional-fields feature is a table that contains three columns, which record only the optional parameters for that form in the study: study ID, form ID and parameter ID. (If a given Form ID/Study ID pair has no entries in this table, it means that no subsets exist.)
 - *Online Help*: While the assertion that few users read online help is probably true (“when all else fails, read instructions”), focused and concise help that is specific to the situation at hand – that is, *context-sensitive* – is indisputably useful. The first step towards designing context-sensitive help is to associate individual units of text with the individual user interface elements. Help is of several kinds. *Explanations* tell you what a particular

feature is intended for and what you can do with it. Many electronic forms come with specific *instructions* as to a particular way to proceed in a given circumstance.

In many clinical research situations, the person doing online data entry is also the person eliciting information from the patient.

In circumstances where variation in the way questions are phrased can affect the nature of the patient's responses, the interviewer is expected to follow a script, which consists of prose that must be uttered more or less verbatim. Script text is much more useful during data capture than during data review: in the latter circumstance, it must typically be out of sight, but available at the click of a button.

- *Dynamic (Conditional) Lists*: Certain pull-down lists (i.e., fields of the enumerated data type) may change their contents based on the user's selection from a previous list. For example, the National Bone Marrow Donor Program screening form asks about the broad indication for bone marrow transplant: based on the indication chosen, the user will be asked to specify a sub-indication. There are about 12 indications, with an average of 10 sub-indications for each indication, each of them very specific to the sub-indication. The original paper form runs to more than 10 printed pages, with instructions such as "If you chose Hodgkin's disease for the indication, go to page 6".

When implementing an electronic version of such a form, it is important to remember *that the electronic version should not be a slavish reproduction of its paper counterpart, but an improvement if possible*. (Features like skips, for example, cannot be implemented on paper, except with instructions, such as "if you chose No for Question 2, skip to Question 5", and users may make mistakes on the paper despite such instructions.) Dynamic lists reduce the large paper form to 2 pull-downs: the second pull-down changes its contents based on the value chosen in the first pull-down.

Another use of dynamic lists is in *anchored rating scales* to determine the severity of a clinical condition. In an anchored scale, the clinician/user is asked to specify a particular finding that the patient manifests (e.g., a particular adverse effect of drug therapy) and then rank it in ordinal fashion, e.g., by assigning it a value on a 1-to-5 scale: to eliminate inter-rater variation, the definition of what each number represents is specified in considerable detail. The term "anchor" refers to the fact that a definition is provided: this definition, of course, varies depending on the finding. For example, the Common Toxicity Criteria (CTC) developed by the National Cancer Institute for the rating of adverse effects due to cancer therapy specify that a Grade 1 fever is one where the patient's body temperature is above 37°C but less than or equal to 39.0°C. Given that there are some 300 different clinical conditions in CTC, it is not reasonable to expect the clinician to remember all anchoring definitions. When the clinician selects "Fever" as the clinical condition to be recorded, the software must not only prompt for the grade, but specify what each grade implies.

5.3.3

Protocol-Subcomponent Metadata: Consistency Checks

It can be seen that operations such as skip logic, computed formulas and cross-field validation depend upon our being able to treat an electronic form itself as a single unit. Skips and the majority of computations and cross-field validations operate within a single form. Electronic forms often change – parameters may be added or deleted, or their definitions

modified, and it is important that the software framework be able to ensure that all the above operations continue to be valid.

For example, if a parameter is removed from a form, any expressions that specify computations or validations that may include the removed parameter will now be invalid, and this situation must be immediately reported to the form designer. This does not mean that the removal must necessarily be prevented – in most form-development interfaces the designer is typically allowed to work with only one parameter at a time, and may indeed intend to modify the expressions that incorporate the removed field. However, before the form is actually deployed either for testing or for production use, all consistency checks must be executed, so that deployment can be prevented if inconsistencies are found.

5.4

A Metadata Sub-schema for Interface Generation

We now describe a metadata subschema that represents the individual elements of a protocol, as well as validation/ergonomic requirements. The tables we provide are simplified counterparts of those in TrialDB. We have simplified them in order to limit the discussion to the issues raised above. In practice, there are numerous additional aspects of clinical study data management that are necessary to model in metadata, such as designer preferences for form cosmetics (screen appearance, colors and fonts), facilitated access to the contents of a variety of controlled clinical vocabularies and occasionally direct data-entry by computer-literate patients (for forms that contain self-reported data), and reuse of the same form across multiple studies with minor changes.

The schema consists of numerous tables, and it is the developer's responsibility to build a friendly, GUI-based front-end to allow the designer to populate these tables (and trap a variety of possible mistakes). While my own work has used Microsoft Access, there are several other tools available. I strongly recommend, however, that (as of the current time of writing) you don't attempt to implement a Web-based designer front-end. In terms of number of lines of code, the designer front-end is the largest component of the framework, and today's Web-development tools don't allow the rapid prototyping of database front-ends that tools such as MS-Access, Visual Basic or even the venerable PowerBuilder enable. Your designer interface will undergo multiple cycles of revision based on designer feedback, and you want each cycle to be as fast as possible.

I will later discuss code-development aspects, but I won't provide code here. Since, however, the code follows logically from the metadata design, it is critical to consider the latter.

5.4.1

Definitions of Form Elements

An electronic-form definition consists, at the most fundamental level, of an ordered list of parameters, with intervening information (such as section breaks or sub-headings)

to facilitate presentation. However, certain refinements are required to this basic model.

5.4.1.1

The Need for Intermediate Groups of Parameters

While one could create one giant form consisting of all parameters of interest, the advantage of creating sub-divisions – *logical groupings* of certain parameters – include the following:

- Some parameters are commonly measured and recorded together – for example, standard laboratory test panels such as hematology or serum electrolytes – and treating these groupings as units in their own right simplifies the design of protocols by allowing the utilization of larger reusable units, rather than having to construct a new protocol a parameter at a time.
- The online help text associated with a parameter group is separate and distinct from that which applies to individual parameters.
- While an electronic form may consist of multiple disparate items, a set of parameters belonging to a logical group typically has a single *time-stamp* (or a time-stamp pair) associated with all the parameters, such as the time a blood sample was collected. Figure 5.3 shows a partial screen-shot of a form used to capture previous details of therapy for a cancer patient.

Here, an individual patient may have had zero or more episodes of surgery or chemotherapy, and the summary details of each episode must be captured. Sets of parameters whose values repeat a fixed or variable number of times in a form constitute a *repeating group*. Repeating groups are specifically used to simulate many-to-one relationships in form data when the parameters are stored in EAV form.

In clinical medicine, most events have a single time-stamp, representing when the event happened – for example, for laboratory parameters, this is typically the date/time the sample was obtained from the patient. In the parlance of temporal databases, such events are termed *instant* events. Some events, however, have two time-stamps, representing the start and end date/times: these are called *period* events. An example is an intravenous infusion that may go on for several hours: a chemotherapy course may last for several months. Very rarely (as for family history data occurring in the remote past), time-stamps may be *omitted*.

Therefore, to facilitate reuse, it is worth introducing an intermediate level – Groups – between a Form and the Parameters in it.

Tables that capture metadata on the three major units of the interface are **Forms**, **Groups** and **Parameters**. While the structure of the tables representing these units is shown in Fig. 5.4, Fig. 5.3 shows how these are used in practice when used for data gathering. This shows an instance (for a particular patient) of a form, “Prior Therapies” (form caption, patient ID, etc. not shown for reasons of space) that contains data for two groups: Prior Surgery and Prior Chemotherapy. The individual columns in the groups – e.g., Type of Surgery, Best Response to Chemotherapy, etc. correspond to data for individual parameters. (The “Date” columns in both groups are not actually based on parameters, but we’ll discuss that issue later.)

Prior Surgery

Surgery Date	Hospital/Facility	Type of Surgery	Outcome of Surgery:
Jun-1-2007	Yale Cancer Center	Excision node biopsy	2 : Resected
Feb-1-2006	Sloan-Kettering	Lymph node biopsy	1 : Biopsied

Add recordDelete record

Prior Chemotherapy

Start Date	Stop date	Hospital/Facility	Chemotherapy Agent/Regimen	Number of Courses	Best Response to Chemotherapy:
Feb-15-2006	Feb-9-2007	Sloan-Kettering	MOPP	4	1 : Complete Response

Fig. 5.3 Repeating Groups of parameters within a form. Unlike in Fig. 5.2 the same set of parameters – here, related to previous surgeries or previous chemotherapy for a cancer patient – can occur zero or more times within a form, depending on the individual patient. Repeating-group data simulates a many-to-one relationship

5.4.1.2
Summary of the Essential Tables

These tables all use a machine-generated long-integer (32 or 64-bit) *primary key unique identifier*: Form/Group/Parameter ID. In addition every item has an *internal name* (that follows programming-language conventions – typically only letters, numbers and underscores are allowed in the name) and a default *caption* that is shown in the user interface. (In the subsequent chapter on clinical information research systems, I will illustrate circumstances where the same item can present different captions to different users.)

Only the captions are visible in the end-user interface of Fig. 5.3; the IDs are not visible – the end-user doesn’t need to care about them – but are present in the code behind the form, and the internal names are not needed (the IDs suffice to access information in the database).

The schema is shown in Fig. 5.4. The list of fields in each table has been deliberately stripped down: in a production design, there are additional fields such as Script, Instruction, Documentation, Current Version Number, Date Created, Date Last Modified, User who created the record, and User who last modified it. The last five fields are required for minimal tracking of changes: in more sophisticated systems, there are ancillary tables that will maintain an audit trail of changes to the contents of these particular tables. (At the very least, an audit-trail table records which row and column were changed, the previous value of that column, who changed it, and when.)

There is a many-to-many relationship between groups and forms, and between parameters and groups. This is because, while there is a generally hierarchical relationship between the three units, the same Parameter is often used in different groups, and the same Group is often used in multiple Forms To accommodate this situation, there are two “bridge” tables: **Group_Form** connects Groups and Forms, while **Group_Params** connects Groups and Parameter. (As I’ll discuss in a subsequent chapter, this is more of an issue in CRISs, where different researcher-users focus on different subsets of the very large universe of parameters in the context of different studies, than in EMRs, where the

relationship tends to be more strictly hierarchical.) The bridge tables record the *sequential ordering of groups* within a form, and parameters within a group respectively. This information, of course, determines the form's appearance when generated. I will not discuss the bridge tables further.

We have also included the **Choice_Sets** and **Choices** tables that were introduced in the chapter on Data types for the representation of parameters based on enumerated values. The field `Choice_Set_ID` in the Parameters table links Parameters with Choice_Sets: multiple parameters may be based on the same Choice Set.

A brief description of the columns in these tables follows.

5.4.1.3

Forms

- *Datetime_Last_Generated*: This tracks when the form was last generated and deployed in production.
- *Choice Presentation*: Enumerated. The default presentation (pull-downs, listbox or radio-buttons) for enumerated parameters.
- *Hide_Skipped_Parameters*: Boolean. If True, then skip logic operates by hiding rather than disabling the inapplicable parameters.

5.4.1.4

Groups

- *Repeating Group*: Boolean. If true, the group is presented as a sub-form, as in Fig. 5.3, to allow multiple instances of the parameters in that group.
- *Event Type*: Enumerated. Instant, Period or Omitted.
- *Event Start Label, Event End Label*: The captions associated with the start/end of event when presented in the form. For chemotherapy, these may be "Start Date" and "Stop Date" as in Fig. 5.3. For a group with a single time-stamp such as laboratory measurements, the event end label does not apply, and the start label may be "Date/time of sample collection".
- *Event Format*: Enumerated. Whether only the Date is recorded or whether the time is also recorded as part of the Date.
- *Allow future Event Start, Allow future Event End*: Boolean. In most clinical record-keeping, dates in the future are an error, but they may be allowed in exceptional cases: e.g., they represent scheduled dates.
- *Show rows in reverse chronological order*: For repeating group data that has multiple rows, the same form is used over time and additional rows of data are added as needed. When such a form is reviewed, it is typically most convenient to present the row with the most recent time-stamp topmost, because recent data is the most important for patient management.

5.4.1.5
Parameters

This table has numerous columns that serve two main types of purposes: validation and presentation.

Columns Related to Validation

- *Data Type*: One of: integer, string, long text, datetime, real, Boolean, enumerated, ordinal, binary (BLOB). Data types are described in the chapter on data types in the medical record. Based on your needs (e.g., if you have to manage a large number of XML-formatted documents) you may also choose to add “XML” to the above enumeration.
- *Decimal Places*: For decimal numbers, the number of decimal places used to input and present data.
- *Choice Set ID*: Used for enumerated data types: links to the Choice Sets table.
- *Mandatory*: Boolean. If true, the field must be filled in.
- *Default Value*: a data-entry convenience, as stated above.
- *Upper Bound, Lower Bound, Upper Bound Warning, Lower Bound Warning*: (text): The absolute and “suspicious” upper and lower bounds for entered values. These values are stored as text even though the data type may be a number or date.
- *Maximum String Width*: For text fields, the maximum number of characters that are accepted in the field.
- *Regular Expression*: Text. The regular expression used to validate a text parameter. The accompanying column *Regular Expression Error Message* is prose that appears if a user makes a mistake in entering a parameter that is validated with a regular expression: it explains to the user what text is expected.
- *Allow future dates*: For parameters of the Date data type, determines whether dates in the future are permitted (see the previous discussion on event time-stamps for Groups).
- *BLOB File Extension*: Text. Used for binary-large-object (BLOB) data types (images, waveforms, Word documents, etc.) The list of file extensions specifies the types of files that correspond to this parameter. (e.g., a parameter used for images might specify .JPG as the file extension, while a parameter for geometric graphics might allow PNG files.) This serves a check to prevent the wrong type of data from being uploaded. This also serves a display purpose: when the contents of such a parameter are displayed after being downloaded to the user’s machine, the machine knows, through the file extension, which application to use for display.
Maximum BLOB Size in KB: Integer: offers some safeguard against a malicious user from carrying out a denial-of-service attack by attempting to upload a very large file that has been renamed with a permissible extension.
- *Computed Formula*: Text. This is non-blank only for parameters whose values are computed: this column contains the expression defining the computation. I will discuss this in more detail shortly.

Columns Related to Presentation

- *Display Width, Display Height*: Integer. The dimensions (in characters/lines) used to present the parameter. While dates, date/times and numbers will generally have default dimensions that do not need to be specified individually, textual data can be presented in scrollable boxes of desired dimensions (or on a single line) as needed.
- *Orientation*: Enumerated: Used to determine whether an enumerated parameter presented as a set of radio buttons will present these buttons laid out vertically or horizontally.
- *Conditional Choice Set*: Boolean. If true, this parameter's display contents are controlled dynamically by another parameter, as described in section 5.4.6
- *ReadOnly, Invisible*: Boolean. *ReadOnly* (often set to True for computed parameters) means that the user is not allowed to edit their contents. *Invisible* means that the parameter is not displayed to the user. This is used for certain parameters that serve to store the results of lengthy intermediate computations, of which the user need not be aware.

5.4.2

Implementing Computed Fields

I have stated that the Computed Formula column of the Parameters table contains, for computed parameters, an expression representing the formula. I now provide as to how this expression is actually executed (i.e., evaluated) at runtime: this material is for the technically inclined.

The expression is specified using a particular expression language: JavaScript and VBScript, which allow ready interpretation of string expressions with minimal programmer effort, are commonly used expression-language choices, especially for applications that are deployed through Web interfaces.

The expression contains placeholders that are substituted by the values in the form.

For example, the standard Dubois and Dubois formula that computes Body Surface Area in square meters as a function of Height (in m) and Weight (in Kg) would be expressed in VBScript as the following (^ is the symbol for raising to a power):

```
0.20247*[Height_in_M]^0.725 * [Weight_in_Kg]^0.425
```

The placeholders in square brackets are the names of other parameters in the form, and these would be substituted by the actual values entered. The resulting expression is evaluated as though it were program code – VBScript (and JavaScript) have a function called *Eval*, that does this – to produce a result.

Note: in an implementation, you must commit to a particular expression language. JavaScript is supported by all browsers, though VBScript has a somewhat richer set of built-in functions. However, the real strength of *Eval* with either language is that you can include a function library containing your own functions and then use these in the expression. The library is included in the page with a one-line directive, but it is typically stored in the Web browser's cache, so that it is not repeatedly re-downloaded each time you open a new form.

Interestingly, in Microsoft Internet Explorer, which supports both languages, a function written in one language can also freely call functions written in the other language: in our own work, we have implemented two libraries. JavaScript is more cumbersome to work with than VBScript, but has certain additional flexibility, such as the ability to create functions that can take a variable number of parameters, that is occasionally necessary.

5.4.2.1

Security Considerations

In certain languages, such as JavaScript and Perl, *Eval* can actually execute *statements*, or even a sequence of semicolon-separated statements. Such statements can have side-effects on program or even machine state. Thus, if the following perl statement,

```
system ("del *.*")
```

were evaluated, the operating system would attempt to delete all files in the current directory, while the following statement

```
system ("print /etc/passwd")
```

on a machine that used an older version of UNIX would try to list the contents of the password file on the user's screen from where they could be saved into a text file and then attacked for weak passwords using a *dictionary attack*. In a dictionary attack, even though the password file lists the passwords in encrypted form, the encryption algorithm is publicly documented, and so the attacker can try to encrypt every single word in an electronic dictionary using the same algorithm, and check if any of the encrypted text matches exactly to the encrypted password of any user. (This attack was actually used against a website whose naïve creator wished to demonstrate what a wonderful and powerful language Perl was.)

This means that, for *Eval* to be used safely, the text fields that are substituted into the expression must be carefully validated. This is why the basic checks – data type, range, string length, etc. – are so important. I will return to the theme of checking user input in a subsequent chapter.

5.4.2.2

Triggering a Computation: Dependencies

The computation associated with the formula should ideally be triggered in the fashion of formulas in a spreadsheet: that is, as soon as one of the parameter values that it depends on (either height or weight in the above example) changes. This way, the user gets instantaneous feedback. In terms of Web programming, an individual field on a form has an *onChange* event, which fires when the contents of the field are changed and the user tries to leave the field. The developer must create a library function that is associated with every field on the form: firing the event invokes this function. Among other things, this function checks for dependencies of various kinds, and takes appropriate action.

Theoretically, one could parse the contents of a formula embedded in the Web page to determine the parameters that a given computed parameter depends on. However, it is faster to determine these dependencies in advance just once, at form-design time, and embed dependency information in the Web page. The table **Form_Formula_Dependencies** in Fig. 5.5 has three columns, the Form ID (the form in which the formula is placed), the “Independent” Parameter (the one determining the computation) and the “Dependent” parameter (the one being computed). In the case of the body-surface area example, there would be two rows, since the parameter body surface area is determined by both height (independent parameter 1) and weight (independent parameter 2).

Computing dependency information at design time is useful because if a designer accidentally attempts to remove either of the parameters height or weight from a form, the system can report that the formula for body surface area will be invalid.

As in a spreadsheet, computations can daisy-chain: that is, a parameter that is computed may have its value used in turn to compute another parameter, and so on to an arbitrary level of complexity. That is, an independent parameter can also be a dependent parameter: the design of the dependencies table allows this.

5.4.3

Modeling and Implementing Skip Logic

Skip logic operates similar to the computed formula scenario: we have independent and dependent fields: In the general case, an independent field can take several values, and for each value (or a range of values) multiple dependent fields may be disabled. The disabling acts like a toggle: that is, if the independent field’s value is changed again, previously disabled fields may be re-enabled. So here, the dependency is not based only on a dependent field, but upon its value.

The table **Skip_Logic_Items** records, for a given form, which parameters are independent fields. (It uses an artificial primary key which is referenced in the table below.)

Skip_Dependents records the following columns:

- The *form_param_Id* points to the independent parameter.
- *Operator* This is enumerated, with the following possible values: Equals, Not equals, Greater than, Less than, Greater than or equals, Less than or equals, Belongs to, Does Not Belong to, Between. The last specifies a range of values. In practice, Equals is most commonly used.
- *Value1*: Text. The value to be tested against. In the case of Belongs/Does not belong, there is a list of comma-separated values. For the Between operator, a second column, *Value2*, is used to specify a range of values.
- *Disabled_Param_ID*: Points to the parameter that will be disabled for the given independent parameter-operator-value range.
- *Default_Value*: (text). This is optional, representing the value, if any, to which the disabled parameter will be initialized once disabled. (Normally, the disabled parameter value will be erased on disabling.)

Theoretically, we could have combined these two tables into one. Our own implementation used a segregated design because we had numerous forms where a single parameter controlled ten or more dependent parameters. (We have also seen some designs that split the information into three levels, with all columns other than Disabled_Param_ID combined into an intermediate level table and an artificial key.)

As for computations, removing a parameter that disables several other parameters will cause the skip conditions to become invalid. This should be checked at design time. Also like computations, skips can daisy-chain. For example, when a particular value is entered for parameter A, parameter B is disabled and set to a particular value. The change in parameter B may in turn affect parameters C and D.

5.4.3.1

Implementing Skips on Web Pages

As for computations, dependency information is computed at form-design time and embedded on the page. Skip logic utilizes the onChange event just as computations do: in fact, the same library function does double-duty in its dependency checking – checking first for pending computations and then for skips.

The simplest way to implement disabling or invisibility on a Web page is to employ Cascading Style Sheet (CSS) technology: you set the dependent field to a class whose attributes specify disabling or invisibility. The attribute “display: none” both hides a field and redraws the page so that the space previously occupied is recycled. A CSS style definition such as:

```
[disabled] {background: lightgray}
```

will change the background appearance of a field on a Web form when its “disabled” attribute is set.

5.4.3.2

Implementing Skips Based on a Combination of Parameters

Sometimes, it is necessary to implement complex skips, based not on values that a single independent field may take, but based on a combination of values in more than one field. For example, in a community clinic, several fields are to be entered only if a woman with one or more children needs special help – that is, if she is a single parent OR if she is under 19 years old at the time of the encounter.

The simplest way to implement such skips is to define a new (invisible) Boolean field, with a name like “Special_Help_Needed” with a computed formula such as (in the VBScript example below):

```
[Single_Parent]=True OR [Age_Less_than_19]=True
```

and then define skips for each dependent field based on the value of this field.

5.4.3.3

Form-Design Aspects of Computation and Skip Logic

The placement of the dependent parameter in the form must *follow* the placement of the independent parameters. That is, in the Group_Parameters table, a given dependent parameter must always have a higher sequence number than the independent parameter that controls it. This way, the (changed) computed result, or the disabling/enabling of fields appears only *after* the values of the independent fields have been entered.

In Western countries, users work on a form from top to bottom and from left to right. So they are always looking ahead, not backward. If a changed field was positioned before the independent field, a change might not be noticed (or disconcerting if it was, because users don't expect things entered earlier to change). The design of the metadata subschema allows sequence-error checks to be made.

5.4.4

Modeling Complex Validation and Alerts

The table Form_Validation is used to capture complex validation metadata, as well as support form-based alerts. It has the following fields:

- *Validation_ID*: Integer. An artificial primary key.
- *Serial_Num*: The sequence of the check. It is desirable to trap and report the most likely or common errors first.
- *Is_Alert*: Boolean. If true, the row contains information on an Alert, not a complex validation condition.
- *Is_Critical_Alert*: Boolean. If true, and *Is_Alert* is also true, then the alert is intrusive – e.g., it must be acknowledged. It may also write the information to an audit trail. We discuss intrusive vs. non-intrusive alerts in the next chapter.
- *Validation_Condition*: The expression to be validated. We provide an example shortly.
- *Error_Message*: The prose of the message (an alert or validation error message) to be shown to the user if the condition evaluates to True.

Example: As for computations, the validation/alert expression must be valid in a programming language (the same as used for computations). An example of an expression that must be True in a Hematology panel is that the sum of the individual components of a differential WBC count must equal 100, since each component represents a percentage. In VBScript, this might be expressed as:

```
[Polymorph_Neutrophils]+[Band_Neutrophils]+[Eosinophils]+[Basophils]+[Lymphocytes]+[Monocytes]=100
```

As in the case of computations, the actual values in the form substitute for the placeholders, and the expression is then evaluated with *Eval*. If the sum of the six fields above (which are all integers) is not equal to 100, the error message to be displayed is something

like, “The Sum of the individual components of the differential WBC count must be exactly equal to 100”.

5.4.4.1

Implementing Validation and Alerts

Because a condition such as the above depends on the values of multiple fields, it is generally not desirable to interrupt the user while edits are still being made. The logical time to test each condition associated with the form is when the user tries to save the form. In terms of Web programming, one invokes a standard function when the save is attempted.

5.4.4.2

Tracking Validation Dependencies

The table `Validation_Dependencies` contains two columns: the Validation ID (which identifies the condition/alert) and the parameter on which it depends. In the differential WBC count example above, six rows of information (one for each parameter) would be required to store the dependency information.

5.4.5

Implementing Complex Computations and Validations That Access the Database

Some computations and validations require the lookup of parameters whose values are not available in the electronic form with which the user is interacting currently, but are stored in the database. For example, some computations use the patient’s age, which in turn must be computed from the date of birth. In any case, the system must still provide instant feedback.

Since the last few years, allowing code that runs in the Web browser to access a database has become much easier through a technology called AJAX (Asynchronous JavaScript And XML).⁴ AJAX allows data to be sent back and forth between a Web page and a back-end Web server (and thence to a database server) without having to refresh the page. The incremental search feature of Google and Wikipedia, which suggests words when you type in a partial phrase, uses AJAX.

Despite its name, AJAX mandates neither JavaScript (VBScript may be used instead if the browser supports it) nor XML. Many sites use a format called JavaScript Object Notation (JSON),⁵ which has the advantage, for complex data structures, that one does not have to write code to parse the XML: JavaScript can consume a JSON data packet directly using JavaScript’s *Eval* function.

In our own work, we have implemented complex computations by adding to our browser library a set of custom functions that use AJAX to fetch various types of data from the database, and using these functions within an expression that will be used for computation or validation/alerts.

5.4.6

Implementing Conditional Lists

The table `Choice_Sets_Hierarchy` of Fig. 5.4 implements conditional lists. Essentially, one choice set (indicated by the `Choice_Set_ID` column) can be a child of another `Choice_set_ID/code`. The latter two columns are indicated with the suffix `_Parent`. To give a concrete example, suppose a parameter based on a controlling choice set A has three choice codes, 0, 1 and 2, which, if chosen respectively populate a second parameter with the contents of choice sets X, Y and Z. Then this table will have three rows, with values (X, A, 0), (Y, A, 1) and (Z, A, 2).

Dynamic lists are also implemented using AJAX: when the `onChange` event occurs in the parent parameter, its current value is sent to the server, and a list of code/caption data for the appropriate child choice set will be sent back to populate a second parameter field.

A practical issue here is that, when the user saves the form's data, the contents of the child parameter's code cannot be interpreted in isolation, as its meaning varies depending on the parent parameter's choice code. To facilitate interpretation, storing both the code and the associated caption eliminates the need to determine the meaning computationally.

5.4.7

Practical Issues Related to Expressions for Computation and Validation

I've stated earlier that the major benefit of creating a framework is that power-user designers can develop forms, freeing programmers for more critical tasks. However, the designers cannot be totally naïve with respect to the programming language you've committed to using: they must at least know how to write syntactically correct expressions, and must also know how to use the various functions that the developer team has provided in the library.

In our experience, learning to write one-line expressions involving function calls is a lot simpler than learning full-fledged programming (even if the expression is several hundred characters long). We make our own designers' lives easier by providing a detailed online "cheat sheet" that lists the various functions available, along with real examples of their use. We also eliminate the need to enter parameter names in square-bracket placeholders manually – this can be error prone, and so we have built a point-and-click expression builder that inserts a selected parameter name at the cursor position.

We also provide an expression tester that lets the designer test the expression by prompting sequentially for a test value for each placeholder, and then displaying the result. Errors such as extra parenthesis can be caught, though if there are other errors (such as calling a function with the wrong number of parameters), the diagnostics are not particularly end-user-friendly.

Finally, developers are still available if complicated expressions need to be developed, or new functions added to the library to serve a particular purpose.

5.5

Testing Generated Forms and Troubleshooting

Using a framework for generating forms does not eliminate the task of testing the form to ensure that it matches its requirements. Testing must not only be done internally (typically, the power users will do this) but also with the intended users. Errors can and will occur in a generated form, and are traceable to the following causes:

1. An incorrect specification, i.e., incorrectly specified metadata: remember that a power-user operating a framework is essentially performing a kind of programming, but at a much higher level of abstraction. Power-users are also fallible: anyone who has ever written a program knows that software does what you tell it to do, not what you mean it to do.

The process of developing a user interface to metadata is one of constant learning. The software developer observes the kinds of mistakes that users commonly make, and sees which kinds can be caught by implementing additional consistency checks. In the skip logic example above, the serial number of the dependent fields must be greater than that of the independent field. Certain mistakes, of course, (such as the user specifying “greater than” in a relational operator when “less than” was intended) can only be caught by testing.

Our own experience has been that the metadata user interface continually gets more robust with time, provided the framework software developer is sufficiently self-critical. Rather than reflexively blaming the power-user, the conscientious developer tries to determine if there was some aspect of the user interface that confused the user, or whether the user’s mistake should have been caught by a consistency check.

2. Bugs in the run-time operation of the code generated by the framework. In my own experience, which involves generation of Web code, this category of errors is the most time-consuming to develop as well as debug. There are several reasons for this:
 - Much of the Web code generation involves creation of browser-based scripts (either in JavaScript or VBScript). The development and testing environment for these scripting languages is considerably inferior to that available for full-fledged programming languages such as Java or the Microsoft .Net languages. In the early days of Web development, about the only debugging aid available was the Print statement: software developers who had been pampered by high-quality debuggers found themselves traveling back in time to the 1960s. Even today, you tend to discover your syntax errors one at a time, during runtime, because there is no batch-checking compilation phase for the scripting languages. JavaScript is considerably worse than VBScript in this matter, because identifiers are case-sensitive, so you also discover your case-variation-typing errors at runtime, one at a time: it is also riddled with features that its original developers may have considered “cool”, but which, nine times out of ten, only result in subtle, hard-to-catch bugs.
 - Another problem is the considerably different programming models for Microsoft Internet Explorer and Mozilla Firefox, which make development of code that will run across multiple browsers challenging. The only way to preserve your sanity is to

use one of the higher-level frameworks, like JQuery, which hide most of the browser-specific variation from you. (JQuery is now officially endorsed by Microsoft, and supported by Visual Studio 2010.)

The latest version of JavaScript (ECMAScript 5.0, ECMA = European Computer Manufacturer's Association) has a slew of features which make JavaScript more robust, but it's anyone's guess as to how soon mainstream browsers will support it. Some JavaScript code libraries attempt to shield you from the nitty-gritty of individual browser models: Google's Web Toolkit, in addition, allows developers to eliminate the use of JavaScript almost completely by programming in Java instead: the Google Toolkit generates JavaScript from the Java code. However, I've yet to come across a code-generation framework that in turn leverages one of these JavaScript frameworks. (My own work has involved generating mostly VBScript, and is Internet-Explorer specific.)

3. Bugs in the code generated by the form-generation framework, where the power-user's specification is correct. Because the code generator itself is written in a language more civilized than a scripting language, debugging is less of a problem. However, the quality of the generated code is critical. One caveat about code-generation frameworks in general: creating a framework that generates clean code that can be followed readily by a knowledgeable programmer, as opposed to creating a framework that merely generates correct code, is not trivial: this is where the software developer's sense of esthetics comes in, and where really outstanding developers distinguish themselves from the merely good or the mediocre. We have known of code generation frameworks whose output resembles Otto Von Bismarck's characterization of the proverbial sausage and the legislative process.

In general, you should aim to keep your generated code as concise as possible by first developing the run-time subroutine library, and then having your generated code simply make numerous single-routine calls to various parts of the library. Such code is significantly easier to understand, provided that you have also documented the subroutine library thoroughly. This approach also has the advantage that if a bug in the runtime code is discovered, you will have to make the fix in only one place, rather than in the numerous forms that have been generated.

5.6

Form Generation Issues

The output of form generation, a text file that contains a mixture of HTML and code, is particularly difficult for non-programmers to follow, and permitting the form's hand-modification in an HTML editor is almost always a recipe for disaster.) So the idea is to generate forms that do not need modification.

To some extent, this is done by sacrificing pizzazz in the interests of standardization and designer productivity. The layouts generated by a framework are necessarily somewhat "vanilla". Henry Ford used a similar approach to the design of the original Model T, when he joked that you could have it in any color as long as it was black.

Some customizability is, however, desirable, especially if it can be achieved in a straightforward fashion. We have already stated that the Forms table contains additional fields regarding color preferences, as well as the choice of style-sheet based “themes” (which determine font and button appearance, among other things.)

Very often, paper forms contain static text (such as section headings or brief sentences) that is distinct from the captions associated with forms, groups and parameters. It is reasonable to expect the electronic form to correspond reasonably to the paper form (especially if data is being transcribed from paper to machine). While not illustrated in the schema, we support the interpolation of static HTML in particular positions (identified by individual parameters) on the form. Captions of form elements may also contain embedded HTML (e.g., italics or underlines) for emphasis rather than pure text.

5.6.1

Static vs. Dynamic Components

At runtime, a form consists of some elements that are fetched from the database (i.e., dynamic components) and others that are static. In our own work, the contents of fields, element captions, interpolated HTML and the choices for enumerated parameters may be edited without having to regenerate the form, because these are fetched dynamically. (I will explain the rationale of allowing captions and interpolated HTML to be dynamic in Chapter 14) These can be fetched rapidly, with an insignificant performance penalty.

On the other hand, in our experience, the logic for determining dependencies involved in computation and skips can be elaborate enough that computing it dynamically at runtime would be prohibitively expensive. (Our team has designed forms with about 200+ skips, 30+ computations, and about 40+ validations.) The approach we take is to read the computation/skip-logic metadata at form-generation time and generate data structures – primarily arrays and hash-tables – that we initialize and insert into the Web page’s scripting code.

Thus, each page holds a miniature payload of the metadata that it needs to function: this approach bypasses the need for the Web form to consult the server-based metadata to determine its required behavior. This approach, of course, has the modest drawback that changes to the skip logic or computed formulas require regeneration of the form: since such changes are infrequent, we can live with this.

5.7

Capturing Data Edits

I now describe a more basic operation: how, once the contents of a form have been validated, to translate the user’s edits into changes in the database – addition, deletion or updates. While there may be multiple solutions to this problem, we have addressed it using the following approach.

- We first create a sub-schema for clinical data storage that links to the metadata tables described in Fig. 5.4

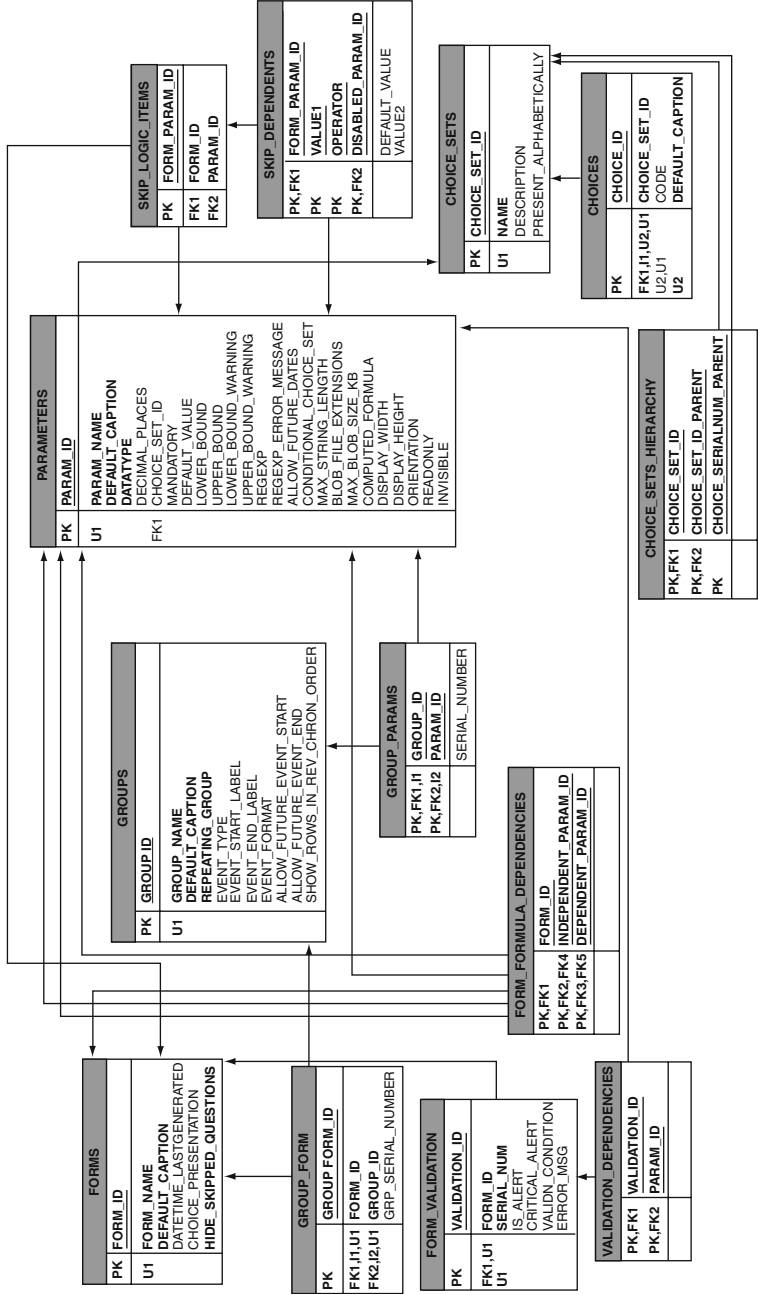


Fig. 5.4 A schema for form elements and validation/ergonomics metadata

- In the generated form, every user-editable element contains associated information that lets the system generate SQL for the correct database operation – so that the appropriate rows and columns in the data subschema is accessed. In Web pages, every element has a property called the ID, which can be used for this purpose.

We now move from the Metadata sub-schema to the Data sub-schema, where data for individual patients is stored.

5.7.1
Clinical Data Sub-Schema

Figure 5.5 shows a stripped-down sub-schema that is used to capture clinical data that is stored in attribute-value form: this is a modified and expanded version of Fig. 4.4 of the previous chapter. In that chapter’s diagram, we deliberately over-simplified the representation of EAV data, treating an EAV row as a triple (clinical event ID, parameter ID, value).

In an actual production design, we must consider the user interface: the design becomes more elaborate because of the need to associate the captured data with the three critical metadata elements of the interface: forms, groups, parameters.

In Fig. 5.5 the clinical events table is unchanged, but now we have two additional tables: **Form_Data** and **Group_Data**. These tables are a direct consequence of normalizing the design of the clinical data. A Form instance, of course, has real meaning to the

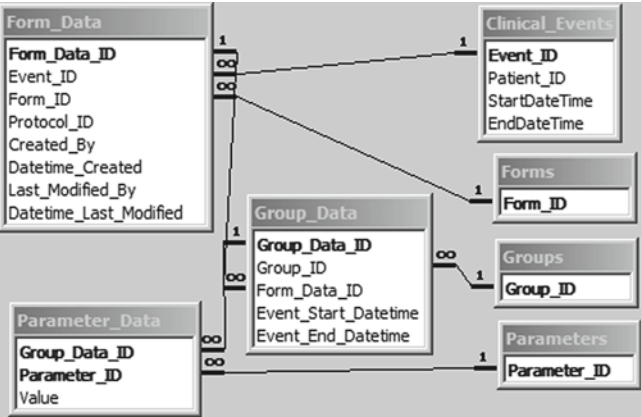


Fig. 5.5 The data sub-schema, which records the clinical data for individual patients. Form_Data represents the segregation of the data into individual forms (e.g., Patient ABC’s blood chemistry of 2/1/2010). A form_Data record has at least one Group_Data record (more than one if the electronic form contains a repeating group). Parameter_data contains the actual parameters in attribute-value form. Note that each of the three data tables links to the corresponding metadata tables – Forms, Groups and Parameters respectively

user who may, for example, wish to inspect the Hematology results for Patient XYZ of 4/2/2010. As stated earlier, a form has one or more groups: there are multiple Group_Data records for a single Form_Data record only when there is at least one repeating group in the form, as in Fig. 5.3.

Both these tables have artificial primary keys, indicated with the table name and the _ID suffix.

5.7.1.1

Form_Data

- *Event ID* links to the Clinical Events table
- *Form_ID* links to the Forms metadata table.
- *Protocol_ID* links to a Protocols metadata table: we have not illustrated or described this table, but it would be expected to contain descriptive information on a given protocol. Since the same form can be used in multiple protocols (this is especially true of laboratory test panels), there would need to be a many-to-many bridge table between Protocols and Forms. CRISs often use *Study ID* as a synonym for *Protocol ID*.
- Other fields capture the date/time of creation and last modification, and the user responsible for each.

5.7.1.2

Group_Data

- *Form_Data_ID* links to the form data table. Remember that a form can have one or more groups, and therefore form data will have one or more instances of group data.
- *Group_ID* links to the Groups metadata table.
- *Event_Start* and *Event_End* date/times record the time-stamps associated with a group date instance. Note: if these are omitted, it is assumed that the Start and End date times of the clinical event apply.
- *Instance*: an integer, which is 1 or greater. For repeating-group data such as illustrated in Fig. 5.3 individual group instances may have instance numbers greater than 1.

5.7.1.3

Clinical_Data

This contains the parameter-value pairs. The “entity” is really the Group_Data_ID: by traversing the links to other tables, one can ascertain other information about the entity, such as the patient, the form, the protocol, etc. In the schema that we use, there are multiple data-type-specific Clinical Data tables. (We have not found the need to create an artificial primary key, as for the previous two tables, because such a value is not used elsewhere.)

5.7.2

Mapping Between Form Elements and Database Elements

Figure 5.5 represents a normalized database design, partitioned into multiple tables to minimize redundancy. (We have seen data schemas where Form_Data, Group_Data and Clinical_Data are merged into one giant table.)

Now look at Fig. 5.3 and use it as a frame of reference. You will observe that a given cell on the form (e.g., row 2 of the Prior Surgery group, column “type of surgery”, value “lymph node biopsy”) is identified uniquely by the combination of Form_Data_ID, Form_ID, Group_ID, Instance number and Parameter_ID.

Consider what must happen when a user requests a form with existing data to be opened. Data must be fetched from the Form_Data, Group_Data and Clinical_Data tables, so that each “Value” of the Clinical Data table is placed in the appropriate element of the form.

- The Form ID and Form_Data_ID are the same for all the editable elements of Fig. 5.3. Therefore the database can fetch all Group_Data elements with a given Form_Data_ID and Form ID.
- Similarly, after getting all the necessary Group_Data elements, it can fetch all the Clinical_Data rows whose Group_Data_ID value belongs to the set fetched in Step 1.
- It is now a matter of placing each “value” in the corresponding element of the form. To identify an element uniquely, its ID needs to contain only the combination of Group ID, Parameter and Instance.
- In this form, the user is allowed to add new rows by clicking the “Add Record” button. The runtime software would simply generate a new instance number, increment it by 1 each time. The exact instance number is not meaningful – there may be gaps in the sequence if a row is deleted. As long as they are in ascending order and uniquely assigned, the system knows how to arrange the data that is fetched, and data from different instance rows is not scrambled.
- After fetching data the first time from the database, we store the Form_Data_ID with the form, and the Group_Data_ID with each row, since both of these will be used for data edits, to determine which row in the Clinical Data table to access. Certain values, such as the Start Date and Stop Date in the Prior Surgery group, are not parameters at all: they actually correspond to the Start_Event and End_Event datetimes in the Group_Data table. To update these rows, the Group_Data_ID is necessary.
- Fields such as the time-stamps and user identification associated with Form_Data are updated by the system software.

For the Web pages that our own framework generates, the *name* property of a field is a combination of the Group ID, Parameter ID, Data Type (a single letter) and an Instance number: all of these are separated by underscores. Strictly speaking, the data type is dependent on the Parameter ID, but its use in the combination saves an extra lookup step. Data type information is necessary to generate the appropriate SQL: values that are strings have to be enclosed in quotes, while date-time values have to be escaped or referenced using a function that converts a string to a date-time. The Form ID, which would be shared by all

elements, is not part of the element name: it is stored as a variable in the form's script code, along with the `Form_Data_ID`.

The technique of assigning an ID (or some other programmer-accessible property) to a form element that corresponds directly or indirectly to the location of an element in the database schema where its data will be stored is called *self-mapping*. Many form-generation frameworks for standard columnar relational data (e.g., Microsoft Access or Visual Web Developer) also use self-mapping: in Access, the element is associated with a table name and column name.

5.7.3

Converting the User's Edits into SQL Operations

When a form is opened, we create a hash table that stores the contents of each field on the form (the key of the hash table is the field name, the value is the value of the contents, represented as a string). The `OnChange` event for each field on the form calls a library function that does several things:

1. It stores the changed value of the field in a hash table, also keyed on the field name.
2. Using dependency information for this field, it performs the actions of skip logic and computations of other fields' values as necessary. If this field controls a conditional choice set, it alters the contents of the latter. If other fields get altered as a result, the `OnChange` event of each of the altered fields is fired, in chain-reaction fashion.

In terms of changes to the database, the first action is important. When the user tries to save the form, the list of changed values is compared with the corresponding original values. (Note that a field may have been changed, and changed back to the original: so a comparison is necessary.) There are three possibilities:

1. If neither the old nor the new values are blank, then a possible change was made. If the old and new values are actually different, an `UPDATE` statement needs to be issued for the corresponding EAV table.
2. If the old value was blank and the new value is not, an `INSERT` statement needs to be issued against the corresponding EAV table.
3. If a non-blank value has been erased, a `DELETE` statement needs to be issued.
4. An additional complication involves addition and deletion of rows in the case of repeating groups, e.g. in Fig. 5.3. For addition of rows, a `Group_Data_ID` for that row must be generated through an `INSERT`, and the value fetched. This value is now used for the `INSERTs` of individual parameter values in EAV tables. If a row that was originally populated with data is removed (this action requires a confirmation from the user), a `DELETE` statement is issued against the `Group_Data` table. A cascading delete relationship between the `Group_Data` table and the individual EAV tables causes the parameter values corresponding to this row to be deleted.

5.8

Choice of Technology for the Forms Runtime Environment

This section is likely to age very rapidly, given the speed at which technology is progressing, but I'll provide overview about my own experiences that you should hopefully find useful.

Many EMR vendors (notably Epic Systems) still shun Web technology for clinical data forms. Web technology is certainly much trickier to work with, but far more scalable, and deployment is, of course, much simpler: special software on individual users' machines is rarely needed. Features like skip logic with section-hiding and optional fields are much easier to implement using a technology like HTML, where a page automatically reformats so that there are no unsightly gaps. In my opinion, with the increasing maturity of Web-development platforms, there is currently no excuse for not moving to Web deployment for end-user access to data. (I've previously stated my preference for non-Web technology for metadata editing.) Therefore in the rest of this section, I'll limit myself to Web-related issues only.

5.8.1

Browser Code Development

Since a form is essentially a unit of code, the nature of the technology impacts deployment. The code in a form is essentially of two types. In order to achieve high responsiveness to the user, a lot of the action – such as simple validation, skips and several types of computation – takes place within the Web browser. All Web browsers run at least one interpreted language: JavaScript, and in the case of Internet Explorer, VBScript as well. Few serious developers would consider themselves fans of web-browser scripting languages in their current incarnation, given the numerous issues that I've emphasized earlier: they are at best a necessary evil that one has to live with.

The limitations of scripting languages become especially significant when one starts to develop thousands of lines of code, as software frameworks require. Until recently, however, you were stuck with them. Today, Google, as I've mentioned earlier, tries to save you the trouble of writing JavaScript altogether by letting you work with Java instead. In my experience, however, their toolkit, despite the recent introduction of a high-level XML dialect, still doesn't offer the productivity of environments like Microsoft Visual Studio. However, some third-party outfits offer GUIs that generate Google-toolkit code. If you are JavaScript-phobic (and you like Java) you might want to check out GWT Designer by Instantiations, which is quite affordable.

5.8.2

Web Server Code: Compiled Versus Interpreted Environments

One issue that affects the choice of technology is form volatility. Our main installation of our CRIS employs a total of some 5,000 forms: we have a support staff that is supporting

diverse groups of users, and forms are often revised – often several times a day, especially during the prototyping phase when designers are refining them iteratively in close collaboration with lead users. A fully interpreted environment, such as PHP or Microsoft’s Active Server Pages (ASP), allows a form to be deployed without impacting the rest of the system.

Certain other technologies, notably Microsoft ASP.NET, allow you to compile the entire Website and deploy the compiled version only. This greatly improves website performance, but such an approach is less suitable for a Website whose contents change more often than daily, because running the compilation is a developer’s responsibility. (Incidentally, ASP.NET has nothing in common with the older ASP technology other than the three letters of its name.)

ASP.NET offers another option, which is to deploy new Web pages, or new versions of Web pages, as they are developed. The first time someone accesses the updated Web site, the entire Website is recompiled. Again, the delay that the first user would experience when a site consisting of more than 5,000 pages is recompiled on demand is hardly desirable. If, however, your site is not especially volatile, then this approach would be still acceptable, because a nightly compile can forestall the “first-user” delay.

Java follows an intermediate approach: individual Web pages (Java Server Pages) are recompiled. But the compilation is to intermediate code that is then interpreted by a runtime environment, rather than being compiled all the way to binary, as .Net does. For websites that are not particularly volatile, a .NET website will perform faster than the equivalent Java-based website. But for large websites that are also highly volatile, Java’s approach of not trying to compile the entire Website all the way may actually offer the best compromise between performance and ease of deployment.

We have actually used a hybrid approach. The stable part of the Website – the data-access library code for example – is built with ASP.NET: the volatile part – the individual forms – are built with legacy ASP, but these forms are themselves placed within a “frame” that is an ASP.NET page, and contains standard code (e.g., for database updates). The two communicate with each other through the Web browser. We speed performance of the ASP code by making it perform its database access through a compiled library (also written in .NET).

The rationale for our approach at the time was that the Microsoft development tools (in particular, the edit-and-run feature, which lets you fix most logical errors in your code while it is still running) were far better than anything else available. ASP.Net is also a very high-productivity environment, doing what it does quite well. However, if you’re committed to Java, the effort required for framework development shouldn’t be dramatically greater than our own. Eclipse has certainly largely caught up – current versions of Java support edit-and-run, called “hot code replace”, and competition from Eclipse has been a major factor in keeping Microsoft honest and responsive to its developer base.

5.8.3

Newer Technologies

One question that several developers have asked us is whether we would eventually leverage XForms technology (see www.w3.org/xforms). Here, one would have to generate, not web pages that contained mostly straight HTML with embedded code (i.e., ASP pages) but

a Web page with embedded XML markup that addresses many user-interface matters directly. (Microsoft itself uses a dialect of XForms in its InfoPath product, part of the Office suite.)

While XForms is definitely better than raw HTML, and allows the developer to specify simple validation (such as range and regular expression checks) easily, currently it doesn't address the harder issues, especially where database access is involved. If, however, XForms matures to the extent that all major browsers begin to support it, it would be foolish not to seize the opportunity to generate more concise and expressive code: it is already widely accepted that HTML's native forms model, apart from being long in the tooth, is primitive and badly in need of a major overhaul.

Hopefully, shortly after this book's publication, HTML version 5 will become a mainstream technology. HTML5 uses an improved Web form model (which subsumes most of XForms's capabilities – further work on XForms has now ceased), though at the moment we don't have much use for its multimedia and image-drawing features in the context of clinical data entry.

In summary, this is one sub-section of the book that I hope gets obsolete fast. Web development environments still don't offer the same level of productivity that most of the tools based on older client-server technology (MS-Access, Visual Basic, even the venerable PowerBuilder) offer. The issues of robust validation appear to have been an after-thought, and to the extent that mainstream software tools let you specify them declaratively (instead of having to first build a framework to do so), we will all lead more fulfilled lives.

5.9

Flowsheets: Allowing Users to Create Their Own and Forms in EMRs

The form-design capabilities of clinical study data management systems are generally much more evolved than in EMRs. This is mainly because the parameters that are recorded in clinical research are, almost by definition, much more in flux, and often fairly esoteric and leading-edge: some of these, such as specialized laboratory tests, may not even have entered the textbooks. By contrast, the majority of parameters and forms in an EMR that has been employed in one institution can be mostly moved to another institution without change.

Nonetheless, the ability to designate new parameters and create forms based on those parameters is still necessary, and EMRs provide this capability in many ways. In the EPIC EMR, for example, there is a subschema related to what the vendor calls "flowsheets", which are customer-designed forms containing sets of parameters that need to be captured repeatedly. The process of flowsheet definition is straightforward and dialog-driven: after creating a new parameter, you assign it a name, data type, range checks and enumerated values. The software allows sequences of values of one or more parameters in a flowsheet to be graphed against time: this is very useful if you want to see how the patient (for example, a diabetic) is doing with therapy.

Flowsheet creation is so easy and rapid – with a turnaround time of less than a day – that their use is ubiquitous across individual clinical departments. Unfortunately, the tools that

let you curate the existing definitions are less advanced than the definition-creation tools, and this is the source of a trap many years down the road, when the organization decides that it would be worth creating a data warehouse. At this point, unless the institution has designated someone as a curator who approves addition of new elements— and one who takes her/his job seriously, at that — inspection of the flowsheet metadata leads to discovery of thousands of definitions, many of them redundantly defining the same parameter, possibly with different names. (One of the more spectacular examples in my experience involved a parameter named “blood pressure” that was defined *as a string*, e.g., “128/78” that combined both systolic and diastolic pressure, which was redundant with parameters that recorded systolic and diastolic pressure separately.) Cleanup of the metadata — and the data that has been recorded using redundant parameters — now becomes a major undertaking.

This issue — metadata that deteriorates through neglect — is a recurring theme that I’ll discuss in Chapter 14.

5.10 Open-Source EMRs

Since an important subset of this book’s audience is software developers who need to get ideas for developing their own (relatively lightweight) EMRs that could cater to small practices, I’m going to point to some systems whose internals can be studied.

5.10.1 VAMC VistA

Long before the term “open-source” was well known, the Veterans Administration Medical Center’s DHCP software⁶ (now renamed to VistA) was freely available, having been created by employees of the US Government, to any US citizen.

One warning: MUMPS code makes the 1978 Applesoft BASIC (designed by Steve Wozniak for the Apple II) look like Java by comparison. Rather than try to study the source (which is being rewritten for a modern platform anyway), I’d advise you to study the documentation from both the user and developer perspective and then figure out on your own what data structures you’d need for the documented functionality.

A second warning: this documentation is vast. Further, on the Vista website,⁷ when you try to access the manuals, you find that they are listed alphabetically rather than from elementary to advanced concepts. Start with the Infrastructure section first, and within the Infrastructure section, start with FileMan,⁸ which lets you define tables. (Fileman makes it so easy to create new tables, despite its VT-100 terminal-oriented interface, that at one VAMC, the administrative staff had used it to create a table that recorded parking tickets issued by VA police.) In studying the documentation, you should note that FileMan implements hierarchical databases, as stated in the previous chapter, not truly relational ones — even though you can operate on them, in a fashion, through a SQL interface.

5.10.2

OpenMRS

The design of OpenMRS⁹ was influenced by the Regenstrief Medical Record, developed in Indianapolis in the pre-RDBMS era – the original developers of OpenMRS hailed from this institution. OpenMRS, however, will utilize an RDBMS such as MySQL, and has been developed using Java and Java-based technologies such as the well-known object-relational mapping software Hibernate.

OpenMRS was developed specifically with the idea of supporting hospitals in third-world countries, which rarely have the multi-million-dollar IT budget that lets them purchase a commercial EMR: a recent JAMIA paper describes an implementation effort in Uganda.¹⁰ OpenMRS is a community effort, so if you take from it, you should be a good citizen and give back – through feedback, if nothing else.

You should be aware that the OpenMRS developer documentation lags considerably behind the user documentation: for example, the data model is provided as an E-R diagram, but the schema is not annotated through descriptions of the individual tables and columns. Certain topics on the Documentation sub-site, such as the “Developer How-to Guide” are currently only placeholders. Hopefully, this situation will improve soon.

References

1. Wikipedia. SOAP Note. 2010 [cited 3/2/2010]; Available from: http://en.wikipedia.org/wiki/SOAP_note.
2. Nadkarni P, et al. TrialDB: a clinical study data management system. 2010 [cited 11/1/10]; Available from: <http://trialdb.med.yale.edu>.
3. Nadkarni PM et al. Managing attribute-value clinical trials data using the ACT/DB client-server database system. *J Am Med Inform Assoc*. 1998;5(2):139-151.
4. Crane D, Pascarello E, James D. *AJAX in Action*. Greenwich: Manning Publications; 2005.
5. www.json.org. Introducing JSON. 2010 [cited 11/2/10]; Available from: <http://www.json.org/>.
6. Department of Veterans Affairs. *Decentralized Hospital Computer System Version 2.1: Programmer's Manual*. San Francisco: Information Systems Center; 1994.
7. Department of Veterans Affairs. VistA documentation. 2010 [cited 10/1/10]; Available from: <http://worldvista.org/Documentation>.
8. Department of Veterans Affairs. *VA Fileman User Manual Version 21.0*. San Francisco: Information Systems Center; 1994.
9. www.openmrs.org. OpenMRS. 2010 [cited 11/1/10]; Available from: <http://openmrs.org/>.
10. Were MC et al. Evaluating a scalable model for implementing electronic health records in resource-limited settings. *J Am Med Inform Assoc*. 2010;17(3):237-244.

Mere capture of data is useful in an EMR, but being able to support clinical and administrative decision-making through means other than the presentation of stored data makes the overall system much more useful.

“To Err is Human”, the famous 1999 report of the Institute of Medicine (IOM),¹ claimed that between 44,000 and 98,000 patients die in hospitals each year as a result of medical errors, a significant proportion of them preventable. This attention-grabbing number was based on an extrapolation of data from three states over different years (New York, 1984 and Colorado and Utah, 1992), so that the true lethal-error rate today may be either much more or much less than this. In any case, the safety record of the US healthcare system is relatively dismal: if the aviation industry had a similar record, the bulk of the industry’s operating budget would be consumed in reimbursing the surviving relatives of deceased passengers, and few of us would ever muster the courage to board an airplane.

The IOM report achieved its goal of shaking up the medical establishment’s faith in its own omniscience. Electronic decision support, much of it aimed at the prevention of errors, is consequently an increasingly important function of EMRs: the use of protocols for data capture, for example, eliminates the variation in the comprehensiveness and accuracy of data capture that would occur between novice and expert caregivers.

This chapter considers another aspect of protocols: software that supports decision-making by one of several mechanisms. These may involve a single patient at the point of care, or involve large sets of patients. Mechanisms to support point-of-care decision-making (*tactical* or *short-term* decision support) include:

- Directing the caregiver’s attention to a situation related to an individual patient that needs prompt attention. Related to this function is the warning against, or prevention of, inappropriate actions when one or more pending actions of a caregiver related to a patient are detected. This mechanism is called *alerting*.
- Facilitating *access to information* related to a patient’s condition. Ideally, the information that is retrieved is focused by considering the patient’s specific findings. The retrieved information allows the caregiver to determine an optimal course of action.
- Storing, computing and presenting information about the optimal course of action that considers the patient’s condition as well as specific clinical presentation. The mechanism here is *electronic guidelines*.
- Facilitating the optimal course of action through *automation* where possible: such automation is coupled both to guidelines as well as alerts. Automation related to workflow

is employed in many cases where the action needed is not urgent, but must be performed within a reasonable timeline.

Mechanisms to support decision-making for *sets of patients* (*strategic* decision support) are important for matters related to research or public health. This is one area where current EMRs provide little or no support other than allowing the export of their data (either all or selected subsets) to an external system where it can be queried, analyzed or “mined”. This is not necessarily a bad thing: it is better for a system to focus on what it does well rather than trying to do everything indifferently.

Data mining and analysis are specialized areas that I will consider very briefly. The difference between data-mining and traditional analysis is that traditional analysis encompasses techniques that are intended to *confirm* (or disprove) hypotheses. Data-mining techniques, by contrast, are intended to look for patterns in the data and *suggest* hypotheses. The suggestions must then be verified by working with different (and typically larger) data sets, which are often collected prospectively for the express purpose of hypothesis verification. If verified successfully, the results of mining or analytical efforts may be fed back into short-term decision-making processes. (A good introduction to data mining is found in²: the authors have also created the open-source Weka software,³ which is a good starting point for experimenting with data-mining algorithms.)

The field of clinical decision support is quite vast: Greenes⁴ deals specifically with this theme. In this book, I’ll focus on the software issues. The systems that I will consider are mostly *open-loop* systems: that is, they provide inputs to the caregiver, but rely on the caregiver to take the requisite action. Sometimes, however, it is more appropriate to employ a *closed-loop* mechanism, that is: the software both detects and responds to the situation. The most well-known examples of closed-loop systems are medical devices that monitor a physiological parameter, e.g., heart rate, blood glucose, and respond by activating a pacemaker or controlling the rate of insulin infusion respectively. Asynchronous (batch) processes, such as identifying patients due for follow-up for specified medical conditions through their visit history and visit indications in the medical record and placing them in a scheduling queue, are also examples of closed-loop systems. I will not discuss the highly specialized area of real-time decision-making/decision-support software embedded within medical devices in this book, but I will consider batch processes.

Let us now explore alerts in detail.

6.1

Introduction to Alerts

An *Alert* is an electronic mechanism that warns the provider about a situation related to the current patient whose data is being processed that requires her/his attention. Alerts commonly arise during the operation of *computerized physician order entry* (CPOE) systems. They can relate to almost any factor related to the patient.

- *Drug-drug Interaction Alerts*: The vast majority of alerts relate to potential interactions between the medications that are being ordered for the patient. The existing set of

known drug-drug-interaction pairs is already in the tens of thousands, and grows steadily, with more being discovered every month. Given that even experts in the field cannot remember them all, electronic assistance is highly desirable. All interactions are not of equal concern, however: many of them are predictable from a drug's pharmacological profile, and some interactions may even be synergistic or desirable in a given clinical condition if the dosage of the drugs involved is appropriate.

- *Other drug-related interactions* can occur between the medications and the patient's clinical conditions: thus, certain medications aggravate particular conditions and are contra-indicated. There may also be interactions between medications and laboratory tests: some medications may interfere with the test used to measure a substance, while other medications will influence the level of the substance through their physiological effects.
- *Allergy alerts* are based on a check of the current medication against a list of the patient's known allergies. Especially in intensive-care situations, overlooking of documented allergies is a common and readily preventable cause of medical error. Further, a patient allergic to one member of a chemical family of drugs (e.g., the penicillins) is likely to be allergic to other members as well.

The “garbage-in, garbage-out” factor applies here. Sometimes patients report an “allergy” to a particular drug without specifying how they have reacted to it, and on more detailed inquiry, may reveal only that they are less tolerant to specific side effects, such as drowsiness: certain software applications do not recognize this distinction.⁵ This distinction is important because in many conditions (e.g., critical care in hospitalized patients), the risk of particular side-effect may be considered acceptable compared to their benefits.

- Alerts may fire when an *out-of-range value* returned by a laboratory test measurement indicates the need for expeditious clinical intervention. Examples are very high serum potassium in patients receiving certain medications for high blood pressure, or high blood glucose in a previously well-controlled diabetic.
- *Reminders* are alerts related to a prompt for a recommended or desirable action: e.g., a diabetic patient may be due to be scheduled for an eye exam or a Glycosylated Hemoglobin measurement. Such alerts are driven by a check of the patient's visit history and the conditions that the patient is known to have.

Kuperman et al.⁶ provide a good review of decision support (including alerting issues) for CPOEs.

6.2 Implementation of Alerts

Certain alerts (such as prompts for recommended actions, or range check alerts) work best in *batch mode*. Here, a given alert operates against a large set of patients in the EMR. Patients with a given condition who are due for screening can be identified by a process that runs nightly, while out-of-range alerts can run after a laboratory-analytical batch process is completed. By contrast, alerts for interactions and allergies must be activated during

an interactive operation related to a single patient, such as processing of an order for a set of medications.

There are various strategies for implementing interactive alerts; all of these are based on storing the logic of the alert in structured form and/or the information it needs as machine-interpretable metadata.

- When the variables on which an alert depends are all within the current electronic form, the alert can use the same mechanism as complex validation, which we discussed in the User Interface chapter. However, there are some significant differences.
 - An alert does not indicate that the data is invalid, but only recommends that some action is to be taken expeditiously, assuming that the values entered are accurate.
 - Compared to validation, alerts use *inverted logic*. While a validation-error message appears if the validation condition is false, an Alert must fire if the alert condition is true.
 - Validation errors *must* be addressed by fixing the errors indicated, or the message/s will not go away: inconsistent data cannot be allowed to enter the database. By contrast, alerts may not be 100% accurate, a point we address below, and so the clinician may often be permitted to ignore certain alerts that are designated as non-critical.
- The majority of interactive alerts turn out to be much more complex, relying on information recorded previously, and stored either in *older instances* of the form (i.e., related to previous clinical encounters), or stored in other parts of the patient's electronic record (e.g., allergy information). Ideally, in cases where the alert consists of a textual message, the content of the message would have to be tailored appropriately based on historical data. For example, a high glycosylated hemoglobin (HbA1C) in a diabetic would be a cause for concern, but if the previous HbA1C was much higher than the current value, the patient is actually improving: the caregiver is doing her/his job, and does not need to be issued dire warnings.
- Use of electronic reference content: Alerts related to interactions utilize content provided by an EMR vendor or CPOE-software vendor. Such content is typically developed by third party content-vendor partners of the EMR/CPOE vendor. Here, the metadata has a relatively simple, regular structure, and fits naturally into an RDBMS design: the alerting routines simply run well-defined queries, or a set of standard operations, against it. I'll describe this in greater detail in a subsequent chapter when I consider table-driven approaches.

The content vendors provide periodic upgrades to the content based on new information distilled from the recent medical literature. Most content-vendors will also sell their reference database directly (on a subscription basis) to individual customers who intend to build custom applications based on the content. Here, the price tag is typically high enough – in the tens of thousands of dollars per year – that only reasonably large customers (or customers who can obtain a grant to foot the bill) are able to afford the subscription.

6.3 Accuracy of Alerts

Drug-drug interaction alerts are most logically implemented within CPOE software, or the CPOE subcomponent of an EMR. Much CPOE software is stand-alone. Many organizations that are not convinced about the benefits of, or who cannot currently afford, a full-fledged EMR, may still decide to invest in a much less expensive stand-alone CPOE system.

It is important to note that most CPOE systems generate their alerts based upon incomplete information. Drug-drug interactions detection mostly needs only two types of information: the current order, and a reference interactions database. Drug-disease interaction detection, however, requires additional information, namely, access to the patient's current diagnoses. This is where several challenges lie.

- *Software Integration Issues:* The CPOE software may be built by a different vendor than the one who provides the EMR, and integration may be incomplete. Incomplete integration often occurs because a large EMR vendor acquires a smaller CPOE-software vendor rather than developing their own: the vendor's marketing department starts to extol the benefits of their "complete" solution while the software-development team is still scrambling to implement full integration. Consequently, all (or most) of the information in the EMR may not be available to the CPOE, which then generates many nuisance alerts.
- *Incomplete Information:* Even a fully integrated and well-designed CPOE/EMR system is limited by what information is available on the patient. For a patient admitted to a hospital for the first time, the system will only generate alerts based on the medications present in the patient's orders. The patient may already be taking several other medications for a chronic condition that are not in the order list, because the patient is admitted for an unrelated, acute condition. The newly given drugs may interact with the old, but the CPOE system may know only about the former.⁷

Medication reconciliation is the task of creating and maintaining an accurate list of a patient's current medications, and comparing it to the current orders, so as prevent errors such as omissions, duplications, dosing errors or drug interactions. Ideally, such a list should be accessible to all involved in the care-giving process within an institution, and even beyond it. The latter capability is important in circumstances when the patient has relocated temporarily or permanently and is being managed by someone other than his usual/former providers, and is one of the factors behind the momentum toward a trans-institutional *personal health record* (PHR). PHR efforts have been led by organizations such as Google and Microsoft, which have a commercial model based on advertising (while free for actual patients): partnering with them are government agencies such as the Department of Veterans Affairs.

Medication reconciliation would seem to be a function tailor-made for electronic support. Astonishingly, however, few CPOE systems have built-in support for this process, which again requires inter-operation with the EMR proper. Achieving such integration often takes significant in-house software development, often in partnership with CPOE and EMR vendors, as reported by Anderson.⁸

- *Computational Sophistication Issues:* Even in cases where the CPOE and EMR are provided by the same vendor, the amount of information that the CPOE accesses from the rest of the EMR may be very limited. While some structured and encoded data – e.g., ICD diagnosis codes – may be accessed by the CPOE in an attempt to fine-tune its recommendations, much of the information that is relevant to the alert process may be in the narrative part of the clinical record. Such information is much harder to process: we are not aware of any CPOE that can generate alerts based on real-time processing of narrative text. While one could write a custom program to do this, the effort involved is considerable, and the false-positive and false-negative error rates may require much tuning effort to bring down to acceptable levels: a future chapter on terminologies will touch on some of these challenges.
- *Individual Variation:* Even for the problem of drug-drug interactions, knowledge of the patient is often necessary. For example, most CPOEs will red-flag any order that contains a combination of a calcium channel blocker that has effects on the heart (e.g., diltiazem or verapamil) with a beta-blocker, because this combination is known to slow down the heart rate significantly, and also has a risk of precipitating heart failure. The situation, however, is much more complicated in terms of clinical decision-making.
 - Both these side effects are dose-dependent – low doses of each in combination are reasonably safe, and such a combination actually be therapeutically superior to a maximal dose of a single drug.
 - To complicate matters still further, what constitutes a “low” or “moderate” dose depends largely on the patient. Following an oral dose, both these families of drugs are absorbed in the intestine and travel to the liver, where they are extensively inactivated by chemical transformation. The quantity of the drug that is spared by the liver acts on the rest of the body: this initial chemical transformation is called the “first pass” effect.

The extent of the first-pass effect varies significantly across individuals, due to genetic factors, not all of which have been fully characterized. Even within the same individual, it varies depending on whether the drug is taken with meals or on an empty stomach, and what food or drink has been consumed: citrus juices such as grapefruit (and to a lesser extent, orange) interact with both drug families. Consequently, the dose that is required to achieve a given therapeutic effect also varies markedly.

- *Incorporating Knowledge of the Individual Patient’s Previous Responses to Medication:* For the reasons stated above, it is very hard to *predict* how a given patient is going to respond to a given dose of a drug. However, knowledge of how a patient has *already* responded to specific medications in the past can potentially be utilized to make alerts more accurate and selective.

An example is allergies. A patient who is allergic to one member of a family of drugs might be expected to be allergic to any member of that family, or to related families. Many systems will automatically raise an alert in a patient with documented penicillin allergy who is prescribed a cephalosporin (which has a similar molecular structure). However, reality is more complex. The incidence of cross-sensitivity, which was previously believed to be around 10%, has been claimed to represent an overestimate, and

the “second-generation” cephalosporins differ significantly in chemical structure that cross-allergy does not seem to be an issue.⁹ In any case, if that patient has been treated with a cephalosporin before without allergic reactions, such an alert would only have nuisance value.

Utilizing previous medication history to improve alert accuracy is a challenging problem. To begin with, of course, the record on which the software makes its decisions must be of sufficiently long duration. Secondly, adverse events must be recognized accurately, recorded preferably in a standard encoded form (as opposed to highly variable narrative text) and an individual adverse event must be attributed accurately to the causative agent. The last is not easy to do if the drug is relatively new on the market and its full adverse-effect spectrum has yet to be discovered; when the patient is receiving multiple medications; or when the underlying disease condition may have contributed to the observed clinical manifestation.

Given that many order-entry systems are not fully integrated with the medical record, expecting the additional level of sophistication required to utilize data that may go back several years is possibly over-optimistic. But such a challenge is representative of the useful things that one can do, if all of a patient’s data across multiple health providers were available to an application.

- *Allowing for Clinician Judgment:* Most clinicians who prescribe a beta-blocker and calcium-channel blocker in combination know what they are doing, and will monitor for adverse effects, in part by telling the patient what symptoms to watch out for. Heart failure, for example, can present as breathlessness on exertion and swelling of the feet. Such clinicians would resent constantly being reminded of this drug interaction each time they refill the patient’s order. Such an alert would only be appropriate if the CPOE determined that the patient *already* had a very low heart rate or findings suggestive of heart failure. While heart-rate is usually captured in structured form, swelling of the feet or breathlessness are often recorded as clinical narrative that, as already stated, is challenging to process in real-time when one desires to identify thousands of possible interactions.

6.4

The Choice of Thresholds

Many alerts, such as those based on laboratory test values, are activated by thresholds – e.g., values above or below normal – where out-of-range values must be acted upon speedily. For example, if an patient’s Serum Potassium is greater than 5.5 mMol/L, then most laboratory systems are designed to send an alert to the physician (and the patient is also to be contacted immediately, if ambulatory) because this condition needs to be treated as soon as possible: a level above 6 mMol/L can be life-threatening.

In such situations, designing alerts rationally can be tricky: the threshold must be chosen carefully. In the case of the serum potassium example above, an alert that fired at the 6 mMol/L threshold might come too late to do the patient much good, since any clinical intervention aimed at lowering potassium takes some time to commence working. On the

other hand, an alert that fired if the potassium was just above normal might only serve to irritate the healthcare provider, who may already be aware of a clinical condition that would predispose to this (e.g., the patient is being treated with a drug that is known to elevate potassium).

Consequently, the issues of sensitivity and specificity must be considered. The appropriate threshold may vary in different environments, because of the different relative weighting that may be given to false positives (that result from too sensitive a threshold) versus false negatives (that result from too specific a threshold). The weighting, of course, would vary with the nature of the alert.

6.5

Workflow and Software-Integration Considerations

Based on the clinical importance of the alert, it may be necessary for the EMR to *log* the alert, as well as to log the caregiver's *acknowledgment* of the alert. Forcing the caregiver to acknowledge an alert is understandably an interruption of the latter's work process, and it must therefore be employed judiciously. In case a patient has an adverse outcome later due to failure to act on a specific alert, the log's contents can help to resolve issues of culpability.

Some alerts may be considered semi-critical, but if the software is designed well enough that it accepts its own limitations of incomplete information, it may allow the caregiver to *override* certain alerts. Here, the caregiver clicks an "override" button, and in some cases may be required to provide a brief explanation for the reason why the alert was overridden. Other alerts are so critical (e.g., alerts for certain combinations of drugs that are known to be lethal) that they cannot be overridden in any circumstances: the caregiver has no choice but to fix the condition that caused it. Such alerts resemble validation in their rigidity.

Ideally, the severity of an alert – minimal-level being non-interruptive, low-level requiring only acknowledgment, intermediate-level requiring an explanation to override, and high-level impossible to override – should depend on the condition for which the medication triggering the alert is being ordered. Thus, a warning about postural hypotension (significant drop in blood pressure when the patient rises from a lying position) may be intermediate-level if the patient is ambulatory, but may be ignored in a hospitalized, immobile patient in intensive-care being treated for a life-threatening condition. Again, adjustable alerts of this kind require not only complete integration with the EMR but also incorporation of significant medical knowledge into the alerting algorithms: I am not aware of such capabilities in current software.

When the alert is generated through batch processes such as a laboratory-test sample run, however, integration of the alerting mechanism with external software is desirable. For example, rather than relying on a human in the lab to contact the caregiver, automatic mechanisms such as paging with text messages or E-mails may improve the timeliness of response.

For *reminder*-based alerts, the action of simply nagging the provider has limited utility. Where possible, the software must facilitate workflow. That is, if the system is capable of

knowing that a particular intervention such as a physical checkup or vaccination is due, it should have the ability to actually place that intervention in a scheduling queue, so that the patient can be contacted by phone or E-mail for an appointment: in most cases, an intervention such as vaccination can be combined with the next scheduled appointment.

In such situations, whether the caregiver should be prompted for approval or not is an open question. Ideally, if the caregiver has accepted in advance that facilitation of scheduling would simplify her/his life, then notification for each and every patient would not be necessary, so that the term “alert” becomes somewhat of a misnomer: the software functions like an aide to which certain tasks can simply be delegated. In practice, because the scheduling software and the EMR may be provided by different vendors, such workflow facilitation may have to be programmed in-house. However, the process of building a framework to automate the process, while tedious, is not rocket-science.

6.6

Alerts and the Law of Unintended Consequences

Alerts are intended to reduce medical errors, so one would assume that the more alerts that are generated, the better. EMR-software vendors certainly seem to make this assumption. In case of harm to a patient due to a true-positive alert being ignored, the severity of the consequences to the caregiver’s institution depend on whether the caregiver is an employee of the institution or a consultant, but the EMR-software vendor would seem to be insured from product-liability suits, since the software generated the response it was supposed to.

However, such excessive caution may have effects *opposite* to those intended. Many CPOE systems, by default, are set up so that *every* potential interaction or adverse effect, however minor, is alerted. This configuration decision has its roots in the vendors’ desire to play it safe and escape legal liability, in the same manner that labels on children’s toys warn that the toys should not be chewed or swallowed – not that 2-year olds are capable of reading such labels – and labels on electrical appliances warn that they should not be operated in showers. Therefore, one of two things happens:

1. Alerts proliferate to the point where system usability is seriously compromised. Consider the following scenario:
Given a medication-order list consisting of N medications, there are $N*(N-1)/2$ pairs of possible interactions that are tested: there is a thus significant likelihood of the caregiver being alerted multiple times. As I have already stated, most interactions are minor and/or expected. Further, as also previously stated, a CPOE that generates an alert using incomplete information is likely to generate numerous false-positive alerts. Having to acknowledge each message in a blizzard of alerts (if only by hitting an “acknowledge” button) constitutes a significant distraction. Weingart et al.¹⁰ observed that 90% of allergy alerts are *correctly* overridden.

The additional seconds of interruption per order add up, over time and across all caregivers, to result in significant productivity losses. Clinicians now begin to complain that they spend more time wrestling with the computer than taking care of their patients,

and suspect that the software was designed to protect the organization from liability rather than with the caregivers' and patients' best interests in mind. At least one physician¹¹ argues that patient safety and physician efficiency must go hand-in-hand rather than sacrificing the latter in the interests of the former. We know of at least one hospital where the residents went on strike after being forced to use a CPOE system that was notorious in informatics circles for poor usability, and another where the staff physicians revolted en masse.¹²

If such a situation persists long enough, it has another undesired consequence. Humans have adapted to survive in a variety of hostile environments, both natural and man-made (e.g., Auschwitz). In their interactions with poorly designed electronic systems, they protect their sanity by tuning out irritations. If the software "cries wolf" too often, there is a real danger that healthcare providers may be unintentionally conditioned, like Pavlov's dogs, to reflexively ignore alerts that they have come to regard as nuisances. As in Aesop's fable, the occasional genuine, critical alert may be missed. The false-positive phenomenon accounts for the occasional studies in the medical literature that show that CPOE implementations result in either no improvement or a *worsening* of patient care in terms of medication-related adverse events.

2. Spectacular CPOE implementation failures have been so well publicized that most CPOE vendors now allow individual alerts to be customizable – i.e., turned on or off before deployment. However, in keeping with the vendor's desire to avoid product liability, the default configuration has *every alert* turned on. A thoughtful institution that does not want to drive its caregivers crazy must now have its local implementation team work very closely with individual departments to determine their preferences through a consensus-building approach.

Since the universe of alerts ranges in the thousands, this means that the implementation takes a long time after the software has been purchased before it goes live. This is not necessarily a bad thing – slow and deliberate implementation after external internal testing is preferable to the high-visibility flameouts that often follow a "big-bang" approach to deployment. However, it does take time and effort, and this effort must be replicated by *every* customer of the software vendor. Essentially, one is spending a lot of effort *undoing* the efforts of the vendor simply to make the software usable. Collaborations between institutions to borrow or share alert sets may reduce this workload, but this is only possible if agreements are in place to prevent Good Samaritans who offer their alert sets to others from getting sued by a borrower if a patient error due to alert omission occurs in the borrower's institution.

Phansalkar et al.¹³ provide a good review of human-factor considerations in the design of alerting systems. Many of these recommendations come from non-medical areas such as industrial engineering where alerts have been employed for a long time. Examples of recommendations include: visual alerts must be within the user's visual field and as close to the site of the problem as possible; they must use simple, direct statements in active voice; they must also suggest what corrective action to take. While these recommendations might seem self-obvious, software exists or has existed that violates at least one of them.⁶

As noted by Miller et al.,¹⁴ alerts must be raised as soon as possible – e.g., as soon as a drug is entered, so that corrective action is immediate. However, complementary

actions – e.g., ordering a test that monitors the drug effect, in cases where it has a narrow safety margin – are best suggested at the end of the order-entry session. Better still, complementary actions should be ordered automatically, if not already part of the current or standing orders for the patient, with a notification to the provider. The last suggestion follows from the requirement that workload be reduced by the system where possible.

6.7

The Lessons of History: The Microsoft Office Assistant

It may be worth drawing comparisons between poorly implemented alerts and the Assistant (“Clippit”) of earlier versions of Microsoft Office, which has been described (in *Smithsonian Magazine*) as “one of the worst software design blunders in the annals of computing”.¹⁵ A paper by Microsoft’s research team¹⁶ describes lessons learned from admittedly useful prototyping work that helped identify the requirements of the software prior to its implementation. Interestingly, this July 1998 paper, which must have been originally sent to the scientific conference where it was presented right after Office 97’s product launch, conveys the impression that the work was a success: so much for the premature announcement of triumphs.

It is worth contrasting the project’s objectives and the software’s requirements, which are indeed very reasonable, with what was actually implemented. Since the Microsoft Research team did not produce a follow-up paper (and is unlikely to do so in future, given the project’s ignominious end), this responsibility falls on me. Every one of the requirements listed below applies to clinical-alert scenarios.

- *The software must be unobtrusive, activating only when necessary.* The Assistant popped up inopportunely, suggesting help that the user never asked for. (“It appears that you are writing a letter. Would you like to use help?”)
- *The software’s intervention must be useful.* When the Office user requested help, the results were rarely what were desired.
- *The software’s behavior must be customizable.* The Assistant did not distinguish between a novice and a proficient user who had been using Office for several years. Unfortunately, there was no way for advanced users to customize the help behavior by creation of a user profile.
- *The software must adapt to the user’s observed expertise/competence level, and learn from information gathered during its use.* As for CPOEs and EMRs, extensive manual customization takes time, and it would be nice if the software could gradually customize itself.

The Assistant prototype used probabilistic (Bayesian) reasoning, which was intended to get progressively more accurate over time by consulting an action/response history maintained by the software. Maintaining such a history is not trivial, because the *context* of every operation must also be recorded. However, it is not frightfully difficult either: game-playing software, notably all the leading chess programs, can be configured by the serious player-user to employ such adaptive behavior. While not necessarily

employing Bayesian computations, such adaptation makes the programs progressively harder to beat: they gradually learn from the human opponent's playing style and will choose systems and/or styles of play deliberately designed to make this particular opponent most uncomfortable. (Such programs are ideal sparring partners for professional chess players looking to sharpen their skills and overcome their weaknesses.)

In the Office Assistant scenario, it would not have been difficult to maintain a frequency histogram of the usage of individual menu items. A Word user who uses regularly uses Styles or Outlining mode, or creates Tables of Contents, or an Excel user who uses pivot tables, almost certainly doesn't need help composing a letter or an elementary formula.

In the production version of the Assistant, the intended Bayesian approach was discarded in favor of a rule-based system *where the rules were excessively simple and never utilized usage-history data*. As a result, the software was no smarter after 6 months of use than it was after initial installation.

Interestingly, while the Assistant project's motivation was rooted in Artificial Intelligence research, the implementation team became a victim to the very common pitfall (by no means limited to AI) of underestimating the problem's complexity. The Assistant's implementation was so naïve as to exemplify only Artificial Imbecility.

Further, it seems that this naïveté was apparent even to some Microsoft developers. Within the Office implementation team, many were unconvinced of the project's direction and benefits. To quote Steve Sinofsky, currently President of Microsoft's Windows division,¹⁷ "...the Office Assistant was famously named TFC during development. The "C" stood for clown. I will let your active imagination figure out what the TF stood for."

6.8

Conclusions and Outstanding Issues: Customizability

The lessons for clinical alerts are clear. As for the Office Assistant, half-baked attempts to address a very complex problem may be worse than no attempt at all. Consensus-based customization is better than using the vendor's software as is, but it still represents a "one size fits all" approach. Even within a single department, all clinicians do not have the same expertise: an alert that a highly knowledgeable clinician considers needless may be genuinely useful to an intern in the first month of training. Further, even consensus-based customization decisions may need revision. Thus, after experience with the software (and possibly after some near-misses or medical errors that might have been prevented by an appropriate alert) users may decide that certain alerts should be turned on after all.

Most general-purpose software, such as the components of the Microsoft Office package, allows fairly detailed personalization. It is not very likely that a serious user of Office would accept a preference profile designed by someone else, yet users of CPOE software and EMRs are expected, for the most part, to accept an equivalent situation. Several CPOE vendors, in an attempt to improve their products' acceptability, now allow customization at the level of personalized order sets.

Order sets are lists of items, e.g., combinations of laboratory tests, medications, etc., that are ordered as single units. In the management of common conditions, their use saves significant clinician time, because an order does not need to be built one item at a time. In addition, standard order sets designed by a department within an institution ideally represent the best-accepted therapy for most patients: a clinician needs to vary, in terms of adding or removing items from the order, only in uncommon circumstances. In general, however, there is not always only one way of doing things correctly, so many clinicians may demand leeway in being able to define (or have defined for them) order sets unique to the way that they like to operate.

Customizability of alerts with respect to who sees which alert is currently limited in today's commercial software. Some systems allow *role-based* customization: thus certain alerts (e.g., regarding pharmaceutical incompatibility – certain drugs should not be mixed in the same container) are seen only by pharmacists but not by clinicians. Customization of alerts at the individual level is more challenging. *A priori* customization, apart from taking considerable effort, is not guaranteed to work, because certain users may later change their minds, with repeated use, decide that certain alerts should be turned on (or off), in the same way that certain Office users decide that personalized menus are not such a good idea.

It is not too hard to store individual profiles of users in shared-database software such as EMRs: preference information can readily be stored against individual logins within the database. It is also not too hard to build software that gradually learns the user's preferences. Thus, when a non-critical alert appears for the first time, the software can offer a given user the option of turning off its future appearance. In an era where off-the-shelf voice recognition software can adapt to an individual user's accent and cadence of speech, the limited ability of EMR/CPOE software to customize itself, or be customized for, the individual user, should be blamed on vendor inertia rather than technological limits.

References

1. Committee on Quality of Health Care in America IoM. *To Err Is Human: Building a Safer Health System*. Washington, DC: National Academy Press; 1999.
2. Witten IH, Frank E, Hall MA. *Data Mining: Practical Machine Learning Tools and Techniques*. San Francisco: Morgan Kaufmann; 2011.
3. University of Waikato. Weka: data mining software in Java [cited 10/1/10]. Available from: <http://www.cs.waikato.ac.nz/ml/weka/>. 2010.
4. Ash JS et al. Organizational and cultural change considerations. In: Greenes RA, ed. *Clinical Decision Support: The Road Ahead*. New York: Elsevier/Academic; 2007.
5. Kuperman GJ, Gandhi TK, Bates DW. Effective drug-allergy checking: methodological and operational issues. *J Biomed Inform*. 2003;36(1–2):70–79.
6. Kuperman GJ, Bobb A, Payne TH, et al. Medication-related clinical decision support in computerized provider order entry systems: a review. *J Am Med Inform Assoc*. 2007; 14(1):29–40.

7. Bobb A, Gleason K, Husch M, Feinglass J, Yarnold PR, Noskin GA. The epidemiology of prescribing errors: the potential impact of computerized prescriber order entry. *Arch Intern Med.* 2004;164(7):785-792.
8. Anderson HJ. Medication reconciliation: is there a better way? *Health Data Management Magazine*, January 1, 2010.
9. Pichichero ME. Cephalosporins can be prescribed safely for penicillin-allergic patients. *J Fam Pract.* 2006;55(2):106-112.
10. Weingart SN, Toth M, Sands DZ, Aronson MD, Davis RB, Phillips RS. Physicians' decisions to override computerized drug alerts in primary care. *Arch Intern Med.* 2003;163(21):2625-2631.
11. Howell D. Medication reconciliation, CPOE and patient safety: one physician's viewpoint [cited 10/2/10]. Available from: <http://www.executivehm.com/article/Medication-Reconciliation-CPOE-and-Patient-Safety-One-Physicians-Viewpoint/>, 2008.
12. Connelly C. Cedars-Sinai doctors cling to pen and paper. *Washington Post*, March 21, 2005.
13. Phansalkar S, Edworthy J, Hellier E, et al. A review of human factors principles for the design and implementation of medication safety alerts in clinical information systems. *J Am Med Inform Assoc.* 2010;17(5):493-501.
14. Miller RA, Waitman LR, Chen S, Rosenbloom ST. The anatomy of decision support during inpatient care provider order entry (CPOE): empirical observations from a decade of CPOE experience at Vanderbilt. *J Biomed Inform.* 2005;38(6):469-485.
15. Conniff R. What's behind a smile. *Smithsonian Magazine*; 2007:51-52.
16. Horvitz E, Breese J, Heckerman D, Hovel D, Rommelse K. *The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users*. In: Cooper GF, Serafin M, eds. *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, San Francisco, CA, 24-26 July 1998*, University of Wisconsin Business School, Madison; 1998.
17. Sinofsky S. Project management at Microsoft [cited 11/3/10]; Available from: <http://blogs.msdn.com/techtalk/archive/2005/12/16/504872.aspx>, 2005.

The previous chapter considered the various kinds of alerts. I now discuss how these can be implemented, and explore the issues related to standardization – more precisely, the lack of it. The presence of standards allows sharing of intellectual effort, so no one can be opposed to it in principle. However, while standardization has been enforced in many areas of biomedicine, standardization efforts in decision support have not been very successful to date. Let us first look at one of the root sources of the problem: namely, no one can mandate that all EMRs be built exactly alike.

7.1 Differences in EMR Designs

Every EMR is based on a database schema that is unique to that system's vendor, with the individual tables and columns having different names. This is of course expected, but even the same type of information (e.g., demographics, laboratory tests or antimicrobial sensitivity results) may be modeled differently in different systems. Further, while certain common principles, such as the use of Entity-Attribute-Value models, are used, the extent to which they are employed varies across systems. Even within the same schema, data that is conceptually similar in requirements (e.g., attribute sparseness) may be modeled differently within different sub-systems.

Legitimate reasons for such differences include different world-views between system architects, or the deliberate specialization of one system to solve a particular kind of problem. Bad reasons for differences include the absence of a unifying vision and project-management discipline. Large systems are built by teams of programmers with varying expertise, experience and domain knowledge, and unless the organization is compulsive about employing best practices such as design and code reviews to keep everyone on the same page, all parts of the schema design do not reflect the same degree of thoughtfulness and foresight.

- One problem, by no means limited to EMR schemas, is that there may be several tables that were originally designed for a particular purpose, but their use was abandoned either because the technology whose information they captured went out of fashion (this was the case for vectorcardiography), or because the table design was found to be non-adaptable to changing needs. The schemas of both Epic and VAMC's VistA, for example, are littered

with numerous “dead-end” tables whose use is deprecated by the vendor/maintainer, but which are still left around for backward- compatibility reasons: there may still exist some customer or site that made the mistake of using them when they were first deployed.

- Some technologies make it easier to make mistakes. Systems based on MUMPS, for example, are not limited by the 255-columns-per-table limit that was widely prevalent in most relational database engines until the mid 1990s, and so the temptation to add columns freely as a quick fix to a design problem becomes hard to resist, especially if the software framework automatically presents these new columns to the user without the need for extra programming effort. (Remember that, as we stated earlier, MUMPS “tables” are really key-value storage with no limit on the number of keys.) Thus, the antimicrobial-sensitivity-results table of VAMC VistA until recently had something like 400+ columns, each named for an antimicrobial chemical. When new antimicrobials came on to the market, more columns were added.

The problem is that a vast number of the columns now represent antimicrobials (such as sulfadiazine) that are not used any more because the bacteria seen in the USA (and most developed countries) became almost universally resistant to them more than a decade ago. But the columns still remain. VistA’s Patient Demographics table has also grown similarly: for example, there is one column for every war or engagement in which the USA was ever involved: these columns are intended to track whether a given veteran ever participated in a given engagement.

We’ll see later how the heterogeneous-schema issue has impacted standardization of decision support.

7.2

Sharing Medical Algorithms: Arden Syntax

The first effort at developing a standard for sharing medical algorithms, such as those related to alerts, was Arden Syntax, devised in 1990 as a collaborative effort between several leading medical informatics groups that had implemented alerts in production environments. Arden Syntax (so named after Columbia’s Arden conference facility in Manhattan, where the initial meeting was held) resembles a full fledged programming language, though with certain critical constraints. It includes support for certain operations that are common in the medical environment: its syntax was intended to be easy for persons with medical training to use.

Despite Arden’s becoming an American Society of Testing Materials (ASTM) standard in 1993 and later, an ANSI standard, there are only a few systems that implement it (Eclipsys, McKesson, Siemens). Notably, the two EMR market leaders, Epic and Cerner, do not do so, and it is worth considering the reasons for its limited penetration.

7.2.1

A Brief Arden Syntax Example

We introduce this issue by providing a very brief example of Arden syntax below. This example is reproduced from tutorial material developed by DongWen Wang.¹

In Arden, the building block of code is Medical Logic Module (MLM), a subroutine-like unit with accompanying “header” information that primarily serves the purpose of documentation (e.g., author, title, date created, deployment status, etc.). An alert is a type of MLM. (Note that Arden’s usage of “module” differs from modern programming languages, where a module contains *multiple* subroutines, possibly operating on shared data structures.)

The only part of the header that I have preserved in the MLM below is the “explanation” section. In my reproduction, I have taken the liberty of bolding reserved words, which will be referred to later.

explanation:

If physician orders CT scan with contrast, this rule retrieves most recent serum creatinine and blood urea nitrogen (BUN). If creatinine is more than 1.5, the system issues an alert to the physician to consider the possibility that his patient has renal failure, and to use other contrast dyes

data:

```
last_creat := read last f;
last_BUN := read last {"BUN level"};
;;
```

evoke:

```
ct_contrast_order;; // the “event” that triggers the alert
```

logic:

```
if
    last_creat is null and last_BUN is null
then
    alert_text := "No recent serum creatinine available. Consider patient's kidney
function before ordering contrast studies.";
    conclude true;
elseif
    last_creat > 1.5 or last_BUN > 30
then
    alert_text := "Consider impaired kidney function when ordering contrast studies
for this patient.";
    conclude true;
else
    conclude false;
endif;
;;
```

action:

```
write alert_text || "\nLast creatinine: " || last_creat || " on: " || time of last_creat || "\nLast
BUN: " || last_BUN || " on: " || time of last_BUN;
;;
```

7.2.2

Essential Features of Arden

1. Arden resembles programming languages such as Ada and Visual Basic in its use of *if..then..else...endif*, the *and/or* operators, etc., and the Pascal/Ada/C family in its use of semicolons as statement separators (though the use of doubled semi-colons to indicate the end of sections is not standard). Like many modern programming languages, it supports *nulls* to indicate an empty value (nothing returned).
2. An MLM is activated by the occurrence of a particular “event” – something that happens during the patient’s management (such as a CT scan order in the example above). The event is given a symbolic name, indicated in the *evoke* portion of the alert. In practice, there are almost as many events as there are MLMs in the system, though there is nothing to prevent a given event from activating more than one MLM.

3. *Read* and *write* effect input and output respectively. In an operational system, *Read* will fetch information from a database schema. *Read last* gets the most recent value of the parameter. Other aggregate operators that are alternatives to *last* are similar to those in SQL: *max*, *min*, *average*, *sum*, *count*, while *first* gets the oldest value. One can also place a temporal constraint following the *read* – e.g., *within 3 months*, to limit the range of values accessed.

How a particular parameter is fetched is implementation-specific – the implementer can put anything in the curly braces. While the code above uses a symbolic name, an actual example reproduced verbatim from the Columbia alert repository² which fetches the value of blood glucose, is:

- {‘dam’=“PDQRES2”, ‘constraints’ = “C****”;, ‘32308’}. The last number is the parameter ID for blood glucose in the Columbia EMR: the other elements help the interpreter get to the location in the database where the parameter is stored, in a way that is clear only to one intimately familiar with the details of the schema.
4. *Read* will not only return a value, but also the date/timestamp associated with it, which is accessed through the *time of* operator. (As discussed in previous chapters, this typically refers to the clinical event – here, the time of blood collection.)
 5. An alert uses *Conclude* as the equivalent of *return* in traditional programming languages. If true, the *Action* is executed; if false, nothing happens. While not shown here, curly braces are also used to isolate implementation-dependent behavior for alerts, such as the precise notification/alert mechanism (e.g., a pop-up message on a screen, or a pager activation), and the criticality level assigned to a given alert.
 6. An Arden MLM applies to a *current patient* for whom the event is triggered. Thus, “last” creatinine, etc. apply to this patient’s data.

7.2.3

Limitations of Arden Syntax

Using this example, I will now elucidate the reasons for the very limited momentum behind Arden Syntax.

1. Arden Syntax is, for practical purpose, a programming language, and therefore someone has to write a compiler or interpreter for it. However, that is not enough: an event mechanism to fire the alert is also necessary. How a given EMR actually implements an event mechanism, how an event actually activates a particular Arden MLM, and what exact information is passed by the event, is entirely up to the implementer.

Peter Ludemann’s 1994 paper,³ however, describes an implementation in a real institution where multiple systems from several vendors are operational, and “events” can take place in any one of these systems, which must pass data to a central “*event monitor*” or dispatcher. Here, it took three times the effort to build the event infrastructure as it did to implement an Arden Syntax compiler/interpreter per se (and the latter effort itself was hardly trivial).

2. There are no standards for how the contents inside the curly braces following the *read* statement are either specified or interpreted: this is up to the implementer of a given compiler. The same applies to notification/alerting mechanisms and alert criticality. This laissez-faire approach raises the following problems.
 - (a) Since different EMR vendors, as stated earlier, use different physical schema designs, and Arden itself has chosen not to enforce an abstraction layer for accessing individual parameters, it follows that a given interpreter or compiler will be specific to a particular EMR only, since the contents of the curly braces are *tightly coupled* to the implementation of the alert.
 - (b) Similarly, the source code of an alert will not be fully portable across EMRs: the parts within the curly braces must be rewritten for each EMR.
3. EMR database schemas are not static: the physical representation of a given parameter may change with newer versions of the system, and one version of a laboratory test may replace another. Schema changes have happened several times with the HELP system at LDS Hospital, Salt Lake City, Utah (one of the first institution-wide EMRs and a landmark in medical informatics). Also, over time, the assay for blood glucose has changed: earlier assays based on detection of reducing substances have now been replaced by enzymatic assays that use glucose oxidase.

With any of these changes, the contents of the curly-braces will have to be changed if a given alert is to continue to work. In a large system that may contain hundreds of alerts, a given clinical parameter is likely to be used in multiple alerts, and so every instance of access to this parameter will have to be changed.

To permit an implementer to sprinkle a given parameter's hard-coded access method within curly braces across a dozen or more locations in Arden code violates the principles of modularity and information hiding. Standard texts on software development, such as McConnell,⁴ warn programmers to use symbolic constants instead of hard-coded numbers in their code: the curly-braces approach, to put it politely, is very fragile. Note that while there is nothing to *prevent* an individual implementer from designing a loosely coupled approach, so that hard-coding within curly braces is avoided, it is more than likely that naïve implementers will emulate the Columbia example.

I would argue that curly braces are not needed at all. An alternative, service-based approach that we will look at in a future chapter separates the issue of database access from the logic of the MLM. The MLM, which can be treated as a black box, gets what it needs to as data that is passed in, and any changed data is passed back.

4. An Arden MLM behaves like an individual “rule” in a rule-based software system, to which I will provide a detailed introduction in a subsequent chapter. A given MLM is intended to operate mostly independently of other MLMs in the system, activating when a particular event occur (e.g., when the clinician orders a CT contrast for a particular patient), and its “universe” of data is implicitly limited to that single patient.

This model is a poor fit for batch scenarios, where one would like to identify and operate on a possibly large set of patients (in the common billing scenario, or to identify all individuals who are due for vaccination). As pointed out by Wright and Sittig,⁵ the “event” model, which is reasonable for real-time, *synchronous* alerts related to *individual* patients, involves unnecessary overhead for batch operations.

Batch-oriented algorithms are readily expressible in all modern programming environments, which support access to a database schema. Trying to achieve this within Arden's framework, which is oriented to patient-at-a-time processing, involves very inefficient workarounds. One must invoke the MLM consecutively on each patient matching the condition to fire the same action (e.g., "refer patient for vaccination"). By contrast, in a standard programming approach one would write one or more queries (e.g., that updated data) that handled the entire set in a single step.

Further, even for event-driven, patient-at-a-time situations, the rule-based approach is not always optimal. For drug interaction detection at the time of filling a prescription, for example, the number of known interactions already ranges in the tens of thousands. It is algorithmically simplest to simply take all the medications that the patient is already taking (plus any that have been additionally ordered), generate all possible pairs of medication combinations, and search an interactions table in a reference database for all entries containing one or more of these pairs, and then report associated information (nature of interaction, severity, bibliographic citations, etc.) on each pair that has been found. Trying to solve an interaction problem with an approach like

...IF the patient is taking Drug-X AND the patient is taking Drug-Y THEN....

are simply not computationally efficient: one would have to search through tens of thousands of such rules linearly. The reference databases implement a table-driven solution to the problem (which I'll describe in a subsequent chapter), instead of forcing one to create thousands of rules that are essentially variants of a single rule pattern.

In summary, there are several situations where the Arden approach toward implementing an alert can be needlessly constraining.

5. If someone is considering taking the trouble to create a language implementation that will work only on a single EMR, there must be significant benefits to doing so in proportion to the effort outlay. However, such a cost-benefits case is hard to make for Arden. In the example above, other than the use of the *last ()* and *time_of ()* functions, and the implementation-dependent *read ()* operation, there is nothing that could not be done in an ordinary programming language. Arden code is not significantly more readable or high-level than the equivalent code in Java/VB (augmented by a reasonable set of library functions) would have been.

In my opinion, the handful of extra functions in Arden that may simplify expression of medical algorithms (e.g., for date/time manipulation) are so readily implementable in any modern language that supports database access to time-stamped values, that the software-development effort involved in creation of a brand-new language for doctors is not really justifiable.

Note: The 1993 paper by Pryor and Hripcsak⁶ justifies the decision to create a new language because "... general-purpose programming languages ... lack medically relevant constructs and data structures such as database queries and time...General purpose query languages like SQL can retrieve data but do not contain the constructs needed to compare data in complex ways. For example, SQL lacks the block structure that would be provided by an IF-THEN statement, and it lacks constructs for time."

This argument would be mysterious to today's Java/.NET programmers, who freely mix SQL statements within a traditional language, or use object-relational-mapping frameworks such as Hibernate and Microsoft's LINQ to minimize SQL use. Today's high-end RDBMSs even allow functions and subroutines ("stored procedures") to be

written using modern programming languages, and these routines can be called from SQL, allowing the best of both worlds. However, even in 1993, this argument was questionable: the deliberate omission of procedural-programming constructs in SQL *mandated* SQL's embedding within a general-purpose programming language for application development.

In any case, the advantages of a “universal standard” language for expressing high-level algorithms, medical or otherwise, are chimerical. In the 1960s, in an attempt to bridge the divide between FORTRAN (used by scientists and engineers) and COBOL (used for business applications), IBM devised the all-encompassing language PL/1, which turned out to be so bloated that its usability suffered: even within IBM its use was slowly abandoned. The fact is that certain languages will always be better than others for specialized purposes: for example, PROLOG, while poorly suited to general-purpose programming, turns out to be very elegant for solving problems involving symbolic constraint satisfaction. Further, commercial cross-compiler technology is now robust and ubiquitous enough that if for some reason one wishes to translate high-level code from one language to another, it is rarely necessary to do it manually.

More important, the example Arden code above is trivial. When one has to address a hard software-design problem, the paradigm of doctors as programmers breaks down (as did the 1950s view of businessmen as potential programmers, which motivated the English-like structure of COBOL). *When you have to solve non-trivial problems, programming is hard no matter how you do it.* The choice of language may make a difference between hard and very hard (the latter would apply if you chose to program in assembler), but it is far less important than the knowledge of algorithms. Both doctors and businessmen who are not motivated to undertake a serious study of algorithms and data structures would be better off delegating the programming task to software-engineering professionals.

6. To express complex and sophisticated algorithms expeditiously, one requires a rich set of data structures and library routines, such as are bundled with modern languages. Arden's data structures and built-in routines are rudimentary: while the latest Arden specification supports the use of external library routines, I find it hard to perceive, for example, how an Arden developer could leverage most of the Java class libraries when the host language will not let you use complex data structures besides simple lists.

In addition, Arden lacks support for certain basic features that are essential for large-scale and/or team-based software projects, such as trapping undeclared-variable errors at code-design time; developer-defined parameterized subroutines and functions; and modular programming constructs such as packages and namespaces. This makes Arden's use for complex (“multi-step”) logic extremely challenging: one spends more effort working around the limitations of the framework than in solving the problem at hand. A paper by Sailors et al.⁷ from the University of Utah describes such workarounds as the creation of a single MLM whose only role was to access the database and get data for the other MLMs (to address the modularity concern of issue 2.b above), and using indicator values persisted to a database to achieve the equivalent of parameter passing to subroutines.

7. Finally, it is not enough merely to implement a compiler or interpreter that works correctly. Today's programmers have become used to high-productivity integrated code-writing and debugging environments: they are highly unlikely to put up with 1960s-style environments. The arguments listed previously against MUMPS in this regard apply to Arden as well.

It's not that stripped-down environments are completely rejected today – the languages Perl, Python and Ruby have bucked the trend – but the no-frills alternatives have been accepted only because, in addition to being genuinely useful in specific circumstances, most distributions are also free. Also, there are thriving developer communities with these languages, creating interesting libraries of components and frameworks that other developers can leverage. (Some developers have also created plug-ins for integrated development environments such as Eclipse or Visual Studio, so that Spartan programming environments are increasingly rare.)

8. The blessing of a particular technology with a seal of approval from ASTM, ANSI or ISO does not mean much. If the standard does too little (or too late), vendors will shy away from it. The list of dead-soon-after-arrival ANSI standards is fairly large. The Network data model, originally developed by an industry consortium, the Conference on Database Systems Languages (CODASYL), rapidly became obsolete when the relational database model proved its superiority. The procedural enhancements to SQL-99 (intended to convert it into a regular programming language, with loops, branches and subroutines) were universally ignored after IBM, Oracle and Microsoft simply decided to allow stored procedures to be written in languages that everyone already knew and used, such as Java (or C#/VB.NET, in Microsoft's case).

I should state that I have great respect for the intellectual honesty and rigor of the prime motivators behind Arden. They wrote a paper describing actual experience in porting Arden MLMs between EMRs,⁸ which quantified the severity and impact of the curly-braces problem: it takes courage to point out weaknesses in your own creation. I wonder, however, if greater awareness of the COBOL experience might have dissuaded them from implementing a language for doctors: Yogi Berra's memorable phrase, "d     vu all over again" comes to mind. COBOL's English-based design did not achieve its original goal of getting businessmen to write programs: on the contrary, its excessive verbosity only irritated professional programmers who needed to get their job done.

The positive aspect of the Arden effort is that the effort of building real implementations had the side effect of helping to identify several of the operations (such as *last*, *time of* and *within*) that must be supported. Anyone planning to implement a decision-support framework would do well to study the Arden specification carefully in order to identify the parts worth borrowing. And for all its limitations, the advocates of Arden can at least point to operational systems that manage real patients.

7.3

Complex Clinical Guidelines

Clinical guidelines⁹ are recommendations intended to assist healthcare providers in appropriate decision-making for a patient's clinical condition, based on the best and most current available evidence. A guideline can be as simple as "Patients over 50 should be screened for colon cancer with colonoscopy every 10 years". As we have already seen, this is not particularly difficult to implement in an EMR as a simple alert, or better still, as a

workflow that streamlines aspects such as patient scheduling. In this section, therefore, I will limit myself to *complex* guidelines, where there are alternative decisions that can be made for a patient, particularly when a standard approach has been attempted and has failed. (In theory, guidelines should also consider the possibility of balancing multiple competing objectives, such as affordability of treatment vs. optimal care, but as of today, few if any guidelines address such issues, partly because the evidence that would support such assessments is also relatively scanty.)

The Agency for Healthcare Research and Quality (AHRQ) maintains an online resource, the National Guideline Clearinghouse (NGC),¹⁰ that can be searched by keyword such as disease conditions or treatments. AHRQ points out, however, that the mere existence of a guideline on the NGC website does not imply AHRQ's endorsement: it is a case of *caveat emptor*. As long as a guideline was developed by anyone other than a single individual (e.g., a medical association, local or national, a professional society or an organization) and as long as the developer provides corroborative documentation that a systematic literature review was performed, and that the current version of the guideline is less than 5 years old, it is considered acceptable.

AHRQ stores summary information about the guideline (title, keywords, creator, funding sources, financial disclosures/conflicts of interest, date released, associated bibliography, current status, etc.), as well as a summary of the guideline's recommendations, with link to the full text of the original document, if available.

7.3.1

Nature of a Complex Guideline

The recommendations within a published complex guideline's narrative text typically follow the following pattern:

- The clinical condition where the guideline is applicable is carefully delimited. Thus, a given guideline may not concern itself with the management of all high-blood-pressure cases, but only those who have diabetes and mild renal failure.
- The recommendations are typically in prose form, since that is the easiest for medical professionals to understand. The prose is generally augmented by a summary in the form of bullet points, and sometimes by a diagram in the form of a flowchart. Many references in the literature use the term "decision tree", but "flowchart" is more precise because, unlike a tree, the diagram may have loops. (If only prose exists, someone who knows both medicine and has at least a modest knowledge of computing must convert the prose into a flowchart that can form the basis of an implementation. The challenges involved will be discussed shortly.)
- The flowchart has several types of nodes:
 - There is at least one desired *end-point* or *objective*. This is typically the arrival of a diagnosis, or the cure of the condition (or control, for chronic conditions that are not curable).
 - Certain intermediate nodes perform *computations* based on the input variables. The computations may be arbitrarily complex, and should be treated by the flowchart as

- a black-box subroutine rather than have their details expanded. An example of complex computation is the determination of the dose of aminoglycoside (e.g., streptomycin, tobramycin) antibiotics in the presence of renal failure.
- There are one or more decision (*branch*) nodes, based on the presence/absence or severity of clinical parameters.
 - Finally, there are *action* nodes, which are specific *recommendations* for diagnostic and/or therapeutic interventions.
- The individual recommendations are annotated with *evidence* in the form of bibliographic citations and their summaries. Ideally, each recommendation is also tagged with at least two quality measures: the *quality of the evidence* assessment, and the *classification of the recommendation*. Both measures are ordinal and anchored: that is, each point on the scale of measurement is clearly described.
 - The best evidence comes one or more randomized controlled trials, while the weakest evidence comes from opinions of respected authorities, expert committee reports or clinical experience based on isolated cases. (Authorities and expert committees have often been known to be wrong.)
 - In terms of recommendation classification it is important to note that certain recommendations may be *negative*: that is, a particular clinical action may be strongly *advised against*. Negative recommendations are hard (or impossible) to place in a flowchart, and don't commonly appear even in prose, but are sometimes included for legacy or other reasons. For example, certain treatments originally thought to be effective, or advocated by certain individuals, are found to be useless and/or the harm from or expense of the treatment outweighs the potential benefits. So the scale here goes from good evidence to conflicting evidence to insufficient evidence *for* an action, to good evidence to recommendation *against* a particular action.

7.3.2

Making Narrative Guidelines Actionable: Challenges

Narrative guidelines are intended to be human-interpretable, not computer-interpretable. While narrative text in electronic form is better than no guideline at all, its limitation is that clinicians who actually need help for a given patient who has a clinical condition and a specific clinical presentation, rarely have time to step away and search a database to see if there is a guideline that applies to that condition. A truly computerized guideline must provide focused help to the clinician at the exact time when it is needed.

There are numerous challenges in converting the prose (or even a flowchart) for a complex guideline into something electronically actionable.

- Ambiguities and under-specification: If a guideline exists only as prose, one must first attempt to convert it into flowchart form, as stated earlier. During this exercise, one may discover *ambiguities* or even *errors* in the branching-decision process that would be fatal to a computer program, or even render the guideline ineffective for an inexperienced physician. Specifically, the decision variables (and their specific values) that would result in one action versus another are often not clearly specified.

This ambiguity is seen, for example, in the guideline of the Society of Obstetricians & Gynecologists of Canada that concerns polycystic ovary syndrome (PCOS),¹¹ a condition of unknown origin in women that results in multiple cysts (tiny fluid-filled sacs) in the ovaries. In this condition, the normal release of an egg from the ovary in the middle of the menstrual cycle fails to occur, and so a cyst develops around each egg that is not released. The clinical findings, in addition to the cysts, include infrequent (or absent) menstruation and excessive production of male hormones. PCOS is an important cause of infertility, and the primary objective of treatment is to induce egg release (ovulation), which is a pre-requisite for pregnancy.

PCOS is not an easy condition to treat. Two last-resort measures to achieve pregnancy are the use of gonadotrophins (hormones that induce ovulation) and drilling of several tiny holes into each ovary with a hot needle or a laser beam. Each option has its risks. Hormone injections are expensive (because of the continuous monitoring required) and have the risk of inducing multiple pregnancies (e.g., twins, triplets, etc.). Drilling, while minimally invasive, is still a form of surgery and has the risks of any surgical procedure: further, the long-term risks of adhesions of the abdominal lining to the ovary have not been fully quantified. The guideline does not clearly specify the conditions where one is preferred to the other. In reality, many clinicians simply prefer the option that they have been trained to perform.

The problem of under-specification occurs when a recommended course of action (e.g., a medication) results in adverse effects. The decision as to whether to stop the medication depends on the nature of the effect, its severity in relation to the underlying condition, and how motivated a given patient is to continue therapy despite the effect. The guidelines typically leave the decision to the judgment of an individual clinician.

- Converting “common medical knowledge” into an electronic form can be difficult. For example, the PCOS guideline above states that ovary drilling is contra-indicated if the patient is a surgical risk. The definition of “surgical risk” is known intuitively by practically all clinicians. In terms of an electronic algorithm, however, identifying risk involves determining the values of a fairly large set of clinical variables. This determination constitutes one of the most widely implemented medical protocols, though possibly only the anesthesiologists who conduct a pre-operative evaluation will be able to enumerate all the clinical variables involved from memory: other clinicians mostly confine themselves to referring the patient for such an evaluation.

From an informatics perspective, the surgical-risk evaluation protocol is the equivalent of a subroutine that will be called in multiple situations. Most guidelines contain numerous references to such units of knowledge, and incorporating all of these in an operational system for decision support would take a lot of effort. If an implementer finds this degree of effort discouraging, one way to scale down the scope of the software is to make it largely “*open-loop*”. That is, the guideline makes recommendations electronically but lets the clinicians be responsible for interpreting them and actually carrying out the recommended actions. The existing (non-electronic) implementations of complex guidelines for specified conditions, as part of quality-improvement initiatives, are open-loop in nature.

- Implicit (unstated) assumptions have to be discovered by someone with medical knowledge, and have to be modeled explicitly. For example, the recommendations may

assume that the diagnosis of the condition being managed has been established unambiguously. For a patient in whom the diagnosis has not yet been made or is not clear-cut, steps must be added to the electronic version to incorporate the diagnostic workup.

- Converting the recommendation into actionable form requires the implementation of bypasses based on a given patient's current state. Every guideline must begin with the simplest case – a patient who has never been treated before for the condition. Such patients undergo a standard workup, and then one or more first-line treatments are considered based on the clinical profile. In the PCOS example above, many patients are significantly obese, and diet and exercise must be tried for such patients before attempting to use specific medications.

However, many patients will fail to respond to first-line therapy, and are following up with the physician. Therefore alternative therapies must be considered. In terms of the flowchart, one must introduce additional decision nodes that consider whether the patient is one where first-line therapy has failed. If so, then one bypasses the initial parts of the flowchart.

- Chronological Separation of Flowchart Nodes: Treatments never work instantaneously: even investigations take some time, so that there are always temporal delays between the nodes of a flowchart when it is applied to an actual patient. Thus, in cases where the first-line therapy has been recommended, one will not know whether it has worked until a while later. For example, diet/exercise for an obese PCOS patient must result in significant weight loss before it has even a chance of benefiting the condition.

In terms of an electronic implementation, one must be able to “suspend” the flowchart at a particular position and resume it later. In the area of business applications, the category of software called *workflow engines* has evolved to meet this need, and as I'll discuss in the next chapter, guideline representation research efforts have tried to support workflow.

- Achieving high patient specificity: Simply converting the prose of a guideline into an electronic document presented to the clinician is not enough: few time-constrained clinicians have the inclination to wade through a 10+ page document that describes management of the condition exhaustively. They ideally need highly focused and concise assistance related directly to the current patient's state.

For straightforward clinical presentations, this is relatively easy to achieve as part of workflow. For example, in CPOE systems, a standard order of a set of investigations/medications for a particular condition, in a sense, embodies a guideline that reflects the best current thinking about that condition. The standard order makes it easy for a clinician to do the right thing and difficult to depart from the guideline. That is, it is ergonomically easier to pick the first order presented on the screen than to remove some medications and substitute others, which takes extra effort. There is significant evidence that such focused help (which also includes simple recommendations such as vaccination or colonoscopy alerts) helps to improve the quality of patient care: see the references of Damiani et al.¹²

Difficulties arise when the standard recipe hasn't worked, or the patient's initial presentation is atypical, e.g., the patient has been seen elsewhere and is now being referred, and the physician is looking for guidance as to what to do. In terms of the flowchart representation discussed above, the software must be able to place the patient directly (and correctly) on one of the lower-level nodes of the flowchart, based upon information already known and available on the patient. This leads to the next challenge.

- Complete integration of the guideline-software framework with the EMR. At the very least, the data elements (e.g., decision variables) required by the guidelines that are being computerized must be in structured form. In theory, one could process narrative text to identify certain elements: however, the hit-and-miss nature of natural language processing, and its relatively slow performance, makes it challenging to use for real-time clinical support. We consider the difficulties in achieving such independence in the next sub-section.

I'll conclude this section by stating that the definition of a "computable" guideline has been somewhat elastic, varying with the scope of work of individual researchers. As of this writing, a software framework that supports a collection of complex guidelines, and which is actually being used in a production EMR to provide electronic decision support for real patients, is still far from being a reality.

Some work has focused on more limited objectives. For example, the GEM (Guideline Element Model) work of Shiffman et al.¹³ has focused on making the representation of the *non-algorithmic* portion of the guideline (that is, its summary and descriptive information) as structured as possible. Such maximal structuring facilitates indexing of the guideline, and thereby provides the basis of efficient search of a guideline database: however, the information that GEM helps organize is not electronically executable, in the context of an EMR or otherwise. The Vaidurya guideline search engine¹⁴ is intended to facilitate retrieval of guidelines that would match a given clinical situation: it indexes a guideline based on a combination of clinical/diagnostic parameters, treatments, body systems, guideline types (e.g., treatment vs. prevention) and clinical specialty.

Such approaches are useful in open-loop systems, but evaluations in situations involving real-time decision-support have not been done. One self-evaluation by Turner et al.¹⁵ of a guideline search engine (PRESGUID) developed by the investigators, which utilized content structured in the GEM format, evaluated performance of two groups of physicians on an open-book test, which used a set of cases dealing with a clinical condition. One group had access to the printed guidelines (on sheets of paper), while the other had access to the electronic version of the guideline. Unsurprisingly, performance with the electronic approach was significantly better. Unfortunately, the experimental design tells us nothing about the merits of the search engine or indexing approach: it only reconfirms the well-known observation that almost any electronically searchable text is better than plain paper. No doubt similar results could be obtained by comparing the printed Encyclopedia Britannica to its electronic counterpart.

7.4

Achieving Schema Independence for Decision-Support Software

Any software framework to support alerting or complex guidelines needs to be able to read data from the host EMR, and sometimes needs to update data as well. Robust, high-quality software frameworks take considerable effort to develop, and such effort can only be amortized if it can be employed across multiple EMRs. For complex guidelines, there are enough informatics-related challenges that the implementer would prefer not to have to

additionally address the formidable challenges of how an individual data element is to be accessed within an individual database schema.

The experience with the “curly-braces” problem of Arden has convinced medical informaticians that the data-access process should be as transparent (i.e., invisible) to the decision-support implementer as possible. That is, it must be independent of the database schema of the EMR that hosts it (other than mandating that the decision variables that it needs exist in structured form).

Achieving schema independence has been achieved in other knowledge domains that employ RDBMSs (e.g., accounting applications) through a combination of two mechanisms:

- Definition of a set of database “views”, which intervene between the guideline framework and the physical schema. That is, the framework developer cannot access the underlying tables directly, but only indirectly through queries on one or more views.
- Definition of a *subroutine library* for common types of operations on the data. While the code in the individual subroutines manipulates the tables, the developer operates only through the subroutines, whose details are invisible.

We discuss each in turn.

7.4.1

Schema Independence Through Views

A *view*, a standard abstraction mechanism in relational database technology, is a `SELECT` statement on one or more tables in the physical schema. The statement selects columns of interest across these tables and may restrict the rows that are returned through Boolean conditions.

To achieve schema independence, it is essential that the view’s columns are “aliased” – that is, referred to by names that are typically different from the physical column names in a given schema. The view itself behaves like a “virtual table”: that is, the user who invokes it in a `SELECT` statement does not know (or care) whether the view corresponds to a single table or to several tables.

The ISO SQL Standard actually specifies such a mechanism for a special purpose. It defines a standard set of views (in a schema called `Information_Schema`) whose use allows a software developer to get information about the components of a database, such as the tables, columns, inter-table relationships and so on. Such information is essential for a variety of software applications such as database browsers, querying and reporting software and user interface/application generators. Being able to access the metadata behind a database in a standard way greatly simplifies the developer’s task of building software that can be targeted to a variety of RDBMSs, without having to rewrite the software for each platform. Most of the current versions of the mainstream multi-user RDBMSs implement `Information_Schema`: a notorious exception is Oracle.

Note that while I’ve called a view a virtual table, this is an over-simplification. From the perspective of *reading* data, a view is indeed indistinguishable from a table. However, if

data needs to be *changed*, such transparency is not possible. For views that are built from multiple tables, there are restrictions on the ability to insert, update or delete data. Specifically, any statements that attempt modifications must reference columns from only *one* base table, and those columns cannot be *derived* in any way (e.g., they cannot be the result of computations or aggregate functions such as SUM or COUNT).

For example, in a view that combines a patient's encounter data (stored in one table) with demographics data (stored in another), you can't have a single statement that attempts to change both types of data: you must divide the task into two statements, updating each table separately.

Such a restriction conflicts with the objective of information hiding – in the above example, it would require the developer to *know* which columns come from which table. If such knowledge is required to be able to change data, you might as well access the underlying tables directly: views by themselves are useless for such a purpose.

Note: Information_Schema works precisely because it is *read-only*. Changes to the schema are only permitted to be made through Data Definition Language SQL statements (such as CREATE TABLE): data-manipulation statements that insert, update and delete the contents of the views are not allowed.

7.4.2

Schema Independence Through a Subroutine Library

A subroutine library achieves information-hiding with respect to operations that change data. The library is typically provided by the vendor: the developer passes a number of parameters to a particular routine that is intended, say, to add new data to a view. Inside the subroutine, individual tables are accessed, and the code also performs housekeeping chores, such as ensuring that the individual tables are synchronized. Synchronization may involve manipulating integer unique-identifier columns, whose existence may be hidden from the user of the subroutine.

If one needed to update certain demographic data and clinical variable values for a given patient, one could have a routine that took a set of parameters, updated demographics data with some parameter values, and then updated the clinical data with the remainder. The caller of the routine does not know or care which physical tables and columns are actually manipulated for this purpose.

RDBMSs such as MS SQL Server and Oracle provide an extensive set of such subroutines (called “stored procedures”) to allow the administrator and developer to perform various query and maintenance tasks. The objective is not always information hiding for database changes: data retrieval can also be performed. As for all well-designed subroutines, a developer who is able to write a single-line parameterized subroutine call is much more productive than when required to compose multiple SQL statements manually. I emphasize,

however, that stored procedures for certain specialized purposes such as database maintenance and tuning are proprietary: this is one area that ISO has stayed away from.

In the extreme application of this approach, the database schema is simply not documented, and everything must be done through the library. The advantage of hiding the schema is that if it changes as new versions are released, developer-written code does not break: the vendor provides a guarantee that the individual subroutines will continue to work, even though their details have changed under the hood. NCBI took this route for their Entrez software-development toolkit, because the tables used to store gene and protein data were undergoing frequent revisions. Certain medical-content providers (notably First DataBank¹⁶ and Health Language¹⁷) also insist that developers work only through their libraries: in the case of First DataBank, their schema has been deliberately obfuscated with cryptic table and column names – almost certainly out of concerns about content theft – and using the vendor’s Application Programming Interface (API) is about the only way to get any useful work accomplished.

Most vendors (for example, Multum,¹⁸ which provides drug-related content, or Microsoft, as stated above) support both standard views and stored procedures. The problem with a complex database schema is that, while there are only a limited number of ways to *change* data, it is very hard for a vendor to anticipate every possible combination of data elements that any given developer might want to *retrieve*. Consequently the subroutine library can become enormous if such anticipation is attempted: I’ve seen one library with 300+ routines, many of which are minor variants of each other.

Few developers are prepared to invest energy in an effort with such a tedious learning curve, merely to achieve what they could accomplish in a few minutes with a visual SQL query composer that directly accesses the schema. In such a circumstance, it is more logical to simply trust your customers and provide views, or just document the tables in the schema well, so that your customers may create views for data elements that they commonly access together.

7.4.3

The Virtual Medical Record

An effort is under way at Health Level 7 (HL7), the organization that supports international healthcare standards, to define a standard set of views intended to allow medical software to manipulate clinical data in a way that is independent of an actual physical EMR schema. The model comprising this set is called the *Virtual Medical Record* (vMR).¹⁹ It is currently defined using a Unified Modeling Language (UML) class diagram (a more full-featured equivalent of an entity-relationship diagram). The names of some of the standard views (e.g., Person, Clinical Observation, and Allergy) and the virtual columns within those views have been defined, along with their data types. The data types themselves are based on a standard, ISO 21090 (Health Informatics – Harmonized data types for information interchange), which has been thoughtfully designed: each individual data type, and the operations permissible on it, is precisely defined.

The vMR itself is derived from the HL7 v3.x Reference Information Model (RIM), but addresses only the clinical data subset of it. The HL7 RIM also concerns itself with administrative and financial data, but these are addressed at too broad a level to form the basis of

an actual implementation of an administrative or financial sub-system of a production EMR. The RIM is concerned with the more limited goal of getting different EMRs to be able to exchange data with each other.

As of this writing, however, it is only fair to point out that the vMR, which is currently defined using UML, is not fully fleshed out to the level of detail that would support cross-EMR software development: in many respects, it does not support the creation of fully operational decision-support software even for a specific EMR.

- Even for the areas that it is intended to cover, the vMR is under-specified. The most widely used ISO data type in the vMR model is CD, which stands for coded data derived from a particular system (typically a controlled vocabulary). For example, in the definition of the view Allergy, we have the virtual columns Agent, Reaction (what happens/happened to the patient), Reaction Intervention (what treatment was required), and Severity, all of type CD, which is reasonable. (For example, the agent would be derived from a controlled vocabulary of substances.)

However, Severity is an *ordinal* measure: a severity of “mild” is less than one of “life-threatening”. As we’ll see later, one problem with the majority of controlled vocabularies (this is true of the widely used SNOMED as well as UMLS) is that they do not represent the concept of ordinal scales. Guidelines that employ ordinal values do so because they need to represent semantics that merely coded/enumerated values cannot. For example, in a life-threatening infection, a documented history of a very mild allergy to a particular antibiotic that is the only one that will work does not rule out its use, because the patient can be given concurrent anti-allergy medications.

I do not know of anyone who has actually tried to port an operational framework of alerts written in Arden (which accesses the physical schema of an EMR natively) to employ the vMR as a layer of indirection, and utilize standard controlled clinical vocabularies as the underpinning of their knowledge representation for coded data. But I’m certain that any such attempts will run right into the above limitations. The only way to work around these is to augment the model locally with ordinal-value information. The danger of doing this without the participation of a standards body is that the framework will end up being specific to a particular implementation, and the goal of universal deployment, which is the motivation for employing the vMR in the first place, will be thwarted.

- The vMR may need to be extended to cover parts of the EMR that it does not currently address. For example,
 - In an EMR, the clinical data does not stand alone: it ties in to administrative and workflow data as well. For example, while it is trivial to identify diabetics in a database and retrieve their most recent HbA1C values, electronically implementing the standard American Diabetes Association guideline – which states that HbA1C should be measured every 6 months in stable, well-managed diabetics and every 3 months in inadequately controlled diabetics or those whose therapy has changed – requires access to the clinical encounter data as well as integration with the EMR’s scheduling subsystem. The vMR does not concern itself with the latter type of data.
 - For the workflows involved in complex guidelines, I’ve stated that because there are temporal gaps in the flowchart, the patient’s flow through the chart must be

“suspended” at particular points and “resumed” later. Supporting such a feature ultimately requires “current patient state” information to be stored in a database. However, neither the vMR, nor the EMRs of which the vMR is an abstraction, address this issue, except in a very general sense - e.g., the progress note.

- As stated above, standard views are not sufficient by themselves to achieve schema independence when it comes to changing data. The operations on the vMR are *not* guaranteed to be read-only: the results of certain computations executed by a step in a guideline may need to be stored as computed variables. For example, we introduced the computation of body surface area and glomerular filtration rate in an earlier chapter.

Given such a limitation, a stored-procedure library interface for data changes also becomes necessary. However, such an interface has not been specified. In theory, given a UML data model of a vMR, someone intimately familiar with a particular EMR schema can implement a data-change model that hides the physical schema completely. However, if ten different teams, working independently of each other, implement what is functionally the same library for ten different EMRs, the chance of being able to automatically port working source code written for one EMR to another are vanishingly small, so that you now have the curly-braces problem recurring in a different form.

In summary, if you attempt to use the vMR at present, you can expect to look forward to the tribulations that are the due of all pioneers. I’m sure this situation will improve, however, as pilot implementations of the vMR identify its limitations, so that the resulting feedback improves the model.

7.4.3.1

Mapping Guideline Data Elements to Structured Elements in EMRs

Even if the vMR becomes fully mature, individual medical institutions who wish to implement alerts or complex guidelines are likely to find that it addresses only the tip of the iceberg. An outstanding challenge involves matching the structured data elements required by a set of guidelines to those *actually available* in a particular EMR. This is because most data elements in an EMR are not provided by the EMR vendor, but need to be added in-house over time based on local needs.

The work of Peleg, Keren and Denekamp²⁰ attempted to map the elements required by the complex guideline (designed by the American College of Foot and Ankle Surgeons, 2000) for management of diabetic foot problems, into two separate systems; they then tried to map data elements in 16 other guidelines to these two systems. Both systems were developed in Israel: one of them was specially used for diabetic foot problem management, and the other was a general-purpose EMR.

The authors devised a methodology that mapped a data element required by a guideline to code (including SQL statements) that fetched or computed the required data. In some cases, the mapping was direct (i.e., fetched with a single SQL statement) but in other cases, complex expressions using Boolean and relational operators had to be composed. The authors give the example of neuropathic arthropathy (damage to joints caused by injuries

resulting from impaired pain sensation). This finding must be determined computationally from a combination of several other findings. While their paper describes their methodology with reasonable clarity, the expression composition requires individuals with deep knowledge of both programming as well as the contents of a given EMR, and it must be accomplished laboriously, one expression at a time.

These authors claim that there were no unsuccessful mappings, but I suspect that the set of elements they chose were those that had a chance of being mapped (i.e., elements that actually existed in structured form as opposed to being recorded in clinical narrative, or elements not captured at all). For example, they mention that one of the EMRs did not capture information on Doppler arteriography test results (which are used to evaluate blood flow to the foot). You can't map to what doesn't exist: to implement a guideline, you have to either change the way you manage individual diseases (e.g., by purchasing and deploying new equipment that you didn't have earlier in your institution), or you have to depart from the guideline in order to accommodate the constraints of local economic conditions.

7.5 Summary

Arden Syntax is a historically important effort that had the effect of identifying the major challenges in implementing complex clinical support. Portability of medical logic with Arden was compromised because Arden failed to address the problem of creating an abstraction layer that would deal with variations between different EMR implementations. These challenges persist in the implementation of complex guidelines. The HL7's Virtual Medical Record effort aims to address this problem, but it is still not fully mature. The task of converting prose guidelines into actionable form that can be employed in clinical scenarios is complicated by issues such as ambiguity, incompleteness and the challenge of mapping data elements required by a guideline to those that are actually available within a given EMR instance.

References

1. Wang D. Knowledge representation with Arden. [cited 9/1/10]; Available from: <http://people.dbmi.columbia.edu/homepages/wandong/KR/krarden.html>. 2004.
2. Jenders RA, Sujansky W, Broverman CA, Chadwick M. Towards improved knowledge sharing: assessment of the HL7 reference information model to support medical logic module queries. AMIA Fall Symposium, 1997.
3. Ludemann P. Mid-term report on the Arden Syntax in a clinical event monitor. *Comput Biol Med.* 1994;24(5):377-383.
4. McConnell S. *Code Complete*. 2nd ed. Redmond: Microsoft Press; 2004.
5. Wright A, Sittig DF. A framework and model for evaluating clinical decision support architectures. *J Biomed Inform.* 2008;41(6):982-990.

6. Pryor TA, Hripcsak G. The Arden syntax for medical logic modules. *Int J Clin Monit Comput.* 1993;10:215-224.
7. Sailors RM, Bradshaw RL, East TD. Moving Arden Syntax outside of the (alert) box: a paradigm for supporting multi-step clinical protocols. AMIA Fall symposium 1998; 1998.
8. Pryor TA, Hripcsak G. Sharing MLM's: an experiment between Columbia-Presbyterian and LDS hospital. *Proc Annu Symp Comput Appl Med Care* 1993: Washington DC; 1993:399-403.
9. Teitelbaum D, Guenter P, Howell WH, Kochevar ME, Roth J, Seidner DL. Definition of terms, style, and conventions used in A.S.P.E.N. Guidelines and standards. *Nutr Clin Pract.* 2005;20(2):281-285.
10. Agency for Healthcare Research and Quality. National guideline clearinghouse [cited 11/2/10]; Available from: www.guideline.gov. 2010.
11. The Society of Obstetricians and Gynaecologists of Canada. Ovulation induction in polycystic ovary syndrome. *Int J Gynaecol Obstet.* 2010;111(1):95-100.
12. Damiani G, Pinnarelli L, Colosimo SC, et al. The effectiveness of computerized clinical guidelines in the process of care: a systematic review. *BMC Health Serv Res.* 2010;10:2.
13. Shiffman RN, Karras BT, Agrawal A, Chen R, Marengo L, Nath S. GEM: a proposal for a more comprehensive guideline document model using XML. *J Am Med Inform Assoc.* 2000;7(5):488-498.
14. Moskovitch R, Shahar Y. Vaidurya: a multiple-ontology, concept-based, context-sensitive clinical-guideline search engine. *J Biomed Inform.* 2009;42(1):11-21.
15. Turner R, Dufour J-C, Bouvenot J, Ambrosi P, Fieschi D, Fieschi M. Textual guidelines versus computable guidelines: a comparative study in the framework of the PRESGUID project in order to appreciate the impact of guideline format on physician compliance. AMIA Annu Symposium 2006; 2006:219-223.
16. First Databank. First Databank [cited 11/1/10]; Available from: <http://www.firstdatabank.com/>. 2010.
17. Health Language Inc. Health language [cited 11/1/10]; Available from: <http://www.healthlanguage.com>. 2010.
18. Multum Corporation. The VantageRx database [cited 6/8/09]; Available from www.multum.com/VantageRxDB.htm. 2009.
19. Health Level 7 Virtual medical record [cited 11/3/10]; Available from [http://wiki.hl7.org/index.php?title=Virtual_Medical_Record_\(vMR\)](http://wiki.hl7.org/index.php?title=Virtual_Medical_Record_(vMR)). 2010.
20. Peleg M, Keren S, Denekamp Y. Mapping computerized clinical guidelines to electronic medical records: knowledge-data ontological mapper (KDOM). *J Biomed Inform.* 2008;41(1):180-201.

8.1 Introduction

Rule-based software systems are a useful technology for addressing problems calling for a high degree of agility in terms of being able to respond to frequently changing external circumstances. They are an offshoot of 1960s “expert system” research – at one time, they were popular for implementing software that performed diagnoses, whether of human disease or automobile problems.

Briefly, a rule consists of a construct like the following:

IF *condition* THEN (*perform some actions*)

for example,

IF headache = present and time_of_headache = morning and vomiting = present and nausea = absent then write “rule out increased intracranial pressure”.

A rule-based system consists of a database of rules (a “rule base”) which may be segregated into collections called *rule-sets*, and a program called an *inference engine* (“rule engine”) that operates on the rules. The important aspect of the system is that the rules, while collectively behaving like a program, are stored as *data*, which is interpreted by the engine. This allows new rules to be added or existing rules to be changed or deleted, without having to modify the engine.

Modern rule engines allow the rules to be extremely sophisticated.

- The *condition* of a rule can be arbitrary complex with compound Boolean logic operators (and, or, not). One can also incorporate mathematical operations as well as functions (including developer-defined functions) in the condition expression. While in our example, all individual sub-conditions use the equals (=) operator, any other comparison operator (e.g., greater-than, less-than, not-equals) can be used if needed. Originally, when the capabilities of rule engines were very limited, rule-based-approaches were regarded as a form of *reasoning with symbols* (as opposed to reasoning quantitatively with numeric data). With modern engines, however, this distinction is spurious. There is nothing to prevent a function from performing quantitative operations of arbitrary complexity. These functions

are typically written in a traditional programming language, compiled into executable form, and then called from the engine. It's a case of fitting the solution to the problem.

- The variables in rules (e.g., *headache* and *vomiting* in the above rule) are defined by the user, as are their types. (For example, headache may be an enumerated variable that can take one of the values (present, absent).) At runtime, the collection of all variables is called *working memory*: the value of a variable may be known or unknown ("null"). Defining variables before they are used in rules prevents errors such as misspelled variable name or testing against the wrong value.
- The *action* part of the rule can perform an arbitrary set of actions. Individual actions include the following:
 - Most commonly, the value of a variable is changed. This has the effect of activating other rules that depend on this variable. For example, the above rule would be considered for activation if a previous rule had determined that headache occurred in the morning (based on the actual time of day that the symptom occurred).
 - Text may be output to the user, or e-mail dispatched.
 - The state of the environment may be altered, e.g., by writing to a file or altering the state of a database.
 - A developer-defined subroutine may be executed. This routine can also do almost anything: it may, for example, ask the user for specific information.
 - The rule engine will terminate its run, typically when it has met its objective.

The syntax used by the rules may be proprietary or may adhere to that of a standard programming language. The need to commit to a particular syntax is important because, as for the similar situation with computed formulas (which we discussed in the chapter on interfaces to EMRs), expression syntax varies between languages: for example, the not-equals symbol is " $\neq$ " in the C/Java/JavaScript language family and " $\neq$ " in the Visual Basic family.

The use of rule-based systems in medicine has a long history, though almost none of these systems made the transition into actual production systems. Randolph Miller¹ provides a comprehensive survey and bibliography, though I should state that there are relatively few lessons to be learned by today's software engineers by actually exploring the design of the rule-based or quasi-rule-based systems described. Modern rule engines, and even the general-purpose programming environments for creating them, have advanced so much that the capabilities of the systems discussed, while path-breaking at the time, seem extremely primitive. However, Miller and Masarie² provide an explanation – rooted in medico-social and human-factors issues – why many of these systems never took root, and these lessons are valuable to anyone who wishes to avoid repeating history.

When a rule-based system is used for diagnostic purposes, an individual rule may not necessarily be 100% accurate – i.e., if its validity is tested with a large volume of real data, there may be both false positives and false negatives. To be useful, however, both its sensitivity and specificity must be high. In some rule systems that are specialized for diagnosis, a rule is associated with a number (typically a decimal between 0 and 1) that represents a crude probabilistic measure of its accuracy. This way, if a succession of rules is applied,

these measures can be combined in some fashion to give an overall statistic that represents the confidence in the prediction. In most modern rule-engines, however, this approach seems to have been de-emphasized, in part because these numbers, when assigned by human beings manually without the use of data sets, tend to be extremely arbitrary, and statistics derived from these guesstimates can be dubious.

8.2

Operation of the Rule Engine: Forward and Backward Chaining

In general, the operation of a rule engine can be described as follows: The engine starts, one or more rules are “activated”. If their conditions evaluate to True, the actions are executed. This results in other rules being activated, and so on, until the engine terminates.

The starting of the engine is triggered by one of two strategies:

- Setting of the values of certain variables (e.g., by the user through a form-based dialog) will activate the rules that depend on these variables. After these rules execute, the values of other variables are set or changed. This turn activates another set of rules. This strategy – moving from known data to the rules that depend on them – is called *forward chaining*.
- The user specifies one or more goals, and the engine then tries to identify *actions* whose execution might satisfy this goal. The *conditions* for these rules are then tested (with each sub-condition being treated as a goal). This strategy is called *backward-chaining*. The Prolog programming language uses backward-chaining: some rule engines, such as Jess, fake backward-chaining through forward chaining: consult the Jess documentation by Friedman-Hill³ for details.

Most modern rule engines use forward-chaining, for two reasons:

- This approach is better adapted to a user interface (such as a standard form) that prompts the user for known information before invoking the engine.
- A large rule base can contain thousands of rules. If a rule engine evaluated every rule in the database consecutively with each activation cycle (this was the case for older systems) the operation of the engine would be unacceptably slow. A very efficient algorithm (Rete, devised by Charles Forgy⁴) is available for rapidly identifying candidate rules for activation in a forward-chaining strategy. Essentially, Rete maintains in RAM a list of rules that are matched or partially matched by the current working memory (i.e., the values of all known variables): this list is recomputed only when the values of one or more variables change, and the recomputation is selective for those rules that depend on the changed variables. Rete does this by decomposing a compound condition into a branching tree structure where lower level nodes correspond to atomic sub-conditions (e.g., headache=true) and are linked to other lower level nodes by intermediate nodes corresponding to the AND operator.

8.3

How Rule Engines Work

The Rule engine interprets a rule by evaluating a condition and executing the associated actions if the condition evaluates to true. Condition evaluation can use one of two approaches:

1. If the rule base is not too large (i.e., computational efficiency is not critical), and the syntax adheres to that of a standard and freely available scripting language (JavaScript, VBScript), the scripting language's Eval function (introduced in the earlier chapter), which evaluates text as though it were program code and returns a result, can be used.
2. If computational efficiency is paramount with a large rule base, a condition is decomposed at design time into a form that allows efficient evaluation. Taking the condition in the example:

headache=present and time_of_headache=morning and vomiting=present and nausea=absent.

Since all the conditions are linked with the AND operator, the result would be false if any one of the sub-conditions is false. So we could restructure this as AND (headache=present, time_of_headache=morning, vomiting=present, nausea=absent). This information can now be restructured as in the following diagram, Fig. 8.1.

The top level node represents the Boolean operator, the intermediate nodes represent the relational operators (here, all are “equals”), and the lowest level nodes represent the variables and the values they are to be compared against. Evaluation occurs by evaluating one of the lower-level relational expressions (e.g., headache=present). If this expression turns out to be false, the end-result is guaranteed to be false, so the evaluation can be “short-circuited”. If true, the other relational expressions are evaluated in turn.

This structure (stored in RAM) is also used in implementations of Rete. The top-level node is connected to a node representing a Rule (typically, this node only stores the rule's ID). The individual variable nodes are also connected directly to the Rule nodes: given a variable, one knows immediately which Rules depend on it, and given a Rule, one can immediately determine the variables that it depends on.

I've already stated earlier that instead of a variable being tested against a value, an arbitrary function of the variable (or of several variables) can be tested instead. So here, one of the terminal nodes would be an expression that invoked the function.

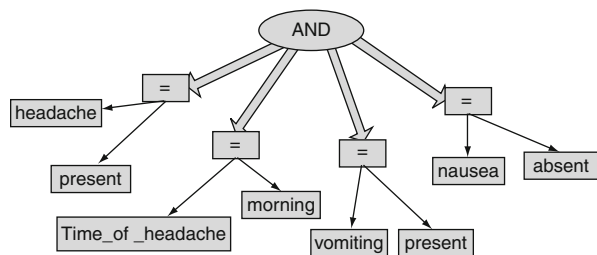


Fig. 8.1 Decomposing a rule condition into a structure that allows efficient runtime evaluation

The function's result can be determined with the Eval mechanism. Eval is relatively inefficient computationally, so its use is restricted only to those situations where it is absolutely necessary.

Storing the decomposed rule condition in the rule base involves using what is called a *stack* representation, which was at one time employed by several scientific calculators. While the notation corresponding to this representation is more machine-friendly than human-friendly, it has the advantage of eliminating the need for parentheses where individual operators have a specific precedence. A Boolean operators, the order of precedence from highest to lowest is NOT, AND, OR. All relational operators have the same precedence.

In a stack representation, one can use a *prefix* notation (operators precede operands) or *postfix* (vice versa). The prefix representation of the above condition (with each token representing a separate record in the database) would be:

AND=headache present=time_of headache morning=vomiting present=nausea
absent

It is straightforward to construct a tree structure, as in the above diagram, from a prefix representation.

8.4

The Strengths of Rule Frameworks

Rules are effective as a means of representing knowledge that can be represented in IF.. THEN format. Because many problem-solving approaches rely on an IF... THEN approach, at one time it was believed that rules might be a universal, general-purpose approach to model the behavior of experts. The Rule metaphor influenced the design of many systems, including Arden Syntax, where a Medical Logic Module's alert is driven by the test for a condition.

One can, of course, implement IF..THEN statements in traditional programming languages. The distinguishing feature of rule engines, stated earlier, is that *rules are stored as data*. Interpretation of data is always computationally much slower than the execution of a compiled program, but has the advantage that, in circumstances characterized by a high degree of continual change, it is much faster to change data than to change program code.

- Such speed of change is important in situations where changes to the behavior of software must be made in hours or days. An example is tax software, which must often incorporate late-breaking changes in federal or state tax codes, after the software was already shipped to its customers. Commercial tax software typically incorporates a stripped-down rule engine, and enabling update of the software by downloading files containing rules from the vendor's Web site is faster than the creation of a patch that

modifies the software (which approach would be necessary if the rules were embedded in the program code rather than stored as external data).

- The speed and ease of change also makes it easier to experiment – which means that even if an experiment fails, it is likely to fail relatively quickly and cheaply, and revisions can be made rapidly. For many applications, such as determining credit or investment risk, IF..THEN statements can be used to embody knowledge (which may not have crystallized within an organization at the time the application was conceived). Rule engines offer a simple way of creating and iteratively revising and testing a prototype that, if successful, can be converted into a traditional program. Certain engines contain tools that will partially automate this for you.

More important, such changes do not necessarily need to be made by programmers. The rule-authoring environment of modern rule-engine software has evolved to become highly user-friendly, and can be operated by non-programmer power users. In theory, then, a non-programmer expert who knows a particular field thoroughly and who can express that field's knowledge as rule can effectively act as a software developer. In practice, disciplined software-engineering practices are as important as in traditional programming: subject-matter experts have to make the effort to learn these skills. Also, implementing custom functions to be used within rules, must be done by software developers: these functions, if adequately documented, can then be used by the power-users.

In organizations where the IT department is notably backlogged with respect to application delivery (or simply too slow to respond efficaciously to changing circumstances) rule-engines have been marketed to non-IT management, promising the benefits of organizational flexibility, as well as a means of putting management rather than IT in the driver's seat with respect to software. As for all technologies, it requires considerable knowledge to know the limits of technology.

8.5

Challenges and Pitfalls of Rule-Based Approaches

We now consider the challenges encountered with rule-based approaches. Some of these are not specific to such approaches, but apply to any insufficiently-understood technology that may be hawked as a panacea by its vendor.

8.5.1

Conflicting or Redundant Rules

The flexibility afforded by the ease of rule addition and change can be double-edged. It is easy to inadvertently add rules that may conflict with existing ones: the effect may only be noticed when a system that was working well in the past now starts to behave strangely. As in traditional software engineering, one must evaluate the effect of a change to a system with extensive testing before going live.

Prevention being preferable to cure, however, it is more useful, when adding a new rule, to be able to determine which existing rules exist depend on the variables that are used in the new rule's condition. Similarly, if a rule alters the value of a variable in an action, it is important to know which rules would be consequently activated. If such rules are identified, both *conflict* – same or near-identical condition but significantly different actions – and *redundancy* – highly similar conditions and actions – may be discovered.

Identification of dependencies would seem to be essential functionality for any rule-editing environment, but not all systems offer this: while the freeware toolsets (e.g., Drools⁵) are getting better with each release, they lack this feature currently. The high-end systems, which offer “Business Rule Management Systems”, will provide this feature, but they cost about \$50 K minimum.

8.5.2

Software-Engineering Skills Do Not Become Obsolete

Rule engines promise to enfranchise management with respect to software development, but with modern engines, programming is necessary to be able to use them to their fullest capabilities.

- As stated previously, the development of libraries of functions that can be called from rules allows rule conditions to be more expressive. Rules for administrative purposes may take advantage of financial functions, for example. Some engines have in fact failed commercially because of their non-extensibility: a few did not even support elementary arithmetic in rule-condition expressions.
- Some rule engines will allow developers to create structured data types (“classes”) in a language such as Java, import them and then use them in rules. For example, a class “Patient” may have the properties Patient_ID, Gender, Date_of_Birth, Address and so on, as well as a function like Age_in_Years() which computes the Age from the current time and the Date of Birth. The advantage of structured data types is that rather than creating a large number of separate and disconnected variables, one can refer to multiple properties of a single variable instead, e.g., CurrentPatient.Gender. A structured data type is preferable where applicable, because it often corresponds to a row in a relational table with which the rule engine interoperates.
- It is necessary to understand programming concepts such as recursion (where a subroutine calls itself) to create certain types of rules judiciously. A common example given in most rule-engine manuals is a rule where a preferred customer with a total order of more than \$200 is given a discount of 5%. (Such a rule has the effect of lowering the cost of the order: in other words, the variable “order” is used in both the condition as well as in the rule action.) If the engine encounters a preferred-customer order with a value of \$400, the rule will fire, and a 5% discount will be offered, and the total order value will now become \$380. However, since \$380 is still bigger than \$200, the rule can fire a second time, triggering another 5% discount, and so on, so that the customer ultimately gets more than a 50% discount. To prevent this cycling, rule engines allow the rule creator to specify that a rule must have *non-recursive behavior*: i.e., it must fire

only once and then stop. To detect the possibility of anomalous behavior, it is desirable to have an authoring environment that red-flags a rule that has the same variable in both the condition and the action.

- Without sufficient programming expertise, it is easy to cobble together a system that works, but performs terribly to the point where users are frustrated by sluggish response. Lou Agosta⁶ cites an example of a poorly optimized system which multiple rules repeatedly fetched the same data from a back-end database instead of fetching it once and then caching it for reuse (which would have required modest programming).
- When the number of rules become large, it is necessary to have supporting software (a rules management system or RMS) that facilitates their maintenance. Among other things, the RMS should support the following:
 - An audit trail of which user creates or modifies a rule and when. Older copies of changed rules should be preserved so that one can go back to them if errors are discovered in changed rules.
 - Categorization of rules and search based on multiple criteria – date of creation/last modification, category, rules that depend on a variable or that alter a variable, and so on.

The operation of an RMS requires skills that are highly similar, if not identical, to those of database administrators. This is one area that domain-experts do not care to spend time on, but which is critical for long-term viability of a rules-based solution.

In summary, successful deployment of rule-based systems involves a partnership between domain experts and software professionals.

8.5.3

Comprehensiveness and Reliability

A rule-based system is only as good as the rules that it operates upon, and hence is influenced by the knowledge of the individuals who specify the rules. It is now that one may discover that a self-proclaimed expert is not necessarily so and that there are limits gut instinct. A set of rules must be *reliable* (i.e., reasonably accurate) as well as *comprehensive* (i.e., covering most of the possible range of inputs).

- *Comprehensiveness* can be facilitated by using approaches such as decision tables, which enumerate all possible combinations of all possible values of variables, and specify the actions for a given combination. Several software packages will generate rules from decision tables, though it is possibly more straightforward to use a decision table directly in the circumstances where it is a good fit.
- *Reliability*: It is remarkably hard to come up with rules through simple intuition or “experience”, and such rules may not perform very well when tested. Therefore it is desirable to try to *induce* rules from large data sets. If the “action” of a rule involves classification of a single variable only, there are several algorithms, such as Ross Quinlan’s C4.5,⁷ which can generate a *decision tree* from such data using a statistical approach. (C4.5 was essentially made open-source through its detailed description by

Quinlan in a book: a new version, C5.0, said to achieve major improvements in performance, is proprietary.)

Quinlan's algorithm is used when where the input and outcome variables are binary or ordinal: if continuous, they must be converted to ordinal by discretizing into intervals. It involves progressively splitting a dataset into two or more subsets based on the value of an input variable so that the separation between the groups with respect to the outcome variable is maximized (in mathematical terms, this is called "maximizing the information gain"). The process is repeated for each subset (using a different input variable) until the information gain with each successive split drops below a threshold of significance. As for decision tables, several programs will generate rules from decision trees, though again there is no reason why the tree can't be used directly.

8.5.4

Alternative Approaches May Be More Appropriate

A rule-based approach is by no means universally applicable, nor is it always the most efficient means of representing knowledge even when it may be applicable. It takes fairly deep knowledge of both subject matter as well as computing to understand where rule-based approaches will and won't work. Since such knowledge is not commonly present in a single individual, it means, once again, that domain experts and experienced software developers must collaborate closely. As a result of the collaboration, the latter end up learning a lot about the expert's domain, and the expert in turn picks up significant knowledge about software engineering practices.

For purposes such as diagnosis, for example, rules are only one of several formalisms that may be employed. When there is a single outcome variable (e.g., likelihood of survival in a particular disease given a pattern of findings), standard statistical methods based on multi-linear or logistic regression should be tried out first, and non-linear methods such as neural networks tried next if prediction with the former is not satisfactory. Decision trees are an alternative to neural networks: if both approaches turn out to be equally reliable for a given data set, decision trees have the advantage of being more readily understood by a layperson.

8.6

Case-Study: Implementing a Rules Management System

If you use an open-source environment such as Drools to create a rule-based system, you will eventually need a means of managing your rule content. While the Drools tutorials illustrate a simple approach that stores content in a spreadsheet, this is hardly recommended when your rules need to be categorized into large numbers of rule sets. Figure 8.2 illustrates a schema that stores the content of Rule Sets, Rules, Variables used within Rules and dependencies of Rules on Variables. The advantage of a schema is that it allows you to organize rule content systematically as well as maintain dependency information (e.g., rules that use

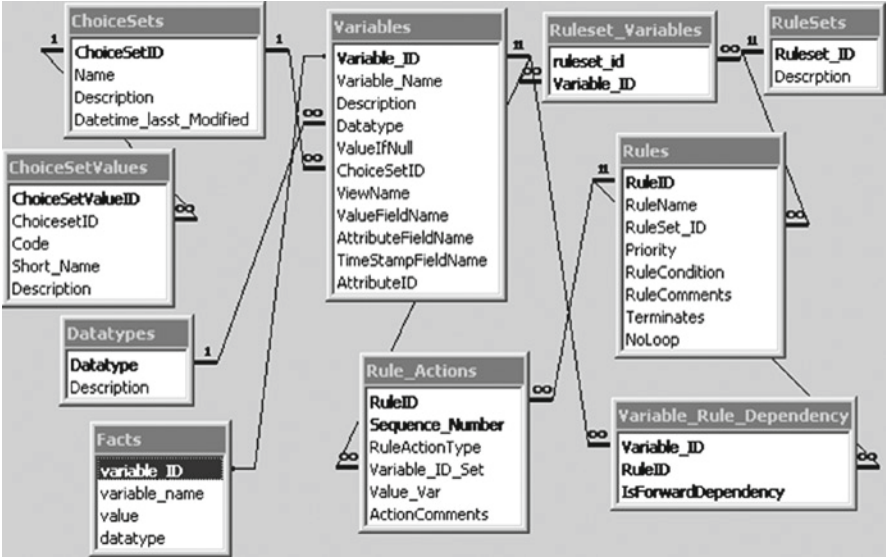


Fig. 8.2 A schema for managing rule content

a given variable, variables changed by a given rule) that can be queried across multiple rules and rule sets. Since individual rule sets are typically maintained by different people (even though they may use some of the same variables), the schema is the basis of a shared repository: you may implement selective editing access if needed.

This schema was originally designed to be embedded within a larger database, where the rule actions primarily manipulated data through SQL statements. We built our own lightweight rule-engine: this work is described in Brandt et al.⁸ which the reader may consult for additional details. The engine is definitely not in the same league as the Drools/Jess engines, but for our relatively simpler and specialized needs, we had to either extend these engines to operate on database contents, or we had to implement a basic expression evaluator. We chose the latter path, using VBScript as the expression language and its Eval() function, described earlier in the User Interface chapter, as the basis of dynamic expression evaluation. The nice thing here is that one can extend the system with an arbitrary set of sub-routines/functions that can be called via Eval(), and most or all of this functionality can be tested in a design-time environment.

If we had to do it all over again today, however, we would probably use an Object-Relational interface such as Hibernate™ to do the database access, and use the Drools engine instead of building our own. The rule-storage component of the schema is still necessary, however.

Variables: The columns that define a variable are the same as those for Attribute definitions in an EAV/CR design. In fact, if your rule engine is embedded within an EAV/CR schema which is described in Chapter 11, you would eliminate this table and use the *Meta_Attributes* table of that chapter instead.

RuleSets (RuleSet ID, Description) are used to group Rules by the problem they are intended to address.

A **Rule** has a symbolic *Name* and a *Condition* (long text) to be evaluated. *Comments* describe the rule's purpose; *Priority* indicates preferred order of execution among currently activated rules. This field's role is limited to serving as a developer-supplied hint: its value is used only as a tie-breaker to determine the order of equally prioritized rules that have been activated by the Rete algorithm, discussed earlier. *Terminates* (Boolean), if True, means that the engine halts after the rule is executed. *NoLoop* (Boolean, default True) means that the rule is non-recursive.

Rule Actions, which is related many-to-one to **Rules**, records the sequential actions that are performed (in order of *Sequence_Number*) if the rule's condition evaluates true. The *Rule Action type* is one of "Set a variable's value", "Print a Message" or "Execute Code". *Variable_ID_Set* refers to the variable whose value is set, if the Action type is Set Variable Value. *Value_Var* is a multipurpose field that indicates the expression that is used for assignment or printing, or the name of the procedure to be executed (with parameters).

For "execute" actions that invoke SQL, we created a simple "wrapper" function called `SQLExec`, which took a string as a parameter, and could be invoked from a scripting environment. If you are using an industrial-strength rule engine, you will have to implement something similar.

If you are using an existing rule engine, the contents of these tables allow you to emit code in the programming language of that engine. Note, however, that the syntax of the expression language used for the rule condition must be identical to that used by your engine.

RuleSet_Variables records a critical dependency: the variables on which a Rule Set depends. This allows a design-time check to ensure that all variable used in rule expressions for a given rule set actually apply to the rule set.

Variable_Rule_Dependency records which rules *depend* on a variable (forward dependency), as well as which variables' values are *set* by a given rule (reverse dependency). The columns *RuleID*, *Variable_ID* and *IsForwardDependency* (Boolean) are self-explanatory. This table is populated at design time by parsing the rule-condition and rule-action expressions and locating instances of variable references (variable names enclosed in square brackets).

This table is also used to check, using a standard transitive-closure algorithm,⁹ for *cyclical relationships*, where a rule directly or indirectly activates itself: while sometimes necessary, these can lead to infinite looping or wrong results if incorrectly specified. (The transitive closure¹⁰ of a given rule is the set of all rules that can be reached by traversing its

dependencies; it is computed by starting with a given rule ID, getting all reverse-dependency links and determining all the variables that depend on this rule. One then uses each variable in turn and traverses all forward-dependency links to determine all the rules depending on each variable. As new rules/variables are discovered, they are loaded in memory. The process continues recursively until either the lists cease to grow, or a cycle has been discovered: the latter happens when the list of newly loaded rules includes the starting rule.)

Facts: This table can be omitted if you are not building your own engine. It implements the component of working memory that deals with instantiated variables and their values. This table is emptied at the start of engine execution, and facts are “asserted” (that is, records are inserted) or their values updated as execution proceeds. This table’s existence is not critical: the developer may choose to replace it with an in-memory equivalent such as a hash table, which will give better performance in any case.

8.7 Conclusions

Rule engines are only one of the tools in the software developer’s armamentarium. It is this realization that led the developers of rule-engines to emphasize integration with existing programming language (and other software, such as database applications). However, their versatility (and the fact that it is not especially difficult to write a rule-execution engine) has led them to be incorporated in other software, such as workflow management systems, which I will consider in the next chapter.

References

1. Miller R. Medical diagnostic decision support systems—past, present, and future: a threaded bibliography and brief commentary. *J Am Med Inform Assoc.* 1994;1(1):8-27.
2. Miller R, Masarie F. The demise of the Greek oracle model of diagnostic decision support systems. *Methods Inf Med.* 1990;29:1-8.
3. Friedman-Hill E. JESS, the Rule Engine for the Java Platform. [cited 11/29/03]; Available from: www.ghg.net/clips/CLIPS.html, 2003.
4. Forgy C. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artif Intell.* 1982;19:17-37.
5. Browne P. *JBoss Drools Business Rules*. Birmingham: Packt Publishing; 2009.
6. Agosta L. Business Rules meet the Business User. [cited 11/2/10]; Available from: <http://intelligent-enterprise.informationweek.com/showArticle.jhtml;jsessionId=HYXHSZHIOSJO PQE1GHRSKHWATMY32JVN?articleID=167100318>, 2005.

7. Quinlan JR. *C4.5: Programs for Machine Learning*. San Mateo: Morgan Kaufmann; 1993.
8. Brandt C, Gadagkar R, Rodriguez C, Nadkarni P. Managing complex change in clinical study metadata. *J Am Med Inform Assoc*. 2004;11(5):380-91.
9. Ben-Gan I, Kollar L, Sarka D. *Inside SQL Server 2005: T-SQL Querying*. Redmond: Microsoft; 2005.
10. Goodman N. Bill of materials in relational database. *Info DB*. 1990;5(1):2-13.

9.1 Introduction

Attempts to use Arden Syntax for complex guidelines, such as that of Sailors et al.¹ discussed earlier, revealed the poor fit of the Arden model for a problem that it was not originally intended to address, as indicated by the numerous workarounds necessary to create an operational implementation. This and other similar experiences have motivated several research efforts which intend to address these limitations.

However, currently none of these approaches have resulted in operational frameworks working within a production EMR. While the work that has gone into their creation has helped the medical informatics field greatly in identifying the formidable problems involved in complex guideline implementation, they should be considered experimental simulators at best (unlike Arden, where operational implementations actually exist).

One major thrust of several approaches in complex-guideline implementation has been to borrow ideas from research in workflows, to which I will provide a fairly detailed introduction. A major reason for the current emphasis on workflow software is that it represents the latest incarnation of the silver bullet with respect to the software-application-backlog problem: i.e., software applications take too long to develop, and when they are finally ready, the situation for which they were developed has changed. Therefore it is desirable, in theory, for knowledgeable, non-programmer, power-users (e.g., clinicians, administrators, managers) to be able to create production software.

This chapter will, after introducing workflow technologies in some detail, try to separate myth from reality, both with respect to how far the workflow metaphor can be applied to guidelines, and with respect to real limitations of the technology and the weakness of existing standards.

9.2 What Is a Workflow?

In informatics, a *workflow* is a model of a set of connected operations, involving a combination of humans and hardware/software, which are necessary to carry out a standard process. Examples of such processes are the movement of individual parts down

an assembly line in manufacturing, the scheduling of a patient, collaborative editing of a controlled medical vocabulary, or collaborative software development. Pictorially, a workflow may be represented as a kind of *flowchart*: as we'll see later, workflow software often lets non-technical individuals design a workflow using diagramming tools.

The characteristics of workflow processes are the following

1. They involve multiple individuals or computational/software units, often beyond the span of a single organization. For example, the sub-process that leads to detection of low levels of a given item's inventory are different from the sub-process that involves its reordering from an external vendor, and the sub-process involved in receipt of a new stock of the delivered item. Based on the organization, a step in the process may be either entirely automated, or require some human oversight or intervention. Where humans are involved, they need to be notified to begin their involvement in a process, and their individual actions must be tracked. There may also be timelines for certain actions (and electronically generated reminders to ensure that these timelines are not exceeded).
2. Their "life-times" – duration of running after initial activation – are highly variable, taking minutes (e.g., inventory reorder) to months or years (e.g., evolution of projects from proposal to completion). The latter durations are much longer than the maintenance cycles of most software and hardware, which would involve operations such as periodic system shutdown for software upgrades. Therefore a workflow must be *persisted*. "Persisted" is a fancy way of saying that the state of a given workflow must be saved to a database, from which it can be restored later when it resumes. Persistence is especially important given that, for the long-duration workflows, a process spends the overwhelming proportion of its time in a "waiting" state.
3. Certain steps in a process may occur in parallel. For example, a particular purchase may need to be approved electronically by at least two individuals, but the approval need not be sequential – notifications can be sent to both individuals simultaneously. The process must be designed so as to be robust in the presence of contingencies: for example, the intended human recipient of a notification may be unavailable or even have left the organization, and electronic communication outside an organization may not be reliable. A large part of the complexity of workflow design stems from the need to anticipate such contingencies and specify "fall-back" operations that "compensate" when contingencies occur.
4. At any point in time, there may be hundreds or even thousands of running workflow instances. By "instance", we mean that the same workflow runs with different data, e.g., numerous items may be in a reordering workflow, and at various stages of the reordering process. However, because real-time response is not always needed, a given instance does not need to continually tie up a CPU waiting for something (an external event) to happen. Instead, it must be activated when needed, consuming minimal machine resources in the interim. In many circumstances, such as the workflow involved in online purchase of an item, the computational infrastructure supporting workflows must be highly scalable, recruiting more hardware units as needed. (Tens of thousands of customers may be accessing Amazon online at any time.)

5. A given instance must be *auditable*: that is, individuals with the necessary authority must know at what stage of the process that instance is in. Since, as stated earlier, hundreds of instances may be active at any time, the workflow software also needs to generate an electronic trail in the form of reports and documentation.
6. Because of the long lifetimes of certain processes, it is quite likely that a given process may change even while numerous instances of it are still running. For example, an organization may be acquired by another, or policies governing a process may be changed in response to new circumstances. Such changes must be allowed without disrupting what is already running: in other words, they must be highly *flexible*, with support for the existence of multiple versions of a process.
7. The “silver-bullet” train of thought, which has led to the creation of certain graphically-oriented “workflow languages” that I will discuss later, asserts that workflows should be capable of being *authored* by non-technical individuals who lack programming skills but who understand the organization’s business or part of it – hospital administration, inventory management, human resources and the like. Such a paradigm of workflow authoring is not entirely unrealistic, but, as I’ll emphasize, it has its limitations.

9.3 Is a Workflow Software Framework Essential?

If the creation of an all-encompassing software framework that supports the design and implementation of production workflows seems like a tall order, it is. While academic research into workflows goes back to the 1980s (for informatics applications) or even the 1900s (if we consider the work of Frederick Winslow Taylor in the early 1900s² on the conceptualization of the assembly line and specialization of human tasks), software to support workflows based on the above capabilities has evolved only in the last 10–15 years or so.

Workflow software that provably meets all the above needs, which will work for any type of organization, and is affordable for the individual developer or small business, does not exist. While “universal” (and very pricey) commercial workflow management systems exist, investment in them involves a significant degree of risk. First, the fitness of a given commercial product for all of the capabilities identified above is not guaranteed. Learning all aspects of the software thoroughly and adapting it to meet the organization’s specific needs may involve extensive (and expensive) customization, which takes commitment of financial, time and people resources. Second, most implementations, as we’ll see, involve a large component of proprietary technologies or approaches. If the software turns out to be a mismatch to the organization’s needs in one or more respects, it is not easy to port one’s efforts to another vendor’s product: the investment in organizational resources beyond the initial purchase must essentially be written off.

The risk factor is the reason why, especially in smaller organizations, most workflows still tend to be implemented on a case-by-case basis through traditional programming techniques coupled with standard database technology. Such approaches may not be elegant, and efforts may be duplicative, but they are much safer. The cost of providing for such ad

hoc workflow needs is also seemingly small, since the task is perceived to be one of creating new features incrementally versus re-architecture of the system, which a general-purpose solution might entail.

Irrespective of whether one builds workflows by oneself using a home-grown approach, or whether one purchases vendor software for the purpose, realistic workflow processes connect disparate individuals and systems. In implementing such connectivity, one's efforts can be more productive if well-established, general-purpose technologies are leveraged. I'll now discuss the most important such technology, Web services.

9.4

Workflow and Web Technologies

While Web technologies have nothing to do with workflow concepts per se, they are particularly well suited to the implementation of workflows, even when these take place entirely within an organization. As we'll see, certain "workflow languages" explicitly assume the presence of a Web infrastructure.

- The HTTP protocol is inherently stateless, i.e., after a Web server sends a browser on a client machine a page, it ends communication with the client and retains no memory of the client in its RAM. Instead, memory of client state is saved using mechanisms such as "cookies" stored on the client, or as entries in a database associated with that client's IP address. This statelessness is what enables a Web server to serve a much larger number of "clients" than older approaches. If used to implement workflows, a large number of workflow instances can be "active" without continually consuming CPU resources when a given workflow moves into a "waiting" state.
- All of today's machines, from desktops to high-end servers, to even mobile devices, understand HTTP.
- The multi-tier client-server approach used by Web-database applications (where, at a minimum, a Web server intervenes between a client machine and a database) is also scalable. Communication between a Web server and a database is limited to the short time (i.e., fractions of a second) that it is actually necessary to perform a database-related action. This way, a limited number of physical connections to the database can be multiplexed to serve a much larger number of concurrent users. The Web server infrastructure that controls the user interface can be scaled up through "server farms", where additional physical servers can be recruited when concurrency loads are heavy.

It turns out that a relatively recent Web-related technology, Web services, which I'll now describe, is particularly suitable for workflow implementation involving multiple systems.

9.4.1

Web Service Basics

A software “service” is nothing more than an application that behaves like a subroutine in a traditional programming language, except that the subroutine can reside on a different CPU, across a network or the Internet. The subroutine is a black-box: that is, the caller only knows what it is intended to do, what to send it, and what to expect back. In a *service-oriented architecture* (SOA), much of an organization’s mission-critical software is implemented as a set of inter-communicating services working in coordination. An SOA is said to be “loosely coupled”: that is, the individual services may run on machines with different types of hardware and operating systems, and all these details are irrelevant to the overall system design, allowing one hardware unit to be replaced with any other, as long as the service provides the same functionality. As in the case of ordinary subroutines, one service can call another, and so forth to an arbitrary degree of complexity. Several services may also be bundled together to quickly create a new application when the need arises.

A Web service is a service that uses Web infrastructure (specifically HTTP) as the means of inter-machine communication, even though the CPUs may be adjacent to each other or reside within the same Intranet. The advantage of HTTP, when communicating with CPUs across organizations, is that it is easier to invoke a service across an organizational firewall: most firewalls are configured to allow HTTP messages to pass through.

The Web service is invoked through a URL-like mechanism (or an actual URL). Parameters are passed either as XML content or by mechanisms identical to those used for Web pages (GET and POST). With GET, the parameter or parameters are appended to the URL, while with POST (typically used for Web-based forms) the information is sent in the message body as a chunk of arbitrary-sized data consisting of delimited key-value pairs, (An example of GET is the URL <http://www.amazon.com/dp/0684833395>, which returns information on the Joseph Heller classic “Catch-22”: the number in the URL refers to the book’s ISBN.)

The service will send data back to the caller. In simple cases, any delimited format can be used: a popular format used for services that support AJAX technology is JSON (JavaScript Object Notation), which can be used directly by the code in a Web page without parsing. In complex cases, an XML-based format may be used.

Most software-development toolkits shield the developer from the details of worrying about the SOA plumbing, so that it is possible to focus on the problem to be solved. Accompanying Microsoft Visual Studio, for example, there is a tutorial that lets you create a simple but fully operational Web service (a Fahrenheit-to-Celsius converter) in a few minutes.

The advantages of an SOA are that it is possible to scale up the performance of a given software system by recruiting additional hardware (with the caveats described shortly). Also, if well designed, an individual Web service that fulfils a specific need can be utilized by a large number of applications. Amazon, for example, composes a Web page that provides details of a given item (e.g., a book) through multiple Web services: one that

provides publisher information, one that provides customer reviews, one that provides information on what purchasers of the item also purchased, and so on. Some of these Web services are accessible free of charge to software developers, while others have to be paid for on a per-use basis: the per-use charge may work out to a fraction of a cent, but thousands of service invocations may be made by software applications in a month, so that the overall charges are a source of revenue.

However, SOA is not a panacea. The skepticism about the hype surrounding SOA is reflected in articles such as Bell³ only partially tongue-in-cheek “DOA with SOA”. It does not substitute for sound software-engineering discipline or thorough knowledge of the problem to be solved.

9.4.2

Challenges in Implementing SOA

Some SOA advocates have claimed that an SOA is “simple” because it allows a complex problem to be decomposed into smaller and more manageable ones.⁴ The same argument could equally be applied, however, to programming subroutines in general, and such a claim is misleading, despite the name of one of the XML/HTTP based protocols for SOA (SOAP – which originally stood for Simple Object Access Protocol – but which is by no means simple). The fact that you *can* decompose a problem doesn’t guarantee that you will know the best way to do it. There are two types of challenges in devising an SOA: one is technical and the other is not.

9.4.2.1

Identifying and Implementing Reusable Functionality

If you are going to make something a service, the issue is to identify which other software applications—applications internal to your organization, or those of your business partners or customers – can make use of it. If external users of the service are envisaged, the question is whether they will be willing to pay for it on a per-use basis.

We introduced the idea of “loose coupling” earlier – the independence of an SOA with respect to hardware and operating systems. This leaves open the possibility of recycling legacy software applications by converting them into services, which conceptually seems to be as simple as eliminating the user interface and making sure that the code can accept parameters that are passed in programmatically. Since Web services talk only to other machines, not directly to users, this is not an issue: any user interface associated with the technology is strictly for software developers who are testing a service.

The challenge, however, is not primarily technological. In reworking existing functionality into a “service”, one often discovers that the functionality must be broadened (i.e., generalized) or otherwise dramatically modified from its original form, to do so. In other words, *the assumptions underlying the original software design may not be valid any more*. The creation of at least one successful Web service, involved changing the organization’s business model: the service provider had to change from a competitor of certain (much larger) companies to a partner.⁵

It is difficult to foresee all implications of providing existing capabilities as Web services. For example, unless issues related to software localization are fully addressed, an old application now fashioned into a Web service may continue to output English text that is meaningless to non-English speakers. For scientific applications, issues such as units (US/British vs. Metric) start to rear their heads. One well known failure of a NASA Mars mission (the Mars Climate Orbiter) occurred after the orbiter went off course, because the contractor designing the thruster controller assumed the units of force to be in Newton while the computer software at NASA that remotely controlled the orbiter assumed the units to be Pounds.⁶ While this fiasco did not involve Web services, the underlying issue – the risk of miscommunication – is the same.

Such reworking requires not only a thorough understanding of the knowledge domain for which the service is being developed, but also some degree of experimentation. For example, just because you think someone might pay for a particular service doesn't mean they will. Even Amazon, which has essentially bet its shirt on SOA technology and is hailed as an SOA success story, has had misfires: a Historical Sales service (a pay-per-use service that, given an ISBN or UPC, sent back information on Amazon's sales of that item for the last 3 years) has been discontinued for lack of customer demand.

9.4.2.2

Performance Issues

The rationale for creating a service that provides data that would not otherwise be available to a client is obvious. If, however, the service only performs pure computation, the amount of computation performed must not be too trivial. For example, creating a Fahrenheit-to-Celsius converter Web service is all right to learn how Web services work, but it is hardly the basis for a realistic implementation. The conversion involves two simple arithmetic operations (subtraction by 32 followed by a multiplication by 0.5555 or 5/9) that a modern mid-range personal computer can perform within a *nanosecond*. It does not make sense to create a separate service, to which the number to be converted must be sent over a network or Internet, with the result being sent back the same way – a process that takes tens of *microseconds* at best (if you have the latest and greatest network hardware and all of the services exist inside your organization), or several *milliseconds* (if one of the services resides outside the local-area-network, i.e., on the Internet).

If you take existing software and convert every subroutine in it into a Web service running on separate state-of-the-art hardware, the chances are that your application will perform at the level of a 1980s PC, because the multiple CPUs on which the services run will spend most of their time waiting to receive data and sending processed data back.

In other words, *a Web service, to be useful, must perform a non-trivial unit of computation* (which may involve operations such as database lookup) that is significant in proportion to the unavoidable inter-machine communication overhead. The ideal Web service is one that receives as input a relatively modest amount of data (e.g., an ISBN, as in the Amazon example) and does a lot of computation with it.

The sticking point is that determining what constitutes “a lot” requires that you actually benchmark something operational. Desktop quad-processor CPUs now run at 3 GHz or more: the latest version of software-development platforms such as Java and .NET provide

parallel-computation libraries that take advantage of multiple processors. On the other hand, while network speeds have reached 10 Gb/s with Ethernet, network latency (the initial delay preceding the transmission of data) is still relatively significant. The ping/pong latency (the time taken by a signal travel from one machine to another with a separate network address) and back is around 1–10 μ s on supercomputer-class machines with special interconnects⁷ and 100–300 μ s on ordinary server-class hardware with 10 GB Ethernet⁸ network cards and fiber-optic cabling. This means that a service must either do a lot of computation, or must itself perform significant local I/O – for example, accessing a component of a distributed database – so as to free the caller from having to perform such I/O itself.

Note: One justification for converting existing code into a Web service, made by David Chappell in a tutorial written for no other than Microsoft Corporation,⁹ is that the same functionality needs to be invoked by multiple types of applications—desktop clients as well as server-based applications. But again I would introduce a caveat. If the service consists only of a few lines of code, you again have to trade off performance vs. maintainability: replication of this code in more than one place does not necessarily constitute a maintenance nightmare, especially if the functionality is not likely to change. (For example, the formula for Fahrenheit-to-Celsius conversion will be valid for as long as these units continue to be used.)

I mention this because even today, there are medical informatics papers such as the one by Fehrea et al.,¹⁰ which talk about a Web service implementation of an Arden medical logic module that computes body-mass index (BMI). The single-line formula for BMI calculation ($= \text{Weight in Kg}/(\text{Height in meters})^2$) is no more complex than the temperature-units converter.

9.5

Languages for Workflow Specification

When implementing a given workflow in terms of an SOA, it is useful to be able to specify the overall design in terms of the following:

- The calling chain: which services invoke which other services.
- The data that is sent from one service to another, and the data that is sent back.
- How error conditions are handled. (For example, the called service may not be able to locate some requested data, or it may take so long that the caller may have no option but to give up.)
- Whether communication is synchronous or asynchronous. In asynchronous communication, the caller does not wait for the called service to finish and send something back, but does something else in the meantime. In the synchronous case, the caller waits – sometimes up to a certain time limit before giving up.

- The design may specify that certain services are to be run in parallel rather than sequentially, while others have to be called in sequence. (Amazon’s book information, for example, is generated by running the individual services in parallel, and then collating the individual results.)

A high-level “language” that allowed specification of such details would be a significant productivity aid. I’ll discuss some efforts here in detail.

9.5.1

Business Process Execution Language

Business Process Execution Language (BPEL) was the first language to support such a design. BPEL, also called WS-BPEL (WS=Web service) is based on XML. It originally evolved from disparate but related ideas by several IT vendors: the standards body now responsible for BPEL is the Organization for the Advancement of Structured Information Standards (OASIS). Several vendors now provide BPEL implementations, which work reasonably well. However, though BPEL is supposed to be a “standard”, it has the following problems:

- For non-trivial SOA designs, the XML that represents BPEL is not easy to either write or understand. Programming using XML dialects that contain a large number of tags has been likened to root canal surgery without anesthesia: in terms of productivity, it is just one level above assembler. Since the connection between systems is naturally visualized as a diagram, a graphical notation, analogous to that used by UML, is much more natural to use. However, OASIS decided that visual notation creation was beyond its purview. As a consequence, every BPEL vendor has devised their own (mutually incompatible) diagramming approach.
- The original BPEL specification was standardized prematurely. It turned out to be so weak in terms of what it could do that every BPEL vendor added proprietary extensions to get it to solve real problems. A new version of BPEL (version 2.0) fixed some of these issues, but in a way that was not fully backward-compatible: there was no clear-cut way of automatically porting an application from BPEL 1.x to 2.0.
- There are still several areas of business processes that BPEL 2.0 does not address. For example, every workflow involves humans: they may initiate certain services, they make decisions, and must also act as fallback in case certain computational processes fail (or if the workflow has changed due to external circumstances, but there’s not enough time for software developers to implement the modified workflow). Again, every vendor addresses this problem in their own way. IBM and SAP have proposed an extension to BPEL called BPEL4People that deals with the human aspect, but OASIS is, as of this writing, a long way off from standardizing it.

As a result of these issues, any customer who has chosen to move to the BPEL bandwagon is pretty much married to a particular vendor in perpetuity or until the vendor ceases to exist – the very situation that a standard is supposed to prevent.

9.5.2

Business Process Markup Notation

The failure of BPEL to support a standard diagramming notation is a serious concern in terms of reducing its usability. An alternative means of expressing workflow, which is intended to be used by high-level designers rather than hardcore programmers, and which therefore *has* a standardized visual representation that resembles the common flowchart, is called *Business Process Markup Notation* (BPMN).

BPMN is fairly easy to learn: even if you do not have the budget to invest in high-end BPMN packages, there are templates for BPMN in diagramming tools such as Microsoft Visio. However, there are significant differences in the semantic models underlying BPMN and BPEL. This is in part because the two were originally designed for slightly different purposes, and mainly because the two are maintained by different standards bodies. The BPMN specification is maintained by the Object Management Group (OMG), which does not appear to be on talking terms with OASIS. This situation in turn creates several problems:

- A picture is useful for understanding how something is intended to work, but ultimately it must be converted into a format that is more suitable for computation (like BPEL). BPMN itself has an XML equivalent (not intended for human consumption, but for software tools that need to operate on it) called XPDL (XML Process Definition Language). XPDL, however, is quite different in purpose from BPEL, and is incompatible with it in various ways. For example, XPDL has been devised with the intent of capturing and reproducing a BPMN diagram precisely, so that every object's information in the XML ends up containing X and Y coordinates, even though such information is less than relevant to a domain expert. BPEL could care less about these.
- The BPEL-BPMN incompatibility has led some vendors such as Microsoft (whose Biztalk Server product is a “universal” workflow designer aimed at integrating existing software) to ignore BPMN entirely, sticking with their existing diagramming approach that corresponds directly to BPEL. Microsoft, in fact, seems to be de-emphasizing BPEL itself in favor of their homegrown alternative, Extensible Application Markup Language (XAML). In effect, their attitude towards the two standards is much like Shakespeare's Mercutio in *Romeo and Juliet*: “A plague on both your houses”. While Microsoft is only one of several players in the workflow area, it is an important one because, as we'll see, it now offers a workflow toolset as part of the Windows operating system itself.
- Some vendors (notably Oracle) try to straddle the divide by using BPMN as the visual notation for BPEL. This attempt has not been entirely successful, since the two have different semantic models.
 - Certain advanced BPMN constructs cannot automatically be transformed into BPEL equivalents, so if you hope to generate BPEL, you have to use a significantly limited subset of BPMN capabilities. The software will not help you in this respect: it will merrily let you paint yourself into a corner.

Therefore, as a BPMN user hoping to create something operational (as opposed to a nice diagram for pedagogical purposes), you have to know what will not convert.

This knowledge requirement is paradoxical: BPMN, supposedly a tool for non-programmers, now requires significant knowledge of a programmer-oriented technology (BPEL) to use effectively. It follows that a specification that is already in BPEL cannot be automatically transformed to BPMN either for the purposes of visualization.

- Consequently, *round-trip software engineering* (the ability of a software tool to maintain its generated output synchronized with its higher-level model) becomes impossible. Remember that a software design is never static, and keeps evolving. If you use BPMN to generate BPEL, you have not built an operational architecture: significant tweaking of the BPEL output must be performed by the software developer. Once this tweaking is complete, the BPMN diagram does not reflect the true state of the system. Contrast this to software such as Microsoft Access’s Visual Query By Example, Visual Studio’s Web Developer or HTML editors such as Adobe Dreamweaver, where minor (or major) changes to the diagrammatic/visual representation results in appropriate changes to the code, *and vice versa*.

Round-trip engineering is a technically challenging problem to solve, but it cannot be ignored if a tool or software-development paradigm is to have any longevity. I emphasize this many of the clinical-guideline frameworks that were inspired by workflow languages failed to address it. Tools that do not support round-trip engineering end up as “shelf-ware” – i.e., products that sit on the shelf unused after an initial honeymoon period. At best, they only generate, in a one-time operation, a partial version of the final system. After being let down a couple of times by the inability to go back and forth between the model and the implementation, developers regard the tool as being a trap, and tend to avoid it altogether. Eventually, the model and the implementation cease to bear any resemblance to each other.

- The BPMN specification, like BPEL itself, does not support human processes: as for BPEL, every vendor has had to work around this by implementing their own extensions to BPEL for this purpose. Consequently, the “lock-in” problem manifests here as well.
- Finally, BPMN’s functionality overlaps with that of another OMG-supported standard that already has a much larger user base than BPMN will ever have. The Unified Modeling Language’s Activity Diagram (“Activity Diagram” is essentially a beefed-up flowchart) can do pretty much all that BPMN does with respect to workflow patterns, albeit less concisely, as shown in a head-to-head comparison by Stephen White.¹¹

The question is whether the modestly enhanced expressivity of BPMN over a small part of UML is a reason to use it. While White claims that BPMN was intended for non-programmers with domain expertise/business knowledge to use, while UML is intended for software developers, I find this argument unconvincing: high-school students can interpret flowcharts intuitively, and a UML Activity Diagram is not that much harder to learn to understand and to create.

For creating serious software that facilitates critical operations, tools that are modestly more difficult to use but capable of delivering necessary functionality eventually tend to be preferred over deliberately dumbed-down tools that have too many restrictions on what their use can achieve, and which insult their users’ intelligence. White states his belief that at some point, BPMN and UML Activity Diagrams may merge. If that happens, it is not hard to predict which standard will gobble up the other.

Also, for non-trivial designs a BPMN diagram can become complex enough that familiarity with programming concepts (such as branching based on decision points and loops) becomes necessary. As I've emphasized repeatedly in other contexts, at some point a user who desires to get things done has to graduate from amateur status. It is now that the limitations of a purely graphical approach to system design become apparent, as I'll now discuss.

9.5.3

BPEL and BPMN Limitations

The real problem with both BPMN and BPEL is that their robustness for complex SOA designs has not been seriously evaluated. Neal Ford¹² provides numerous reasons for skepticism. Pointing out that visual workflow development lacks several standard capabilities that traditional programmers have come to rely on, Ford refers to BPMN and BPEL as "doodeware" tools that "let you create pretty pictures that collapse under scale". To summarize his arguments:

- As the complexity of a system grows, the visual programming approach will lead to a "hairball effect", where numerous individual services are connected by lines that start running together, so that the overall design becomes hard to inspect visually.
- Looking at a diagram it becomes hard to identify workflow units that can be reused. Even when you do succeed in identifying such units, neither of these approaches supports true subroutine capability with parameter passing. Standard BPMN has a construct called a sub-process, but it has no concept of parameter-passing: sub-processes are little more than a means of minimizing visual clutter by allowing a collapse-and-expand capability in the diagram. (This incapability is seriously limiting, and hard to comprehend, considering that FORTRAN, the world's first high-level programming language, had already acquired full subroutine capability in the mid 1950s.) Commercial and academic implementations of BPMN may add this necessary capability, but in unique and of course non-standard ways.
- One is forced to program standard constructs like *if* statements and *for/while* loops by drawing flowchart-like pictures, so that professional programmers take several times as long to accomplish a given task.
- It becomes hard or impossible to write tests for the workflows using a visual paradigm. (Any design must be tested before deployment, and a suite of tests must be maintained and run periodically whenever changes are made to software, to ensure that bugs have not been introduced by the changes.)
- All production systems evolve after initial deployment, and it is important to know how different versions of the system differ from each other. For textual code, there are tools such as the venerable *diff*, which compares two text files and reports the differences, and which is the basis of version control. However, there are no tools currently that render a meaningful summary of differences between two images. (It is possible to compare two different streams of the XML counterpart of the images programmatically, but the differences must be transformed into an interpretation that is meaningful

in terms of the problem domain. Presentation of copious XML differences merely in terms of tags and attributes can trigger migraines in the beholder.)

- Ford reiterates the issue of proprietary features: BPEL/BPMN implementations may be “standards-based” but are not *standardized*. As with ANSI SQL, the vendors have a vested interest in maintaining customer lock-in and in ensuring that a weak standard will remain “toothless in perpetuity”.

Ford points out that BPMN and BPEL have been marketed to less technically-oriented managers who are more likely to be impressed with simple demonstrations of graphically constructed software, and who fail to ask the hard questions that technically skilled developers would.

9.5.3.1

The Limitations of a Pure Visual Programming Metaphor

Visual software development makes sense when one is creating user interfaces such as forms or Web pages, or when what is being designed is intrinsically visual in nature or maps naturally to a visual metaphor, such as architectural plans or database schemas. Here, it is natural to compose such interfaces by dragging and dropping objects from a toolbox on to a “canvas”, and specifying the properties of each object through a dialog. However, even “Visual” Basic, which popularized this approach for interface development, switches to traditional programming mode when the developer needs to specify what happens, say, when a button is clicked.

Representing services as nodes in a two-dimensional diagram, connecting them with lines, and then specifying the properties of a connection in terms of what is sent and received, may possibly make sense in terms of affording more productivity than enumerating pair-wise connections textually. However, workflow diagramming also includes more traditional flowchart elements such as branching and iteration, and in this respect, flowcharts have ceased to be considered a productivity aid since the early 1950s, when programmers worked in machine code. Even for high-level design or description of an algorithm, flowcharts are less succinct (and less clear) than “structured English” (the use of natural language augmented with structured-programming syntax), especially when supported by code-editors that assist indenting of branching and loop constructs.

Being forced to program conceptually trivial operations through diagrams when superior alternatives exist can be infuriating to any programmer whose skill exceeds novice level. (After going through the Microsoft tutorial on creating and running workflows using their toolkit, Windows Workflow Foundation¹³ – you are expected to create an application for a number-guessing game – the reaction I’ve encountered from several developers is whether it is worth spending so much energy to ultimately achieve so little.)

When attempted in software packages in the 1980s and later, flowchart-style programming has not survived for long. The Apple Macintosh database software Helix became a niche product after programmers developed a lasting allergy to its forced “visual programming” paradigm. Also on the Macintosh, the 4th Dimension database software originally offered visual programming as an alternative to a Pascal-like language, but eventually

abandoned it as the latter's feature set expanded: at this point, the visual paradigm became unwieldy to the point of non-usability.

It is by no means certain that BPMN and BPEL, which are slightly more than half a decade old, may meet a similar fate. What is certain is that, in their current incarnations, they have serious limitations that are likely to prove crippling over time if not corrected. More important, it is hard to see how these limitations can be fixed unless an alternative to the visual programming approach (but one that is still higher-level than working with reams of XML) is devised.

9.6

General-Purpose Workflow Toolkits

Providing workflow-related functionality is an important aspect of advanced application software development for organizational needs. Dedicated workflow management systems constitute a major investment that few have the luxury to make, and these are often hard to integrate with existing applications. However, code libraries that support workflow operations, whose routines can be called from one's own applications, offer a potentially attractive means of improving developer productivity.

The movement to offer such support parallels the provision of lightweight RDBMSs (such as Microsoft Access and SQL Server Express Edition, Borland Paradox, Sybase SQL Anywhere) as a standard component of developers' toolsets from the mid 1990s. This resulted in better-quality software that used a significantly more robust means of data storage, as opposed to the previously prevailing approach of devising one's own storage mechanism from first principles. Concurrently there was a jump in SQL literacy as programmers flocked to take advantage of the added capabilities. If nothing else, the availability of low-cost workflow toolkits may be expected to have a similar effect on workflow-concept literacy.

There are numerous Java-based workflow code frameworks (several of them open-source freeware), that offer a version of BPEL/BPM, or adopt a different approach. (A summary of these, with no comparative evaluations of their respective merits or weaknesses, is listed at ¹⁴) Since the last 4 years, Microsoft has joined the fray by making their toolkit – Windows Workflow Foundation (WF) – part of the Windows infrastructure (part of the Microsoft.NET framework). Given the current dominance of the Windows platform for desktop computing, the availability of WF, and the fact that it is free (as long as you own a Windows license), Microsoft promises to change the rules of the workflow-software business, in making the technology accessible to the individual developer and small business.

9.6.1

WF Essentials

WF uses a combination of Microsoft's proprietary XML-based workflow language (XAML, mentioned earlier) and an extensive code library. While one can wrestle with

XAML in a text editor, Microsoft allows XAML creation and editing using a diagramming approach with Visual Studio, by dragging and dropping icons from a palette onto a diagram, and then specifying the properties of each icon.

If you choose to work with diagrams alone, you are essentially programming branching, loops, etc. using a flowchart metaphor. Some of the individual steps in the flowchart (which correspond to statements in a programming language) involve the use of expressions, such as Boolean conditions or arithmetic formulas, and these must be expressed in a particular syntax. The expression language that WF uses currently is Visual Basic.

As explained later, it is possible to create custom workflow modules using a .NET programming language. Software developers who are driven to distraction by the visual programming metaphor may maximize the use of the custom-module approach.

The current version of WF (4.0), released in Spring 2010, is a complete overhaul, to the extent that it is incompatible with previous versions – which are still supported, but whose use is discouraged for new projects. Several aspects of the previous versions – for example, a forward-chaining rule engine – have been mysteriously dropped without explanation in the current version. (To be fair, the engine was both underpowered and difficult to use: Microsoft lacked a clear vision of who its intended users were. Rather than simply migrate an underwhelming version of the software, it makes sense to first step back and evaluate past efforts critically.)

In effect, Microsoft implicitly acknowledged that the previous versions were misfires. While this about-face has made many early-adopter developers unhappy – books on WF written as recently as 2009 are now obsolete – the overhaul was apparently based on external and internal feedback from developers who had pushed WF to its limits. The author's own experience with WF4 is that, while it is a significant improvement over previous versions, it still has a long way to go before it can attract the second wave of non-adventurous developers. Integrating WF with existing applications containing a significant amount of code still takes far more effort than it should.

Given the history of BPEL and BPMN presented earlier, such changes seem par for the course. The fact that such major redesign efforts continue to occur, however, indicate that workflow-software technologies are less mature than their chronological age would indicate. This is what happens when the price of admission is too high: there is a lack of a critical mass of developers whose efforts can serve as a crucible for individual ideas.

9.7 Caveats in Applying Workflow Technologies

9.7.1 Flexibility of Workflow (Re) Design

Compared to traditional programming approaches, the flexibility of the workflow approach stems from the fact that the output of a workflow-design process, where the workflow is designed graphically, is typically intended to be *interpreted* by a workflow-runtime engine, rather than compiled to machine code. In other words, like the rules we discussed in the

previous chapter, workflows can also be treated by the workflow engine as *data*. The rationale is that, if a workflow management system comes with an intuitive workflow-authoring tool, then workflow processes can be authored (possibly by non-programmers) rapidly: consequently, the turnaround time for electronic implementation of workflows is shortened because it eliminates the step of code writing. Therefore one can respond to changed external circumstances more rapidly.

Again, as with rule-based approaches, this argument can only be taken so far. Such flexibility, as with rules, can be a double-edged sword: we now describe some circumstances where such flexibility must be sacrificed.

9.7.1.1

Workflow Software Is Not Programming-Free

The requirement that all aspects of workflow-design software be operable by non-programmers becomes needlessly constraining when a process is complex. At this point non-programmers have to start thinking like programmers, which is difficult when one hasn't been trained to do so: if the workflows to be designed are mission-critical, programming ignorance becomes an unaffordable luxury. Therefore extensibility through traditional programming techniques becomes desirable or even essential. Implementing extensibility requires some forethought and planning, and a problem-solving approach quite different from that of a "rapid reflex reaction" mind-set.

- Workflow software supports the creation of reusable units (workflow modules) from built-in primitives graphically. Essentially, you create a workflow by drawing it, but after testing it, you "compile" it with the vendor's software and give it a name indicating its purpose. You can then treat this workflow as a black-box, and make it available for others to reuse.
- If enough modules are created this way, you have designed a toolbox of reusable workflow components that reflect your knowledge domain. For example, one common workflow component in a hospital would be ordering a radiology investigation. However, identifying truly *reusable* functionality that can be encapsulated this way, and that will apply to multiple higher-level workflows, requires one to think like a software developer. Thus, the module must be *parameterized*: for example, rather than a special workflow for ordering a chest X-Ray, the type of radiology investigation would be a parameter.
- With many workflow-software frameworks, individual, reusable building-blocks in a workflow may be developed with traditional programming languages, to yield compiled code units that can then be reused as above. WF, for example, supports such module creation using a .NET programming language. Such modules can then be added to the workflow-design palette.
- Among the numerous frameworks that exist today, different approaches only differ with respect to how *much* programming is needed. The set of workflow-related operations that a given vendor's framework provides as primitives (i.e., icons that can be dragged and dropped into a workflow from a palette/toolbox) varies. For example, in the high-end

and pricey Microsoft Biztalk Server, one of the primitives is sending an E-mail: in the free Workflow Foundation, this is unavailable, and must be programmed.

- The more sophisticated and numerous the pre-fabricated components that are available for use, the less the effort required by the customer in developing one's own workflows. One reason why workflow software for specialized domains, such as document management and laboratory automation for bioinformatics and chemistry-informatics applications, is more likely to meet its customers' needs than "universal" workflow-management software is that the tasks associated with the domain are better defined. Therefore it is possible for a vendor to create sophisticated components for such tasks in advance.

9.7.1.2

Performance Issues

Interpreted code runs significantly slower than compiled code. If you want very fast hardware response under heavy concurrent usage (as for a busy E-commerce site receiving online orders) relying on a fully-interpreted approach is likely to be a recipe for corporate suicide. Despite the use of multiple hardware units, performance may be unacceptable: previous versions of WF were essentially unusable in high-concurrency scenarios requiring rapid response. Despite Microsoft claims of high scalability, WF doesn't do a good job of taking advantage of parallel-hardware capabilities.

Consequently, a software developer is better off identifying reusable functionality that is stable (i.e., is unlikely to change) and implementing them with traditional programming approaches, creating compiled workflow units. A software team at Merrill Lynch¹⁵ took this approach to an extreme degree, creating compiled custom building blocks (that performed actions related to financial investment), which took advantage of multi-processor CPUs. These blocks became part of a toolset that enabled non-programmers to create workflows using WF only as the scaffolding.

9.7.1.3

Flexible Technology does not Solve All Problems

Even when real-time performance is not an issue, the mere fact that a component of the delivery technology is highly flexible in theory (and in the hands of a suitably knowledgeable person) doesn't guarantee the adaptability of the overall solution. Multi-million-dollar Enterprise Resource Planning (ERP) packages such as SAP, which are intended to link all of an organization's operations together, incorporate workflow concepts. However, ERP packages' reputation for non-adaptability has been notorious. Even now, changing the way an organization operates – itself a disruptive and traumatizing course of action whenever it is attempted – is often easier than to try to change the behavior of the software.

The problem is only partly technological. If you have a workflow that is itself being used as a building block in several dozen other higher-level workflows, it is not safe for a customer who has just purchased the package to jump in and start modifying it. Such

changes may cause the dependent workflows, which expect a particular kind of behavior from this component, to fail.

Such behavior is also seen with business rules: they are so easy to modify that, if you have a system with thousands of rules, they also provide enough rope to hang yourself with: the overall system starts to misbehave strangely, and you tear your hair in trying to find out which changed rules were responsible. Also, rules are so easy to add that, unless your software has high-quality tools for querying and reporting on the rules database *and* you use these tools diligently, you are quite likely to add rules that are so similar to existing ones, as to be redundant.

In other words, as in any knowledge domain that becomes suitably complex, a little knowledge becomes a dangerous thing. The fact that a given technology simplifies the technological aspect of making changes is immaterial: if you don't know how to change something safely, you can't risk changing it at all. ERP software has, understandably, led to a significant boost in gainful employment for ERP consultants.

If the software cannot be modified safely *and* existing processes also turn out to be too ingrained to modify, one has the recipe for yet another spectacular information-technology flameout, which has happened so often with ERP that typing "ERP implementation" in Google will cause the suggestion of the next word "failure".

9.8

Conclusions

While the promise of Next Big Things to solve the software backlog has a long history (workflow engines' immediate predecessors in this regard were rule engines) one always seems to need real programmers when problems get hard enough. Tools that fail to address the needs of complex problems and the needs of the people who make a living solving them soon become delegated to toy status.

Chief among these needs is the ability to integrate with existing external software, particularly databases, both for access to data as well as persistence. Rule engines that have thrived (such as the open-source project Drools) cater to both the programmer and the end-user. Many workflow frameworks in fact incorporate rule engines, to dissuade developers from cobbling together their own solution to workflow. Such frameworks must meet three criteria:

1. The rule engine must be best-of-breed-quality.
2. Software integration must be relatively straightforward.
3. The framework's purchase must not require a multi-million-dollar software budget.

While the basic concepts of workflow remain valid and useful, and Web services are here to stay, current frameworks seem to meet at best only two of the three above criteria: WF, which is indeed affordable, meets only one. For all its current limitations, however, WF seems to have pointed the way for the future evolution of workflow software. If nothing else, WF, like many earlier Microsoft forays into high-priced niche markets (such as the

Business Intelligence and Web Portal areas), will cause turmoil in the workflow arena and cause prices to drop. If robust open-source alternatives eventually develop in the same direction, they will in turn put pressure on Microsoft and other commercial vendors to improve their own offerings.

References

1. Sailors RM, Bradshaw RL, East TD. Moving arden syntax outside of the (Alert) box: a paradigm for supporting multi-step clinical protocols. AMIA Fall symposium, 1998.
2. Taylor FW. Principles of Scientific Management. [cited 11/1/10]; Available from: <http://www.marxists.org/reference/subject/economics/taylor/index.htm>, 1911.
3. Bell A. DOA with SOA. *ACM Queue*. 2007;5(1):54-56
4. Kawamoto K, Lobach D. Proposal for fulfilling strategic objectives of the U.S. roadmap for National action on decision support through a service-oriented architecture leveraging HL7 services. *J Am Med Inform Assoc*. 2007;14:146-155.
5. Lindquist C. A new blueprint for IT (editorial). *CIO Magazine*. 2005.
6. Wikipedia. Mars Climate Orbiter. [cited 4/10/10]; Available from: en.wikipedia.org/wiki/Mars_Climate_Orbiter, 2010.
7. University of Tennessee Knoxville. HPC Challenge. Latency and Bandwidth results. 2010 [cited 11/2/10]; Available from: http://icl.cs.utk.edu/hpcc/hpcc_results_lat_band.cgi?display=opt, June 2010.
8. Burton K. Ethernet Latency: the hidden performance killer.[cited 10/1/10]; Available from: <http://feedblog.org/2006/11/26/ethernet-latency-the-hidden-performance-killer/>, 2006.
9. Chappell D. The Workflow Way: Understanding Windows Workflow Foundation [cited; Available from: <http://msdn.microsoft.com/en-us/library/dd851337.aspx>, 2009.
10. Fehrea K, Mandl H, Adlassnig KP. *Service-oriented, arden-syntax-based clinical decision support MEDINFO*. Cape Town: IMIA; 2010.
11. White SA. Process Modeling Notations and Workflow Patterns.[cited 10/1/10]; Available from: http://www.bpmn.org/Documents/Notations_and_Workflow_Patterns.pdf, 2006.
12. Ford N. SOA and the Tarpit of Irrelevancy.[cited 10/2/10]; Available from: www.infoq.com/news/2009/02/TarpitofIrrelevancy/, 2006.
13. Microsoft Corporation. How to Create a Workflow.[cited 8/8/10]; Available from: <http://msdn.microsoft.com/en-us/library/dd489437.aspx>, 2008.
14. Manageability.com. Open-source workflow engines written in Java.[cited 9/2/10]; Available from: http://www.manageability.org/blog/stuff/workflow_in_java, 2009.
15. Vadaparty K. Multithreaded parallelism in windows workflow foundation. *MSDN Magazine*. 2008.

10.1

Introduction

In the last three chapters, we discussed some of the issues that make the field of decision support challenging, as well as the theoretical foundations of several widely used approaches to decision support. Business workflow technology has been emphasized because it is much more mature (if pricey and proprietary), as a basis for decision support, than similar approaches that are now being explored in the medical area. While practitioners of medicine have long claimed that the problems of medicine are unique, the assertion of exceptionalism can be made truthfully for almost any complex domain, whether it involves the construction of aircraft or the operations of a multi-national investment firm. Proper operations of business workflows that involve large financial transactions is every bit as important to the organization's well-being as the proper management of patients in a healthcare organization. My belief is that reusing and adapting ideas that have worked elsewhere is likely to be more productive in the long term than trying to rediscover first principles.

In this chapter, I will take a look at some of the approaches that have worked in the medical domain, or which are highly promising. I will also discuss some false starts, and the possible reasons for their lack of success.

10.2

Table Driven Applications

In the first chapter, I introduced metadata-driven software as a special subcategory of table-driven application. In software development, this class of application is so important that I'll reintroduce Steve McConnell's dictum over here: if you have a problem characterized by what appears to be highly convoluted logic, consider if you can use a lookup-table-based approach to simplify it. Decision tables, which we introduced in the chapter on rule-engines, are a means of such simplification that are applicable in certain situations.

The major advantage of a table-driven approach is that if a problem can be solved this way, you have to write less of it and change it less frequently. However, you have to abstract

the solution to a greater degree: your code becomes more generic. In circumstances where your application is expected to have a significant life-time, the extra effort involved in abstracting the problem is well worth it.

10.2.1

Example: Drug Interactions

Consider the problem of representing knowledge about drug interactions, where a drug can interact with another drug, a laboratory test, or certain types of food. One could create a rule for each drug interaction, of the form:

If the patient is taking drug-X AND the patient is taking drug-Y then....

With a modest-sized dataset (e.g., a couple of thousand rules), and the use of a forward-chaining rule engine that utilized the memory-intensive Rete algorithm, it is even conceivable that this approach might actually work. However, the total number of documented drug interactions is now in the 100,000 range, which is a bit too large for a rule-based approach. The use of one or more relational tables to capture interaction, however, represents the information very elegantly

For example, the commercial drug databases typically represent drug-drug interactions in a table thus: Drug-1, Drug-2, a textual Interaction Description, mechanism of interaction (if known), one or more qualitative measures of the importance of the interaction (severity, frequency), and one or more bibliographic citations. In a normalized design, each drug would be represented by an ID, and there would be another table recording the drug's generic name as well as brand names, allowing search by either. Similarly, the measures and mechanism would typically be of the enumerated data type, with the explanation of each coded value being stored in separate tables. Since the same citation may apply to multiple interactions, one stores citations in a separate table and typically records only a citation ID against the interaction.

In some systems, even the Interaction Description would be encoded as a number, which would be used to look up a Descriptions table: this is because drugs belong to pharmacological families, and if a member of one family interacts with another family through a common mechanism, the explanatory text can be recorded just once instead of being repeated.

To economize on space (at the cost of speed) some designs store an interaction for a pair of drugs so that the ID of the first drug is always less than that of the second. The algorithm that determines all possible interactions, given a set of medications that the patient is receiving, is as follows:

- For each medication (supplied as either the generic or brand name) determine its constituent drugs.
- Get the ID of each drug.
- Sort the IDs in ascending order.
- Generate all possible pairs of IDs such that the first has a lower ID than the second. If there are N drugs, there will be $N * (N-1)/2$ such pairs.
- Search the database for each pair. If a match is found, store the details of the interaction.
- Optionally, before presenting the details to the user, sort the interactions list in descending order of importance.

Other storage designs store the same pair twice, with Drug-1 and Drug-2 being swapped the second time. This design allows more rapid answering of queries of the type: “list all the interactions of Drug X” (irrespective of whether it is recorded as Drug 1 or Drug 2). The Unified Medical Language System, discussed in the chapter on Controlled vocabularies, uses a similar data-duplication approach for storage of relationships between biomedical-concept pairs

Conceptually, each row in the interactions table is the equivalent of a single rule. The difference is that one is taking advantage of the data’s homogeneity to create a tabular representation of the data. From the perspective of the content creator/vendor, such homogeneity offers numerous advantages. It allows ready implementation of a data-entry interface (with pull-downs for the enumerated fields), as well as robust consistency checks (such as the prevention of duplicate information for the same pair of drugs). Of course, in a relational design, all critical columns can be indexed for rapid access. RDBMSs are also efficient of computer resources, with caching of table contents in RAM: the degree of caching of a given table is typically under the developer’s or database administrator’s control.

10.2.2

Table-Driven Consultation Systems

Quick Medical Reference (QMR),¹ subsequently renamed Caduceus is an “expert system” that provided diagnostic assistance in the domain of internal medicine. Initially developed by Randolph Miller, Fred Masarie and Jack Myers, it is the most notable example of a table-driven system. QMR, implemented in Turbo Pascal, represented a re-engineering of a system, INTERNIST, originally written in LISP. A significant chunk of the original decision logic of INTERNIST was implemented in QMR using bitmaps, which we have briefly described in the context of indexing of database content in an earlier chapter.

QMR contains information about findings (which may be related to the history, physical exam, or laboratory tests) and disease conditions. It is well known that, while the presence of a certain finding may suggest a certain condition, not all cases of the finding will have the condition – other conditions may also be suggested. (The relative likelihood of individual conditions depends on numerous factors, such as the local prevalence of the condition and the patient’s demographic parameters – age, sex, ethnicity, socio-economic status and so on.) Similarly, not all patients with the condition will manifest the finding.

QMR uses ordinal measures (on a 0–5 scale) that are roughly analogous to sensitivity and specificity, that measure the strength of the association between the presence finding and the likelihood of the particular condition (specificity), and also between the association strength between the existence of a disease and the likelihood of observing the finding (sensitivity). In QMR’s measure, 0 means no association, 5 means maximal specificity (or sensitivity).

The diagnostic workup is not always a straightforward process that leads to identification of a single diagnosis: in a complex case (the kind for which QMR is useful) one may arrive at several diagnoses, ranked in descending order of probability. Since several thousand clinical findings and disease conditions exist across all of medicine, one must seek to

optimize performance by being able to rapidly reject the vast majority of inapplicable conditions, before combining up the ordinal metrics to arrive at an overall quasi-probabilistic measure for each candidate diagnosis.

QMR maintains a bitmap against each finding, one bit for each disease condition: 0 means absent, 1 means present. A similar inverse bitmap is maintained for each disease, with one bit for each finding. Given a set of findings supplied by the physician, QMR can rapidly compute the universe of possible diagnoses by combining the finding-specific bitmaps with an AND operation (so that only conditions that have all of the findings present, at any frequency, are identified).

The reverse bitmap is used as an aid to differential diagnoses: given two candidate diseases, one can combine the two disease bitmaps using an XOR operation to identify the bits (findings) that are *different* between the two (i.e., present in the first but absent in the second, and vice versa). If one wants to ask what is present in condition A that are absent in condition B, one can use combine the disease bitmaps using the operation (A AND NOT B).

The bitmaps are stored very compactly. The quasi-probabilistic 1–5 measures associating finding with disease and vice versa are also maintained as tables. (Even at the time QMR was developed – in the 1980s – much of the table content could be cached in memory.) In any case, the performance improvement was dramatic: Miller points out that QMR could do in *seconds* on an IBM PC AT what INTERNIST needed several *minutes* to do on a mini-computer.

Afterword: While tests of QMR revealed that it performed at the level of a senior specialist, commercialization did not succeed. Among other things, the challenges of maintaining and updating the knowledge base – as well as unexplored issues such as adjusting the sensitivity/specificity measures to account for local variations in disease frequency – were formidable.

The issue of local variation turns out to be critical. Shortly after immigrating to the USA from India, I remember playing with a diagnostic system, ILIAD,² developed by Homer Warner's group at Salt Lake City. I entered a set of findings that in India, or most of Asia or Africa, would be considered a fairly classical presentation of pulmonary tuberculosis. ILIAD, however, insisted that the diagnosis was coccidioidomycosis (a pulmonary fungal infection common in the Southwestern USA.): tuberculosis was not even listed in the differential.

There was also the significant issue of legal liability. QMR, like most consultation software, had to be marketed only as an aid to diagnosis and a teaching tool, but the ultimate responsibility had to lie with the physician who used it. Studies conducted by Miller's group showed that the utility of QMR was influenced greatly by the user/clinician's level of medical knowledge as well as proficiency in using the software: in this respect, like a lot of potentially powerful software, QMR was analogous to a loaded gun, something that could not be entrusted safely to the naïve or unskilled.

10.3

Complex Decision Support: Guideline Representation Languages

Clinical-Guideline representation languages were an active area of medical informatics research between 1998 through 2006. These were influenced by workflows, discussed in the previous chapter, which had just become a hot topic in software engineering. The idea was that a complex clinical guideline involves a patient with a broad category of complaint (e.g., chronic headaches or excessive menstrual bleeding) going through a workup involving multiple steps and decision points: for example, certain investigations are ordered only if there are suggestive findings in the history or physical examination. The steps in the workup are often separated in time, – for example there is a delay between the ordering of an investigation, its scheduling and the obtaining of the results. Multiple individuals are often involved – e.g., specialist primary care physician, pathologist, radiologist, or lab staff.

The majority of the languages used an XML-based syntax: some allowed the XML to be composed graphically using a diagramming editor. A survey of some of the leading guideline-language contenders (in terms of the feature set supported by each) can be found in the article of Peleg et al.³ As described below, these efforts did not meet with significant success – in particular, no operational system that interfaced to an actual EMR, and supported management of real patients with non-trivial guidelines was ever built. Therefore I will rarely identify individual initiatives by name, and I won't provide a detailed description of their approaches.

10.3.1

Reasons for Lack of Success

I'll now present an analysis of the reason why advances in this area have been disappointing. In essence, the difficulty of the problem was greatly underestimated, the approach that was chosen was too risky and immature at the time the efforts commenced, and the formidable challenges of building the necessary infrastructure for a guideline language to interoperate with electronic hospital systems (not just EMRs, but billing and scheduling systems) were never addressed. The infrastructure-creation challenges, in fact, dwarf those of designing a guideline language.

10.3.1.1

Imperfect Fit of Workflow Technology to the Guideline Problem

The management of patients for complex clinical guidelines resembles a workflow, and may even involve a workflow *process*, but one or more of the prerequisites for use of a workflow engine may sometimes be missing.

- In many complex guidelines, the guideline merely represents a flowchart-style decision process that guides the clinician's thinking, but aspects such as significant temporal

separation between steps and the need to save state at any particular step are not present. In such cases workflow technology is gross overkill: the conceptual flowchart could be implemented readily using a traditional programming language.

- As stated earlier, one justification for a workflow-based approach to solving a problem is when the problem is likely to keep changing frequently because of external circumstances beyond an organization's control. (If it changes continually because internal decision-makers change their minds capriciously, this problem is not curable through software.)

While clinical guidelines change over time (because new medications, investigations and treatments are introduced), the turnover rate is nothing compared to that in business processes. Published guidelines do not change more often than once a year in the worst case, barring infrequent eventualities such as an abrupt withdrawal of a drug by the FDA: it takes at least that long to convene an expert committee (and schedule to every committee member's convenience) to decide what changes should be made to the existing guideline. So the benefits of an interpretation-based workflow engine that allows rapid adaptation to external circumstances are modest, to the extent that they can be justified at all.

This is not to say that guidelines *never* fit the workflow paradigm: individual guidelines – particularly those that involve a combination of clinical and non-clinical processes – may, but for many others the workflow framework would only constitute needless overhead that gets in the way of creating a working implementation.

10.3.1.2

Limits to the Doctor/Amateur-as-Programmer Metaphor: The Need to Provide Productive Authoring Environments

Medical informaticians are not any more comfortable than the typical programmer when inspecting reams of XML, and so some kind of diagramming approach becomes necessary for guideline authoring if a workflow approach is used. Here, the guideline efforts have gone the same route as BPEL: while some efforts (e.g., the Guideline Interchange Format (GLIF) effort⁴) resulted in creation of a diagramming tool, a diagramming notation is *not part of the standard definition* of any language: only the XML is. Consider whether UML could have achieved its current ubiquity if XML syntax was the only way for a developer to express the semantics of class or activity diagrams.

Further, I've already stated in the previous chapter that a diagramming notation, while desirable, is only a partial solution to the problem of enabling a guideline author to address complexity. The needs of professional developers or advanced authors are quite different from those of the beginner or casual user. For complex problems, algorithm development by diagramming alone can by no means be considered a productive means of producing software. At some point, a coding environment that supports round-trip engineering becomes necessary.

Next, professionals have a large set of tools, such as code libraries: making them reinvent a large number of existing wheels to use a new language is not a good use of their

time. Professional developers must be able to *reuse* existing code. A language must therefore be *extensible*, allowing mixing and matching of traditional programming technologies with diagramming approaches. Extensibility is somewhat difficult to achieve technically, but the problem is by no means formidable: lots of environments do this.

Finally, the extensibility should translate into an *improved design-time experience*. Thus, the routines in one or more referenced libraries must be searchable and accessible to the author through mechanisms such as auto-completion. Most development toolkits such as Microsoft Visual Studio and Eclipse, for example, provide auto-completion (IntelliSense®, in Microsoft lingo), which provides context-sensitive help as a developer types ahead.⁵ Of course, robust design-time error-checking, both of the constructs intrinsic to a given language, as well as external library function usage, is also essential.

While creation of tools might seem beyond the scope of a guideline-language implementer, the concerns of extensibility and reuse are not. And as far as design-time experience goes, Microsoft's Windows Workflow Foundation would have been accessible only to handful of hardy developers without design-time tools in Visual Studio. If you want people to use your creation, you must make it easy for them to do so.

10.3.1.3

Immaturity or Absence of Existing General-Purpose Workflow-Language Standards

When it started, the guideline-language movement had the misfortune to catch the workflow wave at a time when it had just begun its rise – in other words, the technology was at the hype stage. Standard languages for workflow representation did not yet exist: BPEL and BPMN, discussed in the previous chapter, came much later. Consequently there was no body of experience that could be built upon regarding the adequacy or completeness of a proposed set of workflow-language primitives.

I've already mentioned the fits-and-starts evolution of these standards, and the fact that proprietary extensions are required to compensate for their weakness in multiple areas. This has happened *despite* the standards being developed by a consortium of profitable organizations that have been actually implementing workflow software and selling it commercially (as opposed to a handful of academic researchers in a few informatics labs).

Even as of 2010, the workflow-standards committees have yet to get it right, and concerns have been expressed by several commentators (such as Neal Ford, quoted earlier) that the standards will meet the similar fate of ISO SQL, and remain weak and underspecified. In fairness, designing a standard through committee resembles making sausages and legislation. When a proposed means of extending the standard using specific syntax is perceived to favor one of the participants even slightly, the others object vigorously, and the only means to progress is to horse-trade, or simply drop the proposed extension. It is not hard to see why Microsoft decided to go its own way in Workflow Foundation with XAML.

In any case, the guideline-language movement has been among the numerous collateral-damage casualties of the workflow-standards chaos. The risk of bleeding-edge technology is that it will usually change right under you, forcing multiple and expensive mid-course corrections to your architecture.

10.3.1.4

Insufficient Benefits to the Medical-Software Developer

Even for those cases where workflow technologies would apply to guidelines, one question that the “executable guideline languages” haven’t satisfactorily addressed is why one should use any of them instead of a standards-based (albeit proprietary-flavored), commercially-supported, general-purpose workflow framework (BPEL/BPMN or even Windows Workflow Foundation).

Several of the guideline languages did not provide enough added built-in functionality with respect to the medical domain to provide any specific benefits in guideline modeling. The paper of Peleg et al. cited earlier simply addresses their feature set from the perspective of general workflow features, which today’s general-purpose workflow languages (admittedly of more recent vintage) already provide.

It is true that medical-guideline languages differentiate themselves modestly from the general-purpose workflow languages by including some tags for concepts relevant to the guideline domain. For example, the Guideline Interchange Format (GLIF) specified tags for the guideline author, intention of the guideline, eligibility criteria, bibliographic citations, and description. However, all of this information is simply descriptive metadata, playing a role not much greater than that of comments in a computer program. While clearly useful to the *human* user who wants to know about the guideline’s origins and rationale, it is irrelevant to an engine that executes the guideline in a production environment: from the engine’s perspective, it does not matter if these tags enclose gibberish.

Related to this issue, the guideline-language efforts largely ignored the problem of communicating with the EMR bi-directionally, a capability essential in a system intended to deal with patients. (One exception was the SAGE effort,^{6,7} but, as discussed shortly, even here the exploration of the issues was largely theoretical.) It is highly disputable whether the guideline language *itself* should do so. However, any language requires a supporting runtime framework to operate in a production-EMR environment, in the same way that Arden requires an event monitor.

Aside: If the guideline content is to be portable across systems, it must be shielded from EMR-variation vagaries. That is, the guideline’s variable-mapping metadata must be physically separate from the guideline’s algorithmic content: the mistake of Arden syntax in allowing such information to be sprinkled all over the code of the algorithm within curly braces must not be repeated.

Given the sheer variation among EMR schemas, one possible way to address the issue of EMR communication would be to associate the guideline with metadata that maps individual clinical variables to elements in standard controlled vocabularies. The runtime-framework that dealt with EMR access would then operate on this metadata. The advantage of separating the metadata from the guideline algorithm proper is that a guideline's author needs to focus on expressing a published guideline's algorithm correctly. Such an individual may not have the knowledge (or the inclination) to address the task of mapping the variables in the guideline to controlled-vocabulary elements, and thence to those data elements that actually exist in a given EMR. The latter task is best accomplished by a controlled-vocabulary-content expert who also knows the EMR's data dictionary.

The virtual-medical-record efforts of HL7 were partially triggered by the needs of complex decision support. As I've stated, currently these efforts are still immature and evolving. In any case, if any software-development team succeeded in the hard part of integrating a general-purpose workflow engine with an operational EMR, handling the additional task of variable-mapping-metadata would be trivial by comparison.

In summary, isolated language-syntax efforts that downplay the hard problems involved in creating such a framework are unlikely to go anywhere.

10.3.1.5

Technical Challenges of Building an Operational Runtime Engine

While the paper of Peleg et al. describes all of the efforts as supporting "computer-interpretable" guidelines, this phrase can mean a lot of different things. The mere availability of a syntax that supports a variety of operations and the demonstration that a guideline written using this syntax can be interpreted by a simulator means very little. Building an operational workflow engine is a very challenging software-engineering task. Even a behemoth like Microsoft, after three tries at a general-purpose workflow engine, still hasn't pulled it off completely: as I've emphasized earlier, the current generation of Windows Workflow Foundation has more than its fair share of rough edges.

Even an academic-industry partnership provides no guarantee that the task can be accomplished. For example, in the context of the SAGE guideline project, a team of researchers from Stanford and the EMR vendor IDX (acquired by General Electric since 2006) attempted to demonstrate the operation of a prototype guideline execution engine, as reported by Ram et al. 2004.⁸ However, they used Microsoft Access to simulate the database rather than work with a test version of IDX, which they should have been able to set up readily in a sandbox environment. Given Microsoft's well-known and deliberate hobbling of the Access database engine with respect to performance and multi-user capabilities – Access is intended only for "personal" databases or as a front-end GUI development tool for two-tier database applications using a back-end such as MS SQL

Server – this simulation was not likely to be very persuasive as a demonstration of an architecture which, by definition, has to be highly scalable.

If a guideline framework ignores operational-workflow-engine issues, developers confronted with implementation decisions will simply choose to work directly with their EMR vendor's API: an inelegant but working solution is preferable to a theoretically elegant one that doesn't run.

10.3.2

Expression-Language Standardization Efforts

HL7 has been involved in guideline standardization efforts. While no standard guideline language has been adopted, in 2005 HL7 approved **GELLO (Guideline Element Language, Object-oriented)**⁹ as the expression language for clinical guidelines. GELLO is a by-product of the GLIF effort, and is inspired by Object Constraint Language (OCL),¹⁰ the expression language that accompanies the Unified Modeling Language.

My own opinion is that the GELLO standardization effort was premature.

- In contrast to Visual Basic, which Microsoft uses as the expression language for Windows Workflow Foundation (or, for that matter, Java), GELLO like OCL itself, is not computationally complete. For example, there is no subroutine capability, and the standard function library is very rudimentary. Like OCL, GELLO lacks built-in support for the date-time data type, which is problematic for any general-purpose application, let alone guidelines.

OCL was not designed as a standalone language, but was intended to integrate with the host modeling language (in this case UML). Similarly, GELLO is intended to operate within a predefined environment that will contain a set of predefined classes (and their methods/functions) which can be used by the developer. The specifications of the GELLO host have not been defined, however: as in the case of UML (which dwarfs OCL in terms of complexity), such a specification is likely to become complex.

Given GELLO's significant limitations, all that the developer can do, for now, is hope that the host environment provides the functionality necessary to build real-life applications. The question now arises why, if the host environment is intended to fill in the numerous and large gaps in GELLO's capabilities, GELLO itself should be used at all. Most developers would instead be comfortable using a mainstream programming language of their choice for this purpose.

- Experience by developers of OCL-related software such as Demuth¹¹ indicates that OCL's syntax (based on the language Smalltalk) takes getting used to. OCL's syntax is a significant barrier to its use by IT professionals, who cannot perceive the specific advantages of such syntax (there aren't any) and tend to regard OCL as a product of ivory-tower thinking. Since GELLO uses near-identical syntax, similar problems may arise.
- Chiorean et al.¹² who are advocates of OCL, point out that while the need for an expression language is not disputed, OCL in its present state is not sufficient for the purpose. While the language has evolved (the current version is v 2.0), the evolution does not appear to have been driven by the needs of production systems: many limitations

identified in earlier analyses have not been fixed. These authors discuss extending OCL in various ways, but as long as such extension attempts are isolated efforts rather than part of the standard, the advantage of using a “standard” vanishes.

- For all the above reasons, more than a dozen years after its creation, OCL itself currently has very little traction compared to the rest of UML, and the only enthusiasm for it seems to be among academic computer scientists who have not had to wrestle with hard modeling problems for their livelihood. Given the low demand for OCL by IT professionals, many UML tools (including Microsoft’s Visual Studio 2010 Enterprise Edition) don’t support it. I am concerned that the GELLO effort may similarly go nowhere. The advantages of a newer approach must be compelling: mere compliance with a nebulous “standard” is not enough.

The bottom line is that EMR vendors, like any team that has to deliver production software, will use standards only to the extent that they derive a genuine productivity or inter-operability benefits from doing so, in or when a standard is legally or practically forced on them. If none of these conditions holds, they will simply choose the approach that works best for them.

10.4

Web Services for Clinical Decision Support

Several medical informaticians, notably Kensaku Kawamoto (currently of Duke University) appear to subscribe to the practical perspective that one cannot expect existing EMRs to be re-engineered to meet the requirements of a guideline engine. EMR vendors are understandably reluctant to perform the software equivalent of a heart-lung-liver transplant. If progress has to be made, the developers of any proposed solution must work with EMRs as they exist now, rather than assume the existence of the ideal EMR that they wish would exist. Kawamoto and Lobach have designed and implemented SEBASTIAN,¹³ a model of executable medical knowledge based on a Web-service architecture, while (sensibly) ignoring the workflow-operations issues for the most part.

SEBASTIAN defines medical knowledge using “executable knowledge modules”, similar in philosophy to Arden MLMs, except that these use an XML syntax. The implementation uses class library functions developed with Java: most functions are packaged as services (i.e., they are callable over a network). Thus the various functions – chiefly data access – that an EMR provides to a decision-support software application (e.g., get the most recent value of a particular laboratory-based parameter for a particular patient) are accessed *indirectly* through a service.

The idea is that the service’s interface is standardized so that the actual data access logic, in terms of accessing database schema elements, is a “black box” and transparent to the caller, who is only concerned about how to compose the request and in what format the requested data will be sent back. This way, one could replace one EMR with another: while the data-access code will naturally have to be redeveloped, as long as the calling interface remains the same, the software application is unaffected.

A “terminology service” provides a translation facility that maps the data elements used in the guideline to those existing in the EMR: both types of elements are mapped to elements in standard controlled vocabularies (SEBASTIAN uses the Unified Medical Language System).

Kawamoto and Lobach¹⁴ have advocated a standard XML-based protocol (based on SOAP, formerly an acronym for Simple Object Access Protocol) for communication between biomedical Web services. HL7 has now become involved: Kawamoto is an active member of the HL7 technical committee for Web services. The 2009 paper of Kawamoto et al.¹⁵ describe a standardization effort (currently in an early stage) in partnership with the Object Management Group (who are responsible for UML, among other things), for a basic set of Web services that an EMR could provide. While this paper is somewhat sketchy with respect to technical details, the ideas are very clear and elegant. The handful of services that have been agreed upon are, too skeletal to be sufficient for most full-fledged decision support – for example, there is a service to get a single (most recent) parameter value back, but no service to return a stream of values – but this is to be expected: as experience accumulate, more standard services will undoubtedly be defined.

I’m all for a service-oriented architecture, subject to the caveats I’ve discussed earlier: a service must provide sufficient reusable value and do enough work that the overhead of network latency is amortized. Web services definitely make sense in an Intranet scenario where software from different vendors must interoperate. However, I can see circumstances (such as bulk data transfer) where software developers will prefer to bypass the Web-service mechanism, which is currently designed to transfer only a limited amount of data, in favor of vendor-specific and non-portable, but highly efficient, mechanisms.

Also, the IT world has become somewhat less enamored of SOAP, which is anything but “Simple”: SOAP messages carry a large amount of stereotyped XML baggage, which an application program doesn’t care about, so that the latter continually need to be parsed out. SOAP was designed for the worst-case scenario where a message may go through one or more intermediary services, and where the caller may omit necessary parameter values when making the request. (In other words, SOAP was designed to be robust in a Murphy’s Law scenario.) In the much simpler Intranet scenario, where point-to-point communication may be assumed and where the developers of the Web service and the calling applications coordinate their efforts (or at least, should be doing so), alternative, more lightweight approaches, such as Representational State Transfer (REST)¹⁶ are preferable.

Kawamoto and Lobach¹⁴ also envisage a situation where medical services could be available across institutions. They are big on automated Web-service discovery, an undoubtedly useful technology that I remain skeptical about in its current incarnation. Simply knowing that a particular Web service exists, based on an XML-based description published by its creator, provides no indication as to its fitness for its intended purpose – indeed, there is no guarantee that the service even works robustly, or at all. Trying to figure out what a suite of Web services does based on such descriptions is like trying to figure out the usefulness of a Java-based subroutine library based on automatically generated JavaDocs boilerplate, which simply lists classes and their associated subroutines. In the absence of detailed descriptions expressed in clear prose, providing both a high-level overview and lots of examples, the target audience is simply going to spend most of their time scratching their heads. Busy software developers have better things to do with their time than attempt to read someone else’s mind.

Here, unless I see actual demonstrations of scenarios where such services provide value, work efficiently, and also address issues such as patient-privacy and requester-authentication issues, I prefer to adopt a wait-and-see attitude. Certainly, organizations that manage reference content that is widely used by the biomedical community would definitely use such mechanisms to make their content readily accessible to application software. NCBI, which maintains a variety of databases, has already been doing this well before the HL7 effort was under way: privacy and authentication issues aren't involved here.

For the moment, however, I can't visualize a scenario where a software application hosted at one caregiver organization makes a request to a service hosted at another institution asking for *full* details about a particular patient who was previously being managed at the latter institution. Even with a national-health IT infrastructure in place, a lot of other problems need to be solved first.

- For example, patients must be specified correctly in the request. Social Security Number is used in the absence of a universal Patient Identifier, but it has limitations. A mother of five children, for example, may be hard-pressed to recall the SSNs of each of her children accurately.
- A harder problem is that of presenting the data returned in a semantically meaningful way that is independent of the particular EMR where it originated. At present, when information is sent between institutions, it is in the form of hard-copy or fax. This information is human-readable, but is not in a form that can be "consumed" and automatically integrated into the requesting (receiving) EMR. As I've stated in the Introductory chapter of this book, HL7 version 2.x, the current version of the protocol for electronic interchange of medical data, is so underspecified that getting any two arbitrary EMRs to interoperate is problematic.
- Safeguards must be implemented to ensure that such a service cannot be invoked inappropriately: this would imply maintaining an electronic trail of the actions, and the individuals, involved in an individual operation. Such operations have too many potential consequences to be carried out automatically without some measure of human oversight.

Despite the above caveats, I wish Dr. Kawamoto and the HL7 team all possible success. They have seen the writing on the wall: if and when true EMR interoperability becomes a reality, it will have to be based on Web technologies. The only way to make progress is to try to develop consensus standards, and experiment with them using the crucible of production applications.

10.5

The Proteus Guideline Framework

The Proteus guideline framework has been developed by a team led by Dr. Hemant Shah (currently at Henry Ford Health System (HFHS), Detroit). It deals with many of the issues I have raised with respect to the guideline languages. (Full disclosure: While not involved

with the development of the code, I have served as an adviser to the project.) I'll summarize the highlights here:

- Proteus consists of a guideline authoring environment, a developer toolkit, and a runtime environment. It envisages a number of interoperating modules (called "Knowledge Components" or KCs for short) that can be passed data and return a value. A guideline implementation is comprised of one or more KCs, invoking each other in a hierarchical fashion in the same way that subroutines in a program call other subroutines.
- An individual KC can be as simple or as complex as the guideline developer chooses: the Proteus design allows reuse of existing code, which must be enclosed in a "KC wrapper" that lets it communicate with the runtime. Thus, for example, existing code that runs as an executable, but which can be called through command-line parameters (such as a neural network program or a classical multivariate regression module) can be invoked this way.

This design is also highly modular: thus, a given KC may be used in several guidelines.

- The runtime is agnostic as to whether all the KCs in an operational guideline run on a single CPU or whether individual KCs run on a network machine (as Web services). The interface around an individual KC, which only cares about what data is passed to it, ensures a loosely coupled architecture. The physical topology of the KCs is left to the implementer, but does not affect the logical KC hierarchy.
- Many practical issues of implementation that are important for operational guidelines have been considered in depth. For example, the metadata for individual data elements required by a KC is fairly comprehensive. When a KC is called, it may not have available all the data that it needs to return a value: in this case, the user may override its action and supply a value (the alternative, of course, is for the KC to terminate). Also, when a guideline runs for a given patient, it is important to maintain a trail of the path that was followed (especially in situations where a user is allowed to override a guideline's recommendations).
- The access to an individual EMR is handled by a metadata-driven abstraction layer that is part of the runtime environment. The pilot implementation of Proteus involves access to two versions of the institutional EMR, both of which were developed in-house at HFHS: the current production version, which has been in existence for more than 20 years, and the "next generation" version, which has a significantly different database schema and makes much greater use of standard biomedical controlled vocabularies. The exercise of creating this abstraction layer, so that the rest of the runtime environment is unaffected by changes in the back-end EMR, will provide a robust test of the soundness of the metadata design: it will, understandably, rely on standard biomedical vocabularies to provide a mapping between the data elements required by a guideline and those present in the target EMR.
- Finally, Proteus is available as open-source at www.proteme.org: the site also includes a few tutorials. Developer documentation (other than that based on JavaDoc) is somewhat limited, but is expected to improve with time.

My (possibly biased) take on Proteus is that, while the effort is at an early stage, it holds great promise. Its overall design is well thought out, and is in line with the philosophy that

successful software engineering practice must maximize reuse of existing operational software, rather than force adopters of a technology to perform major surgery on what already works (or throw it out) just to be compliant with a so-called standard. The open-sourcing of Proteus will hopefully lead to the establishment of a nucleus of hard-core developers who will enhance the system. While Proteus deals with a higher-level problem than the Web Services effort described earlier, it is in theory fully compatible with the latter. (Proteus is a high-level architecture, while Web services are a possible mechanism for providing the underlying plumbing.)

10.6 Conclusions

This section has been different from the others in this book in that it has mainly been discursive, emphasizing issues where outstanding challenges still loom, and for which consensus about solutions (or even approaches to solutions) have yet to crystallize. Such background is necessary to serve the purposes of this book. I'm hoping for the commoditization of decision-support software, especially that which is targeted at the currently underserved small- and medium-practice market. Such commoditization can come about if a critical mass of serious software developers gets involved in developing useful as well as affordable systems. Such developers need to understand some of the problems that their software must solve, and I've hoped to provide that foundation here.

I've made it clear that, while I'm all in favor of standards, like many of the biomedical software vendors themselves, I remain highly skeptical of putative standards that are designed by committee and handed down without ever having proved their worth in multiple production environments (or even a couple of environments outside the originators' institutions). The highly political nature of standard setting (the World Wide Web consortium is one of the few where the participants seem to work relatively selflessly), does not engender confidence in the mind of the early adopter.

For every one standard-by-committee that has worked, many more have failed. In the area of distributed computing alone, Richard Martin¹⁷ identifies at least three predecessors to Web services that have more or less gone the way of the Dodo, *despite* strong industry support *and* the existence of multiple working implementations: the Distributed Computing Environment (DCE), Common Object Request Broker Architecture (CORBA), and the Distributed Common Object Model (DCOM): all of these had difficulty in operating across institutional firewalls, and in some cases the standard was either too abstract (DCE) or too ambiguous and incoherent (CORBA).

As in the case of numerous other technologies that which we've discussed previously, most of the unresolved issues require vendor-specific extensions to be addressed. If the "standard" is weak enough and the required extensions numerous enough, it may become an object of ridicule. Premature standardization can be the root of much evil: the process of standards evolution should be a Darwinian one, with survival of the fittest. A variety of competing, freely available, working implementations may offer the only way forward.

References

1. Miller R, Masarie FE, Myers JD. Quick medical reference (QMR) for diagnostic assistance. *MD Comput.* 1986;3(5):34-48.
2. Warner HR Jr. Iliad: moving medical decision-making into new frontiers. *Methods Inf Med.* 1989;28(4):370-372.
3. Peleg M, Tu S, Bury J, et al. Comparing computer-interpretable guideline models: a case-study approach. *J Am Med Inform Assoc.* 2003;10(1):52-68.
4. Boxwala AA, Peleg M, Tu S, et al. GLIF3: a representation format for sharable computer-interpretable clinical practice guidelines. *J Biomed Inform.* 2004;37(3):147-161.
5. Wikipedia. IntelliSense. Available from: <http://en.wikipedia.org/wiki/IntelliSense>. Cited 3/2/2010.
6. Tu SW, Musen MA, Shankar R, et al. Modeling guidelines for integration into clinical workflow. *Stud Health Technol Inform.* 2004;107(Pt 1):174-178.
7. Tu SW, Campbell JR, Glasgow J, et al. The SAGE Guideline Model: achievements and overview. *J Am Med Inform Assoc.* 2007;14(5):589-598.
8. Ram P, Berg D, Mansfield G, et al. Executing clinical practice guidelines with the SAGE execution engine. *Stud Health Technol Inform.* 2004;107(251-5):251-255.
9. Sordo M, Ogunyemi O, Boxwala AA, Greenes RA. GELLO: an object-oriented query and expression language for clinical decision support. *AMIA Annu Symp Proc.* 2003:1012.
10. Warmer JB, Kleppe AG. *The Object Constraint Language: Precise Modeling with Uml*, Addison-Wesley Object Technology Series (Paperback). Reading: Addison-Wesley; 1998.
11. Demuth B. The Dresden OCL toolkit and the business rules approach. Available from: http://st.inf.tu-dresden.de/files/papers/EBRC2005_Demuth.pdf. Cited 7/6/2010.
12. Chiorean D, Bortes M, Corutiu D. Proposals for a widespread use of OCL. Workshop on tool support for OCL and related formalisms; 2005; EPFL, Montego Bay, Jamaica.
13. Kawamoto K, Lobach DF. Design, implementation, use, and preliminary evaluation of SEBASTIAN, a standards-based Web service for clinical decision support. *AMIA Annu Symp Proc.* 2005:380-384.
14. Kawamoto K, Lobach D. Proposal for fulfilling strategic objectives of the U.S. Roadmap for national action on decision support through a service-oriented architecture leveraging HL7 services. *J Am Med Inform Assoc.* 2007;14:146-155.
15. Kawamoto K, Honey A, Rubin K. The HL7-OMG healthcare services specification project: motivation, methodology, and deliverables for enabling a semantically interoperable service-oriented architecture for healthcare. *J Am Med Inform Assoc.* 2009;16(6):874-881.
16. Kay R. QuickStudy: representational state transfer (REST). Available from: http://www.computerworld.com/s/article/297424/Representational_State_Transfer_REST_. Cited 11/2/10.
17. Martin R. Web services: hope or hype? Available from: <http://www.cs.rutgers.edu/~rmartin/talks/WS-comrise.pdf>. Cited 4/6/10.

In a previous chapter, I introduced the use of the Entity-Attribute-Value model for representing highly variable clinical data. EAV is appropriate for clinical data because the universe of clinical attributes that can apply to a patient are both *numerous* and *sparse*. I emphasized that the use of the EAV model is unworkable in the absence of a supporting metadata infrastructure that supports validation and presentation and, of course, code that operates on this metadata.

The focus of this chapter is “mixed” schemas, where *data* exists in both traditional tables with columnar attributes as well as with an EAV representation. The supporting metadata, as I’ve stated earlier, is almost always modeled in the traditional columnar way. Mixed schemas constitute the majority of real-world schemas.

In this chapter, I’ll first discuss the various circumstances besides clinical data where EAV designs are appropriate – which automatically leads, by exclusion, to circumstances where they are *inappropriate*. I’ll then discuss the requirements of mixed schemas, and describe an approach, the EAV/CR model, that meets these requirements. The most important requirement, by far, is to minimize the need to write special-purpose code that performs the common operations of data creation, read, update and delete (so-called CRUD): I’ll demonstrate how the appropriate SQL can be generated. Finally, in an appendix, I’ll discuss alternatives to EAV design for sparse attributes.

In the account below, I use the term “class” as synonymous with a physical table or one that is simulated through an EAV approach. This usage is not as strict as that in object-oriented programming, where a class can also have associated code (“methods”). Also, in a relational database, a class may sometimes correspond to more than one table in circumstances where many-to-one repeating data that is logically part of the class has been split off into a related table in order to keep the data in at least first-normal form.

Before reading the rest of this chapter, you may want to skim [Chaps. 4 and 5](#) again to refresh your memory, since this chapter builds on the foundations laid there.

11.1

Additional Indications for EAV Design

There are certain circumstances besides numerous, sparse attributes that justify the use of an EAV approach. We consider each below.

11.1.1

Numerous and Volatile Data Classes, Modest Data Volumes

Here, one encounters *numerous classes* of data, possibly exceeding a hundred: these *classes are in constant flux* (i.e., more classes get frequently added, and their structure may be frequently modified) but the *number of rows per class is, for the most part, relatively small* – e.g., a couple of hundred at most, so that runtime efficiency when operating on a single class is not particularly critical. Within a single class, the attributes are not necessarily sparse.

This situation occurs in basic biomedical research (e.g., there are a limited number of neurotransmitters or neuronal receptor subtypes). It also occurs in the context of tools that support modeling for *prototyping* or *simulation* purposes, where the model will be changed frequently by the experimenter/designer. In both these situation, a schema simulated using an underlying EAV model is somewhat simpler to reconstruct between design iterations. Especially in circumstances where the framework is intended to support simulation, power-users do not need to move data around between tables, and rebuild the user interface, each time they decide to restructure their models. If the metadata-authoring environment for design allows detailed specification of validation and presentation information, then basic interfaces for data presentation, creation and editing data from individual classes can be generated dynamically: therefore redesign of the user interface is also not required.

There is a caveat here. Most simulations are short-lived explorations or demonstrations of ideas. Sometimes, however, the design turns out to have longevity, and over years, the schema accumulates significant volumes of data. At this point, you may discover that certain classes have acquired large numbers of rows, and also that their structure has stabilized, in that there are no more design changes required. At this point, poor performance for such classes becomes a significant issue. You must then consider *remodeling* such classes as traditional columnar tables.

The focus of this chapter is to show you how to design your metadata so that your application becomes immune to changes in the physical representation of a class or attribute.

11.1.2

Hybrid Classes

Certain classes show both sparse and non-sparse traits. That is, certain attributes are *always* populated, while others are highly variable and sparse. I refer to such classes as “hybrid”. This occurs most often when modeling manufactured products. All products share common information (e.g., name, unit price) but the details of a product depend on its category: thus, details of an item of laboratory equipment are quite different from those for consumables such as a pharmaceutical product. Hybrid classes may even occur with personnel information: all people have common information, but certain details would vary depending on the individual’s role in the organization. For some purposes (e.g., payroll) the common information is utilized, while for others, one operates mostly on the variant information that is unique to a particular category of individuals.

In object-oriented programming, one might deal with this situation through “inheritance”: thus, both the classes *Pharmaceutical* and *Equipment* inherit common information from the class *Product*. About the only RDBMS that supports inheritance explicitly is PostgreSQL.¹ In other mainstream RDBMSs, you need to simulate inheritance by creating a base table as well as special-purpose tables for each product category that are linked to the base table (with other tables linked to these, in case there are sub-categories).

In today’s applications, however, product categories number in the hundreds and new ones are regularly introduced. Continually modifying the schema whenever a new product category comes along (and creating a new user interface for each) is needless work. Instead, a metadata-driven approach, coupled with EAV storage, for capturing variable category characteristics is more appropriate.

Specifically, to model a hybrid class, *the attributes that are constant are modeled using a physical table with the same name as the class, where these attributes are explicitly named columns. The attributes that are variable and sparse are modeled using the general-purpose EAV design.* It’s possible to create one special-purpose EAV table for each hybrid class, but such a design, in my opinion, needlessly increases the programming effort for the user interface. The *Object Dictionary* approach that I’ll demonstrate subsequently, however, utilizes an Object ID as the primary key for every hybrid-class table. Because the Object ID that is unique across all data tables in the schema, it becomes possible to use the general-purpose data-type specific EAV tables to record sparse data on *all* hybrid classes in the database.

11.1.3

Maintaining an Audit Trail Across Numerous Tables in a Schema

Here, the objective is to record when and by whom a particular column, in a particular row and table in a database, was changed, along with the field’s previous value. (Such records are called “deltas” – delta is the Greek symbol that is used to indicate differences in calculus.) An attribute-value representation is the only way to allow capture. The audit-trail table has the layout shown in Table 11.1 below.

In the table below, the **Bold** columns collectively represent the table’s primary key. For a given table, the *PrimaryKeyValue* represents the entity; the *columnName* represents the attribute, and the *OldValue* the Value. Writes to this table are activated by an application

Table 11.1 Structure of a general-purpose Audit Trail table

Column Name	Type
TableName	String
ColumnName	String
PrimaryKeyValue	String
DateTimeChanged	Datetime
UserID	Integer
OldValue	String

that, prior to editing, caches the records' old field values in memory: when the user saves the record, the application compares old values to the current ones to determine that a given field's value has changed.

In terms of the EAV model, the "entity" is conceptually a combination of the *Tablename* and the *PrimaryKeyValue*; the "attribute" is *ColumnName*, and *OldValue* represents the value.

As I'll show subsequently, if we maintain a data dictionary of the entire schema, the *Tablename/ ColumnName* pair can be replaced with a unique integer ID. If every table in the schema has a surrogate (machine-generated integer) primary key, the *PrimaryKeyValue* column can also be an integer. In practice, this may not happen – in the worst case, the table may have a compound primary key – and so this column may have to contain a delimited string where the delimiter is a character not normally used for data, such as the character that corresponds to the ASCII value of 255.

In a schema intended to support auditing, every user of the system should be assigned a unique ID, which can be used for purposes such as the above. This ID is typically based on the surrogate-primary-key in a *Users* table. The *oldValue* column is a string, irrespective of the column's actual data type: some developers may prefer to use datatype-specific tables, but that approach is probably overkill, since the audit-trail table is not consulted all that frequently, and one would hardly ever want to do searches based on the old value. Based on your needs, you may prefer to designate an additional column that stores the Current Value, though this value can also, with modest effort, be looked up in its indicated location (select *Columnname* from *Tablename* where *<Primary Key Expression>*).

This approach will not work in circumstances where records are deleted (so that all details of the record containing the primary key are lost). For it to work, one of two approaches must be followed:

1. Deleted records in a table are archived to a shadow table with the same structure as the data table, plus columns recording the date/time and the user who deleted the record. This is best performed through delete triggers on individual tables.
2. Physical deletions must not be allowed: records can only be marked as inactive/deleted. This is the approach followed in so-called temporal databases, which are essentially intended to "remember" forever. With disk space getting progressively cheaper and abundant, this approach is not as far-fetched as it sounds. For large and highly volatile tables where there are numerous additions and deletions, only a modest proportion of the records may be active. This can impact performance, because every single SQL statement issued against the table must be additionally qualified by the filter "and *deleted_status* = false".

Also, for this approach to work, a surrogate primary key (where deleted values are never recycled) must be used; otherwise, the later attempt at inserting a record whose primary key coincidentally matches that of an inactive record may confuse users who don't know about the system's internal design.

In our own designs, we tend to use the first approach, though I have used the second for low-volatility tables of modest size.

The need to maintain a change trail (so-called *delta tracking*) is so common that it cuts across domains: it is as important in "business" applications as it is in biomedicine. Even

if every category in your database is modeled in traditional columnar form, it shouldn't take you long to convince yourself that there is no other way to model this problem.

11.1.4

Special-Purpose EAV: The Semantic Web

The EAV model's generality has been used in the context of Semantic-Web applications, where the nature of the information that needs to be represented is highly variable. Here, much of the information is represented under the hood as "Resource Description Framework (RDF) triples", where a triple consists of a subject (analogous to an entity), a predicate or property (=attribute) and an object (=value).

RDF is a little like assembly language – it is far too low-level for most developers' comfort, and so higher-level languages, such as the Web Ontology Language (OWL) are built on top of RDF. While many applications involve storing RDF content in a database, the technology used is not necessarily relational. A whole set of technologies has developed around RDF, such as a SQL-inspired query language, SPARQL.² I'm not currently a great fan of RDF-triple technology, but that's possibly because I haven't had to use it much.

11.2

Requirements for Mixed Schemas

Mixed schemas contain three classes of data tables:

- **Conventional:** All attributes are implemented as columns. Thus, for a patient demographics table, Name, Date of Birth, Gender, Social Security Number, etc. are individual columns. In bioscience.
- **EAV:** The general-purpose data-type-specific EAV tables are used, containing Class and Attribute IDs that point to metadata tables.
- **Hybrid:** Here, some attributes are captured in a conventional table, while sparse attributes utilize the general-purpose EAV tables.

As I will describe shortly, a shared metadata infrastructure must support *all* classes.

The presence of mixed schemas raises the following issues.

11.2.1

Limitations of the Basic EAV Model

The basic Entity-Attribute-Value model, as described in the previous chapters, turns out to be inadequate for the purpose of simulating traditional schemas. It has the major drawback that the "value" of EAV is restricted to atomic data only: numbers, strings, dates and binary data such as images: the values lack sub-structure. Though binary data may have substructure, relational databases and SQL have traditionally ignored it, leaving it under the

purview of special code that understands the structure. Modern “object-relational databases” allow such code, written in Java or a .Net language, to be hosted within the DBMS and invoked from SQL, so that the distinction is gradually being blurred, but the issues of dealing with binary data are beyond the scope of this book.

The *inability to represent composite objects and inter-class relationships* matters greatly for the volatile-class scenario, where EAV is being used to simulate ordinary tables with non-sparse data. Recall that in a traditional schema, you have foreign-key relationships between tables with referential integrity. For example, if you record a Patient ID in a Visits table, the ID must belong to a valid row in the Patients table. When using EAV to simulate a traditional schema, the equivalent of foreign-keys and referential integrity must be provided by the framework, or else errors will be introduced into the data.

11.2.2

Supporting Routine Operations

The code that supports Creation, Reading, Update and Deletion (CRUD) must be immune to the periodic physical reorganization that is a necessary consequence of database tuning. Such immunity is only possible if the CRUD code, rather than being implemented by hand-coding, is generated *dynamically* by consulting metadata that records information about the physical representation.

11.2.3

The Need for Additional Data Type-Specific EAV Tables

I’ve recommended the use of data-type specific tables such as dates, integers, decimals and strings. For mixed schemas, the list of general-category data types listed above is not enough. We must add two more data types.

- “*Class*” allows values to be entities (objects), so that sub-structure or object references can be modeled. (Actually, we store a *reference* to the Object, its ID, as discussed shortly.) In the metadata describing a given attribute, we must have another column that specifies *which* class the attribute must belong to. *This is the equivalent of a foreign-key reference in the case of a physical table.*
- “*Custom Data type*” indicates that the attribute inherits its validation and presentation characteristics from a custom data type. (See the section on Custom Data Types later.) An additional column in the attribute metadata defines which custom data type we are using.

11.3

The EAV/CR Approach

I now present an approach for managing mixed schemas called EAV/CR,³⁻⁵ devised primarily by my friend and colleague Luis Marengo and myself at Yale – with ideas borrowed

from Tom Slezak of Lawrence Livermore National Labs and Dan Masys, currently Chair of the Biomedical Informatics Department at Vanderbilt University. EAV/CR originally stood for EAV with Classes and Relationships, but as we worked with it we discovered the necessity for being flexible: certain kinds of data were simply better suited to traditional columnar representation, and there was nothing inherent about the model that mandated EAV storage. So now, like many other abbreviations (OLE, SOAP) EAV/CR does not stand for anything. We characterize it, however, as a storage-representation-agnostic approach to developing database applications characterized by the use of detailed descriptions of data classes and their attributes from the data definition, validation, presentation and runtime perspectives, coupled with the use of an Object Dictionary and inter-class relationship information.

An operational EAV/CR framework has three components: metadata, data and generic code. The code continually consults the metadata, and is primarily concerned with generating user interfaces (with static or dynamic components) that support CRUD operations with robust validation. A freely accessible application that is built with this framework is SenseLab,⁶ implemented by Prof. Marengo, a database containing diverse content related to neuroscience (which is populated by numerous researchers associated with Prof. Gordon Shepherd's lab at Yale).

The approach we have taken for metadata design is inspired by the catalog tables of database engines, which I now describe briefly.

11.3.1

RDBMS Catalog Tables: An Introduction

All RDBMS engines contain, for every database, a set of *System Tables* (also called *Catalog Tables*) that record knowledge about the database schema: tables, columns, indexes, relationships, views and so on. Whenever you create a schema object, such as a new table, index, constraint, and so on using the vendor's Data Definition Language subset of SQL, you are really creating entries in the system tables.

The internal catalog-schema designs of individual RDBMSs are proprietary and subject to change with each new version. However, ANSI/ISO SQL defines a standard mechanism called Catalog Views, supported by most RDBMS vendors, which will return most of the schema information that developers care about, using SQL queries. Tools that query the catalog tables can serve a variety of useful purposes. Microsoft Visio and similar programs use catalog information to reverse-engineer a data model from an existing database.

Individual RDBMS catalog capabilities vary greatly. For example, microcomputer RDBMSs such as MS-Access record presentation information (captions, display as text boxes vs. check boxes vs. pull-downs), while high-end "client-server" RDBMSs completely ignore presentation issues: presentation is thought to be a matter best delegated to client software.

RDBMS catalog contents are read-only: they cannot be altered with SQL insert, update or delete statements. Also, because they are so critical to the database, the structure of catalog tables cannot be altered or extended by developers. The EAV/CR framework must therefore implement its own equivalent of the catalog tables – the Dictionary tables – to

record presentation, validation and descriptive/documentation information about the rest of the system. They also contain information about themselves – i.e., they serve the ancillary purpose of self-documentation.

11.3.2

Dictionary Tables: Summary

The structure of the EAV/CR dictionary tables represents a subset of the system catalog tables as well as a superset. It is a superset because of the presentation information. It is a subset because these tables are not intended to reinvent wheels that work very well. For example, the application software does not need to care about which columns of a table are indexed, and indeed cannot do anything about it: this aspect is the concern of those who administer the system and tune it to be responsive to workloads. As in the catalog tables of an RDBMS, the dictionary tables must also contain entries that describe themselves.

The idea of dictionary tables as the basis of user interface generation is not new. Back in 1991, the venerable PowerBuilder (now owned by Sybase) introduced the idea of “extended attributes” – presentation metadata associated with individual tables and columns.

11.4

The Dictionary/Metadata Tables

In the account below, I’ll describe what has worked for us. Based on your own needs, you may want to include additional tables. At the very least, you need to have at least three tables, which serve the following purposes:

- **Class Descriptions:** This table (in our system, we call it `Meta_Classes`) contains one row for every physical table, view and every simulated table in the system – including the dictionary tables. The type of the class can be one of the following: View, Physical Table, EAV, and Hybrid.
- **Attribute Descriptions:** (`Meta_Attributes`) contains one row for every attribute of every class in the system. For physical tables, attributes correspond to columns. For hybrid tables, the Attribute Representation can be Physical Column or EAV.
- **Relationships:** This contains one row for every conceptual primary-key/foreign-key relationship in the system.

Our own schema contains several other tables:

- The **Choice_Sets** and **Choice_Set_Values** tables, introduced in the chapter on Data Types to support the representation of enumerated data types, are, in my opinion, an essential part of any schema, not just a clinically oriented one.
- **Class Hierarchy:** In situations where inheritance needs to be modeled, this table records the relationship between Parent and Child Classes. This situation commonly occurs in bioscience: thus, the classes “Receptor” and “Enzyme” both inherit from “Protein”.

Note: In our own experience, we have not had to actually implement class inheritance very often: in most cases, the alternative approach of *containment* (one class contains a member of another class) seems to work just fine. In fact, Microsoft's .NET Application Architecture Guide, 2nd Edition, specifically discourages inheritance as a good design practice, stating: "Wherever possible, use composition over inheritance when reusing functionality because inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes. Using composition also reduces the inheritance hierarchies, which can become very difficult to deal with."

- **Class Presentation Information:** Data from a single class is rarely presented in isolation. In most situations, when inspecting the details of a single record from a given class, one wishes to see a summary (one-record-per-line) of information that is related many-to-one. For example, when inspecting a patient's demographics, one also wants to see a summary of previous visits/encounters. In many cases, it is desirable to be able to look at such nutshell information from multiple related classes. (Here, to avoid cluttering the display, the related information is typically segregated into multiple tabs, one for each related class.) This table records, for a given class, which other classes' data is to be presented and in what order.
- **Cross-Attribute Validation/Ergonomics Metadata Tables:** The infrastructure that I've described for cross-attribute validation, skip logic and dependency tracking in Chapter 5 will reduce the need for *ad hoc* programming effort when applied to *any* database, not just one dealing with clinical data or one using an EAV design. The EAV/CR schema therefore contains tables with a design almost identical to those illustrated in Fig. 5.2 of Chap. 5. The only difference is that instead of Form ID, we use Class ID, and instead of Parameter ID, we have Attribute ID.
- **Attribute Hierarchy:** You may or may not need this table. This table has two columns, Parent Attribute ID and Child Attribute ID, and models a *very* limited degree of inheritance among attributes. Attribute inheritance in databases turns out to be something of a rat's nest, but in a limited form it turns out to be useful in certain situations where one wants to retrieve data on an attribute as well as all its children, without having to specify the latter explicitly.

This situation occurs in large institutions where there are multiple laboratories all performing certain tests. Thus, strictly speaking, one cannot talk of the serum sodium alone: one must refer to the serum sodium as performed by lab A or lab X. The "normal" ranges in each lab are permitted to vary slightly from each other – though not dramatically so, for reasons of quality-control. For the same patient, different labs may perform their version of the test at different times. In the database, each test variant will have its own separate Attribute ID, so that those who need to know can determine in which lab a test was performed. However, most users may not care about such details, and if they simply want all serum sodium values for a particular patient, the system must fetch all children of a "generic serum sodium" concept. From the attribute-hierarchy perspective, "Lab A serum sodium" and "Lab X serum sodium" are children of "Generic serum sodium".

Class Categorization: This is a means of categorizing classes based on subject matter. SenseLab serves diverse sub-communities of olfactory (smell) neuroscientists. Presenting all of the numerous classes in the system can be overwhelming to most

users, who primarily care about rapid access to classes and functionality of interest to their sub-community. SenseLab also has a team of curators, each specializing in different subject areas.

Prof. Marengo solved the problem of customization and selective editing access by allowing the definition of *subject matter areas*: each of these served as the basis of a *custom Web portal* as well as restriction of edit access. In SenseLab, an individual portal is also associated with extensive metadata related to presentation and administration, such as the default URL, designated administrator/s, theme (foreground and background colors, logos, etc.), and so on. This metadata allows new portals to be set up very rapidly when a request for a new subject area is approved. Obviously, while users can choose a default portal, they can access any of the other portals for data-browsing purposes.

Figure 11.1 shows the essential sub-schema (detailed in Appendix 11.1) of an EAV/CR design. I've omitted the validation/ergonomics tables, to avoid repeating Chap. 5 content. I've included only the non-variable component of the data tables – the most important table being the Objects table discussed shortly. (In a real, mixed schema, you will have numerous conventionally designed tables, but these will be specific to your application.)

11.4.1

Synchronizing Dictionary and Catalog Table Contents

For conventionally modeled tables, much of the information in the Dictionary tables is redundant with that already in the RDBMS catalog tables. If, every time changes are made to the schema, the development team is forced to manually update the Dictionary tables, one can pretty much guarantee that the two will gradually go out of synch, to the point where the Dictionary tables cease to be a source of truth. In situations where application code is constantly consulting these tables, asynchrony can be disastrous.

Therefore, automated or near-automated synchronization must be facilitated. In some RDBMSs, this may mean clicking a button in a developer-oriented interface that populates or updates the appropriate Dictionary content after a design chore has been accomplished. In certain RDBMSs such as Oracle 11 g and SQL Server 2005 and later, there is a powerful feature called Data Definition Language (DDL) triggers, which let developer-defined code execute in response to specific DDL commands, such as CREATE TABLE or ALTER TABLE.

Within the trigger, the developer has access to the necessary information about who made the change, to what object, and when. (For MS SQL Server, you need to know basic XML processing to extract these.) The details of the change itself are recorded as the actual SQL statement that caused the trigger to activate. The hard part here is writing a parser that will convert the SQL into modifications of the Metadata tables' contents. Writing a parser using a proprietary dialect of procedural SQL can be traumatizing, but fortunately you have the alternative choice of a .NET programming language (for SQL Server) or Java (for Oracle).

There are several freeware SQL parsers available, as well as a commercial SQL parser <http://www.sqlparser.com>, which is available both in Java versions and .Net versions. However, in order to create your own triggers, you need source-code that you have to compile, so you're better off with an open-source version even if it's much less powerful than a commercial, fully supported binary that you won't be able to embed.

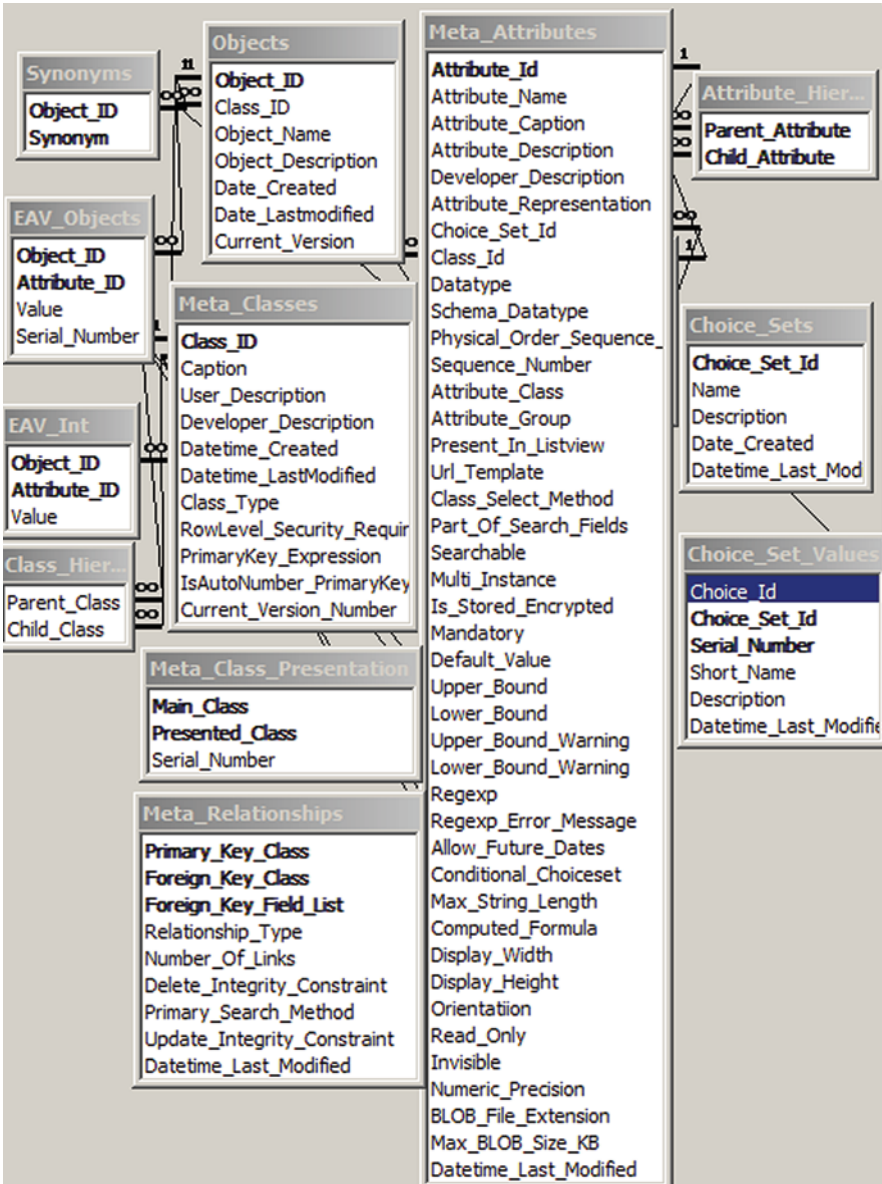


Fig. 11.1 EAV/CR Schema essentials. All tables other than Objects, Synonyms, EAV_Objects and EAV_Int represent metadata

Your parser may only meet the easy 80% of your needs, but that may suffice as long as your code can fail gracefully if it encounters constructs that it can't handle, and logs such failures. (Trigger code, like stored code in general, has only a very limited way of communicating with the user in a client-server setting: it's better for the trigger to return a failure code and for the application to test for this.) Fortunately, DDL statements don't tend

to be as syntactically complex as the data manipulation language – notably the SELECT statement, with all its numerous wrinkles.

You will also need to write routines that perform the following functions:

- Creation of a conventional table for a class previously modeled using EAV, if you decide that the choice of an EAV representation was inappropriate.
- Migration of data for a class previously represented as EAV to a conventional table.
- Migration of data in the reverse direction, if you decide to EAV-ify the representation of a class.
- For hybrid classes, migration of data representing individual attributes from EAV to conventional table and vice versa.

11.4.2

Attribute-Definition Reuse: User-Defined (Custom) Data Types

I am indebted to Randy Neall (Veriquant LLC, <http://www.veriquant.com>, Hendersonville, NC), who has implemented an EAV/CR-based system for medical practice management, for this idea.

Defining an attribute from the presentation and validation perspective takes some effort, and one would like to reuse the effort investment in as many places as possible. An example is a bibliographic citation to a scientific article in the widely used PubMed database. The Pubmed/Medline ID is an integer, but this information alone is only partially useful. To provide an interface that enables the user to reach out to a particular article, one must use a *URL Template* that is substituted with the ID of the article of interest. Here, the template has the pattern <http://www.ncbi.nlm.nih.gov/pubmed/ID> (where the bold “ID” placeholder is substituted with the actual ID). NCBI occasionally changes their API: for example, one had to originally use the template: www.ncbi.nlm.nih.gov/htbinpost/Entrez/query?uid=ID&form=6&db=m&Dopt=b.

PubMed ID is likely to be used in several places in a bioscience database (or even a clinical database, where the system must refer clinicians to supporting evidence for a particular suggested plan of action). One would, however, like to define a custom data type “PubMed ID” so that if the template needs to be changed, it is changed in just one place.

I introduced the concept of custom data types in the chapter on data types. When you need to create a custom data type for the purposes of *presentation* (as in the PubMed ID example above), the mechanisms of the RDBMS, which are not concerned with presentation issues at all, are no help, and so it is necessary to devise our own approach.

11.4.3

Implementing Custom Data Types

There are several ways to implement custom data types in an EAV/CR schema.

1. Create a table, **Custom_Data_Types**, that replicates all the validation/presentation columns of the Attribute Descriptions table, with a surrogate Data Type ID primary key.

In the `Attribute_Descriptions` table, those attributes whose behavior is intended to derive from a Custom Data Type point to the `Custom_Data_Types` table through a foreign key, also called Data Type ID.

2. *Make attributes stand-alone*, and make the relationship between attributes and classes *many-to-many* rather than many to one through a bridge table `Attributes_Classes`. This approach essentially treats attributes as synonymous with custom data types.
3. Utilize a self-join within the `Attribute Descriptions` table. (While such a feature is rarely needed, it is perfectly permissible to use the RDBMS's referential integrity mechanism to have a column in a table be a foreign key into the same table.) Add a Boolean column `Is_Custom_Data_Type` to the `Attribute Descriptions` table. This is set to True if the Attribute ID's validation/presentation properties are intended to be reused.

Add another column called `Custom_Attribute_ID`, which is a foreign key into `Attribute Descriptions` itself (pointing to the Attribute ID primary key field). For robust ergonomics and validity, `Custom_Attribute_ID` is enabled only if `Is_Custom_Data_Type` is false. In the user interface, this is based on a pull-down list that is limited to only the Attributes that are reusable.

My personal preference is for the first approach. The second approach is also workable – Randy Neall's variant of EAV/CR employs it – but in the modeling situations I've had to deal with, only a very small minority of attributes are truly reusable, i.e., the relationship between classes and attributes is mostly one-to-many. (Note: Two attributes that are merely based on the same enumerated choice set do *not* represent the same attribute: the same goes for attributes that play the role of foreign keys.)

11.4.4

Aside: The Deliberate Absence of Descriptive Metadata Can Obfuscate Schemas

The more complex a schema gets, the greater is the need to have a component that documents it: conversely, the absence of such metadata obfuscates a system. In this context, Ralph Kimball and Joe Caserta, in "The Data Warehouse ETL Toolkit"⁷ discuss their experience as consultants trying to get data out of SAP's Enterprise Resource Planning system. Offering their "unvarnished recommendations", they describe SAP ERP as "...like a black hole. Rivers of data flow in, but there is no way to get the information back out ...A contemporary SAP ERP implementation is likely to have a data foundation that consists of tens of thousands of physical tables, exhibiting few DBMS-defined table relationships, with entity and attribute names rendered *in abbreviated German*! Thus, the SAP RDBMS, for all practical purposes, is incomprehensible and proprietary."

Not having wrestled with an SAP schema myself, I don't know how many of these numerous tables represent enumerations stored as one enumeration per table: I'm guessing, based on my knowledge of other systems that don't consolidate their enumerations, that such tables would constitute the majority of the tables. It would be straightforward for SAP to provide Class Descriptions, Attribute Descriptions and Relationships tables to their customers. In the healthcare area, for example, EPIC provides a (somewhat) annotated schema description for their EMR – albeit as a compiled HTML file that can be readily disassembled, using Microsoft's freely available HTML Help Workshop, into individual HTML files whose contents can be parsed into such a relational structure.

While SAP ERP comes with a large report library, these reports meet only a modest fraction of most businesses' decision-support needs: one of the basic rules of databases is that users will always want to query the data in ways that the database's designers could never have anticipated. That is why SQL was invented. In fact, for the vast majority of US customers who aren't fluent in German, SQL's view mechanism would allow developers to create simple views that define aliases (in English) for tables and their columns.

It is possible that SAP, a multi-national corporation, maintains such metadata internally, if only to protect the sanity of its vast base of non-German developers. Such metadata, however, is not available to SAP's customers, despite the latter already having paid multiple millions of dollars to purchase the base system. It is perhaps a commentary on the existing degree of (non) competition between ERP vendors that SAP's customer base hasn't yet demonstrated the collective spine to demand such metadata as part of the package.

Instead, SAP sell its customers an add-on, Business Information Warehouse (which, like the ERP system itself, is hardly cheap): Kimball and Caserta point out that this offering still falls considerably short of what developers can create on their own using multiple best-of-breed third-party tools, if only they could decipher the SAP schema.

11.5

EAV/CR Data Sub-schema

11.5.1

The Object Dictionary Approach

The Object Dictionary approach was pioneered in bioinformatics databases by Tom Slezak's team during the Lawrence Livermore chromosome 19 mapping project.⁸ It has been subsequently adopted in numerous production systems, such as NCBI's Entrez. This approach is what makes EAV/CR possible.

An *object* is an instance of a class: it is the same as what we have been calling the "Entity" in EAV. An *Object Dictionary* is a central registry of all objects in the database, irrespective of the class to which they belong. More precisely, it is a *single table* (which I designate *Objects*) that records *summary* information on every Object. This information consists of an Object ID (surrogate primary key), Preferred Object Name, Brief Description (definition), Class ID (the class to which it belongs), Date-time Created and Date-time Last Modified.

The Object ID is also recorded in other tables where the actual details of the class are stored. For conventional and hybrid classes, which utilize physical tables with columnar attributes, it also serves as the primary key: for EAV tables, the Object ID forms the "Entity" column.

Every class in the system that represents data gets its Primary Key from the Objects table. When a new Object is created, a row is *first* inserted in the Objects table. The primary-key value is then used to create a physical or simulated record that stores the detailed data for that object. In other words, the Object ID serves as a *universal primary key* for all data tables: a given ID is unique across the entire schema.

Based on your needs, the Objects table may also contain the ID of the User who created/owns the object. It is also useful to have a Current Version Number, an integer that begins

at 1 and keeps incrementing each time changes have been made to the Object's contents.

The benefits of a central Object registry are:

- It allows the creation of hybrid classes that share the general-purpose EAV infrastructure, which uses data-type-specific EAV tables. Recording sparse attributes for a particular object is simply a matter of creating entity/attribute-value triples in the appropriate EAV tables, using that object's Object ID.
- As I've stated earlier, for performance reasons it may be necessary to change the representation of a class from EAV to conventional. The existence of an Objects table simplifies such inter-conversion somewhat. It is possible to write generic code, controlled by a point-and-click interface, which consults the metadata and performs conversion automatically. In such cases, because the primary key – the Object ID – *remains* the primary key, you do not need to worry about the conversion failing because of accidental creation of duplicate records that violate primary-key uniqueness.
- In databases of diverse content whose role is to provide browsing functionality for a large community of users (such as the NCBI family), this approach can form the basis of a stripped-down Google-like search interface, where one can search for an object of interest by keyword without needing to specify which class it belongs to. As I will detail in the next chapter, which deals with descriptive metadata, bioscience is a field that is rife with synonyms. "Vomiting" and "emesis" mean the same thing, and the opiate painkiller known as Demerol in the US is called Pethidine in countries with British influence.

To facilitate such search, a **Synonym** table is related many-to-one to the Objects table, containing the synonyms for each object. (Because an Object may have multiple names, the Object Name column in Objects stores the Preferred Name.) Even if the host RDBMS does not support full-text indexes, this table can be indexed to provide a starting point of search for Web-based browsing applications.

In Fig. 11.1, I've included a bare-bones Synonyms table design.

There is one potential performance problem. The Objects Table is a "hot spot" – a table in the database that is frequently accessed by all users almost all the time. For databases that are intended mainly for browsing, where data changes and inserts occur primarily in batch (such as the NCBI databases or the Senselab system), write contention in a multi-user scenario is not a significant issue. Read performance can be optimized for database engines that allow caching of this table in memory. (While the number of rows is large, the number of columns is modest, so caching on modern CPUs with abundant RAM is a reasonable option.)

Of course, every recommendation has exceptions: you don't have to make a fetish out of treating every item of data as an "Object". If you are sure, for example, that a particular class will be represented in its entirety as a physical table without sparse attributes, AND this representation will never change AND it doesn't need to participate in the general-purpose search functionality that you are providing for other classes – the last is a very big IF – then go ahead and designate a primary key for it that is not based on Object ID. The EAV/CR framework allows custom primary-key columns for a class (because information on metadata tables is also captured), and we also store primary-key/foreign-key relationship information in the Relationships metadata table: we might as well take advantage of this capability where it is appropriate.

11.5.2

Representing Sub-structure for EAV Data

In an earlier chapter, I introduced type-specific EAV tables. Using the same approach, it is simple to represent sub-structure through a table called `EAV_Object`, which has the same structure as the type-specific EAV tables except that the Value column contains an `Object_ID` – i.e., like the Entity column, *it is also a foreign key into the Objects table*.

In the user interface, when creating or editing data, it is important to constrain the possible choices that the value column can take. For example, if the value for a particular attribute is supposed to belong to the class Medication, only medications must be accessible. In the user interface, this is enforced either via a pull-down list that shows alphabetically sorted Preferred Names of medications, or (if the list of possible medications is huge) via a user-directed keyword search of the Synonyms table, where the results are restricted to Medication objects only. I'll deal with user interface issues in depth later.

To get the details of the Object referred to in the Value, one would have to locate all rows across all the EAV tables where the Entity column contains this object ID. Thus, one could, in theory, continue with a *recursive* search until all values obtained are atomic. In reality, one does not have to recurse indefinitely. In a browsing application such as SenseLab, for example, when displaying the details of an object, the referred (or sub-) Object as a value is rendered simply as a hyperlink that shows the Object's name (fetched from the Objects table): recursion is under the control of the user. If she wishes to look at details of the referred object, this link can be clicked, so that one navigates only one level at a time.

There is an optimization possible here, which requires throwing more RAM at the problem. Specifically, by caching the *complete* Class/Attribute metadata in memory – this is not far-fetched, because even in a large system, it rarely exceeds 1 MB – you can determine in advance what data types individual attributes belong to, so that you know exactly which EAV tables to access. (For example, if none of the attributes is a Binary Large Object, the EAV table that stores BLOBs or references to BLOBs need not be accessed.) Further, for a given class, you can gather all the IDs of attributes for each data type and store them for future use. This lets you optimize retrieval performance: *rather than accessing the attributes one attribute at a time, you can access the attributes by one data-type-specific EAV table at a time* – fetch all the integer attributes, then all the date attributes, and so on.

11.5.3

Links to Controlled Terminologies

Synonyms are quite abundant in any long-standing field such as biomedicine, as I have stated earlier. Therefore, rather than have your content creators expend effort in creating synonyms for Objects manually, you can utilize the content of large and freely available standard biomedical terminologies, a topic I'll deal with in depth in the next chapter, but which I've alluded to in Chapter 1's subsection on inter-operability.

For this purpose, you have to include (i.e., import) the terminologies' contents within your schema, and provide a linking field (a Concept ID) in your Objects table – Concepts are to terminologies what Objects are to EAV/CR – they are the equivalent of Entities. In

Dr. Marengo's design for Senselab, in fact, such linking fields exist not just for data, but also for metadata – i.e., for the Classes and Attributes tables, because these also refer to biomedical concepts, on which extra information is likely to be present within terminologies.

In our experience, while synonym information is abundant for clinical terms, it is less so for the basic-research applications where EAV/CR particularly applies. This is simply because most research domains, almost by definition, are new enough to have lacked the time for synonyms to arise.

11.5.4

Representing Arrays

We have often found it useful to be able to represent an *array* of objects – that is, a *sequence* of values for the same entity-attribute pair. To implement this capability we add an integer Sequence_Number column to the EAV_Objects table. The Sequence_Number column determines the order of presentation of the individual objects: it may be left null, in which case the objects are not ordered.

For scientific databases, we have found the array capability useful for indexing bibliographic citations by all the objects referred to in the citation text about which we have information in the database. The ability to combine the bibliographic information related to a particular object (e.g., a neurotransmitter) with the structured information also available on it helps to provide a detailed picture about it.

The simulation of arrays is an implementation of what is called non-first-normal form (NF2), which is supported by RDBMSs such as Oracle. In theory NF2 design can be applied to any data type, such as decimal numbers, but we have not have had occasion to use it for anything besides objects. Your needs may be different, so I don't want to be too dogmatic on this point. Go ahead and add a sequence number for all data types if you need it, or omit it if you don't need arrays.

11.6

Data Presentation and Search/Browsing Considerations

The topic of data retrieval from mixed schemas will be explored in depth in Chapter 15, but since such retrieval ultimately relies on detailed metadata, I will introduce the topic relatively briefly here.

When designing metadata-driven interface generation for CRUD operations for individual data classes, it is important to accept that *the interfaces that can be generated automatically must necessarily be somewhat limited*, in that individual artistic license will be sacrificed in the interests of a standard look and feel. While I have not included all details here and in the previous chapter, our metadata for forms (and classes) includes columns for aspects such as fonts/font sizes, foreground and background colors, and other aspects of screen cosmetics, as well as the use of existing Cascading Style Sheets (CSS) for Web pages. Such interfaces may, with good luck, meet 90% of interface needs, but there will

always be some situations where a hand-coded approach – for example, displaying individual data elements in a structure such as a tree or a graph- may be appropriate.

Nonetheless, it is essential to consider the details of what “standard” functionality implies.

- Summary vs. Detail Views: In general, there are two ways to present data from a given class: *List View* and *Detail/Form View*.
 - The List View shows summary information – i.e., a limited number of attributes, typically the most important ones – on multiple records, one per row. A list view is illustrated in Fig. 11.2.
 - The Detail view shows one record at a time in a form layout: details of related many-to-one classes may be presented in embedded list views. Almost all attributes (except those that the developer intends to hide, such as surrogate primary keys) are shown. Detail View is shown in Fig. 11.3.
 - There is also a third “mini-list” view, showing even fewer attributes, which is used when the user intends to *search for and select* a member of a class, usually via a pull-down or a list box. This last view is used when the number of members is modest – e.g., not more than 100 or so – so that if the list is presented alphabetically, the user can locate a desired item by scrolling.

The ability to perform CRUD operations on an individual class of data depends on an individual user’s permissions, which must be maintained in a table that has the primary key (User ID, Class ID). The user interface accordingly changes dynamically, so that certain elements are present or absent. Deletion is enabled through a “Delete” button on the Detail View, and if you prefer, against each row in a list view.

The detail view allows creation of new data and editing of existing data. For creation, an “Add” button blanks out all fields (setting certain values to defaults), and, depending on the preferences agreed to by the software developers and users, the appearance of certain elements may change: for example, elements based on choices may change into list boxes. The changes can be accepted by clicking a “Save” button or rejected via a “Cancel” button. Editing data works similarly, except that, unless the developer desires the interface to change to facilitate data entry, the user should not be forced to switch to an Editing mode. All of these buttons are ideally placed in a common toolbar, as shown in Fig. 11.3.

The Detail View can also server as the basis for a Query-By-Form functionality in order to retrieve data of interest. Data retrieval is a topic with so many ramifications that I will discuss it separately in Chapter 15.

- Drill-down interface: The typical user-interface metaphor is that the user first searches for class members of interest. Sometimes, rather than specifying an individual class first, the user may perform a Google-like keyword search for an object. A list view of the retrieved objects is presented, and the user can then zoom in (drill down) on a selected object, switching to Detail view. The user can similarly zoom in to the details of any many-to-one objects that are in the Detail view.

For a class with a large number of attributes, it is typically desirable to present only a very modest subset of those attributes in list view: typically, just enough information to identify an individual object uniquely. It is therefore necessary to indicate which attributes serve such a role (using a Boolean flag). In detail view, of course, one would show all the attributes (except for those that have been designated to be invisible).

	EVENT	FORM	DATE ENTERED	DATE LAST MODIFIED	PROBLEM	STATUS	FORM IDENTIFIER	Print
View/Edit	STS, AKI Adjudication, Death	AKI Adjudication Form	5/15/2008 10:11:21 AM	11/17/2009 11:54:24 AM	None reported	Verified		Print
View/Edit	STS, AKI Adjudication, Death	Adult STS Data Collection Form (shortened form)	11/4/2007 3:00:30 PM	11/17/2009 11:54:00 AM	None reported	Verified	532729	Print
View/Edit	STS, AKI Adjudication, Death	Medication Inventory	1/11/2010 2:01:53 PM	1/13/2010 2:01:23 PM	None reported	Completed	000000856004	Print
View/Edit	STS, AKI Adjudication, Death	Mortality Form	11/4/2007 3:00:30 PM	11/17/2009 11:54:41 AM	None reported	Verified		Print
<< < > >> Number of records fetched: 4 Showing records: 1 - 4								

Fig. 11.2 A screen-shot of data presented in a List View. It is useful to encapsulate list-view functionality in a reusable object (control), that provides standard functionality such as showing the number of records retrieved (*bottom left*) and allowing navigation when the number retrieved is more than can be displayed on-screen; sorting the contents of the view on a column (by clicking on a button that contains the column name); printing the details of a selected item; and zooming into the Detail view for a selected item by clicking a “View/Edit” button

Tracking System

HELP

Search

Add

Save

Delete

Cancel

Printable Version

List View

Return to main Samples page

3

SAMPLE ID	1905	PATIENT ID	117113
			Search
SENDER SITE	Yale Oncology	SENDER	
SHIPPING METHOD	Federal Express	TRACKING NUMBER	

SENDER INFORMATION

SENDER SAMPLE ID:	12345	DATE/TIME COLLECTED	1/14/2007
SAMPLE TYPE/TISSUE SOURCE:		Blood, whole	

Fig. 11.3 A screen-shot of data presented in detail view. The image is truncated for reasons of space. A toolbar on the top of the screen allows access to common functionality: search (which clears all the data fields and allows Query by Form), Add (which also clears all the fields and then sets some of them to default values, allowing a new record to be inserted); Save (which saves edits), Delete (deletes the current record after confirmation), List View (switches back to the previous list view display, allowing navigation to other records), Printable Version (creates a hard-copy version of the form’s contents), Cancel (abandons edits without saving) and a Help hyperlink that displays form-specific as well as general help

- **Order of Presentation:** Irrespective of the view, attributes need to be presented in a logical sequence, which is not necessarily the same as the physical order in which columns may have been defined in a physical table. The desired sequence of presentation is indicated with an integer sequence number attribute. Sequence numbers need not be consecutive – in fact, to minimize re-editing efforts, it is preferable that adjacent attributes’ sequence numbers have a large gap, e.g., 1,000. This allows repositioning of attributes (which occurs fairly often) simply by changing an attribute’s sequence number so that it falls between the sequence numbers of the two attributes between which it must be placed.
- In theory, one may want to present selected attributes in a different order depending on whether they are presented in list, detail or mini-list view. If you need such a capability, you can create separate columns that record the sequence number for each circumstance: my own experience has been that this is overkill.

- **Grouping Attributes for Display:** When displaying a large number of attributes in Detail view, there are a number of strategies that can be used. One approach would be to lay out successive attributes consecutively, with a user-defined number of columns per row. The desired number of columns is also stored in the metadata for the class: obviously, this can be changed as needed. However, it is desirable to try to group these attributes into labeled logical categories: this allows introduction of breaks with section headings (or, if you want to get fancy with your interface generation, segregation into separate pages of a tabbed interface).

Note: While it is possible that the generated interface – which is typically a Web form - might not fully meet your user's needs and require some hand-customization, try to make your interface-generation framework sufficiently robust that such customization will be the very rare exception rather than the rule. A robust interface implies features such as cross-field validation and skip-logic. If the metadata changes and you have to regenerate the form again, the hand-made edits will be lost, and so the user has to resort to workarounds such as saving the older form under a different name and copying and pasting between the old and the new.

An approach not to be emulated is that employed by Microsoft Access's Columnar Form Wizard, which generates a completely unusable display that requires extensive manual customization. Access lays out fields in multiple vertical snaking columns, with each column having nine rows: with numerous columns, the generated display runs off the screen, forcing horizontal scrolling. The Wizard does not even adjust for the width of the captions that are placed to the left of each field, so these are invariably truncated and need manual resizing. A usable wizard would be cognizant of screen dimensions, and/or prompt the user for the number of fields to be placed per line, with a default setting that auto-computed this number.

- **Display mode vs. Editing Mode:** The appearance of an attribute may depend on whether the data is being browsed or edited. For example, an attribute like PubMed ID discussed earlier would be presented as a hyperlink in Display Mode, so that the user could look up the abstract in PubMed. In editing mode, however, it would need to be presented as a textbox so that the user could enter the integer that is used by the hyperlink template.
- **Searching for an Object of Interest:** In cases where an attribute's value happens to be an object belonging to a class, it is important to consider how finding an object of interest can be facilitated when data is being created or edited.
 - One may present the class members *alphabetically sorted* in a pull-down menu if their number does not exceed, say, the vertical length, in rows, of the computer screen (though the appropriate number would depend on the font and the field's screen position).
 - If more numerous, one would want to present a standard search interface. The contents of the pull-down list would show one or more attributes, but the list of attributes would be even more restricted than those presented in list view. Again, only attributes that assist selective location in a pull-down should be used.

- For a general search interface, modern RDBMSs allow, in addition to *complex Boolean searches* (which I'll discuss in a later chapter), *keyword-based* (Google-style) searches that utilize full-text indexes that are populated with the contents of multiple columns (typically textual) from a class or table. Flagging such columns in metadata allows the creation of such indexes.

Such indexing technology is increasingly powerful: it understands punctuation so that text can be broken on word boundaries: it utilizes stemming (to root forms – e.g., “child” is a stem of “children”); noise words can be eliminated from the index, and it can utilize a language-specific thesaurus. For the English language, several stemmers are available: the most venerable of these is the freely available Porter stemmer.

11.7

The User Interface for Mixed Schemas: Self-mapping

In a mixed schema, a given class of data may be organized conventionally, simulated entirely with EAV, or represented as a hybrid class. A generated user interface must be able to support basic CRUD operations by dynamic generation of the appropriate SQL. The technique that I describe is an extension of the approach described in [Sect. 5.2](#) of Chapter 5. It relies on the technique of *self-mapping*: naming individual data elements on a form using a convention that allows straightforward determination of the element's representation in the database.

11.7.1

Self-mapping for EAV/CR Designs

In a Web page that is based on the EAV/CR design, each element has a unique ID that is a delimited combination of the following: Class ID, Attribute ID, Data Type, Representation, Instance Number and Sequence Number.

- Representation (i.e., Physical, EAV, Hybrid) and Data Type can be encoded as single letters. While you can always determine the representation and data type for a given class/attribute pair by consulting the metadata through a SQL statement, it is simpler to use these in the ID and save a lookup, since you need these so often.
- Instance number is non-zero only for elements that are used for simulated sub-forms (i.e., embedded List Views) that present many-to-one data.
- Sequence Number is non-zero only when your attribute is an array type. Based on your design, you may not use Sequence Number at all, or use it only for objects, as we do.

This information is not enough to allow us to generate SQL for CRUD operations. We need the following extra information:

1. For every class that happens to be a physical table, we need to know the list of primary key fields, as well as the data type of each primary key field. Remember that updates and deletes of the main “One” class in the form must specify the primary key expression using SQL syntax.

2. For classes that represent “Many” data, we additionally need to record the foreign-key expression that links the “Many” table to the “One” table. (In general, when a table in a database is likely to participate as a “One” table in a relationship with another table, it is considered good database design to create for it a *single* primary-key column, if necessary by using an Identity/auto-number column. It is also considered good design to replicate this column name in the “Many” table to serve as a foreign-key column: sometimes, however – as in cases where there is a double relationship between the same pair of tables – the foreign-key column will be different.)
3. For Attribute IDs that correspond to physical columns, we need to store the physical column name.
4. For each class that already contains previously instantiated data (where the user might want to perform update or delete operations) we need to store the primary-key column values for each instance. (For EAV tables, or for hybrid classes, these values will be object IDs.) While points 1-3 above are metadata, this last is data.

While all of the above information can be looked up as needed by issuing SQL statements against the database schema, in my experience it is needed so often that it is best cached in memory, to be referred to as needed. Ideally, the information is stored in the form of hash tables. In the case of metadata, the hash tables are keyed by class ID and attribute ID, and loaded at application startup. (On today’s hardware, the space requirement is extremely modest: frameworks such as ASP.NET allow these hash tables to be stored as “application variables” that are accessible by all users of the application.) In the case of data, primary-key-column values are initialized when the form is opened: these values are indexed by a combination of class ID and instance number.

11.8 Generating SQL for CRUD Operations

In this section, I’ll deal only with Create, Update and Delete. The topic of Read – data retrieval – is complex enough to require its own chapter. It also turns out that read operations, despite the use of frameworks, are particularly hard to automate entirely, because the users’ needs are extremely problem-specific and so mostly unpredictable.

11.8.1 Fundamental Security Considerations: SQL Injection

No software should naively trust whatever its users enter into text fields, whose contents would be inserted into dynamic SQL. I have discussed the vulnerability of the Eval statement Chapter 5. When composing SQL dynamically, you must be aware of the possibility of *SQL Injection*. This is an attack where an attacker enters SQL in a field where ordinary text such as a user name would be expected, with the objective of gaining unauthorized access, or damaging the database contents. The following example is fairly typical. For an excellent in-depth discussion on this theme, see Steve Friedl’s article.⁹

11.8.1.1

Obtaining Unauthorized Access via a Login Screen

Most web-based database applications do not provide direct database logins for individual users. Instead, the application connects to the database with a fixed “service” account that does not actually correspond to any user. This allows greatly improved performance using “connection pooling”: a single connection is multiplexed to serve a much larger number of users. Here, a given user is not continuously connected to the database: the connection is established only at the moment a service is requested of the database – to fetch or change data – and released for reuse immediately after the few milliseconds it takes for the service to execute.

Within the database, user access is controlled by maintaining a list of user-names and passwords; the latter are typically stored encrypted. The login screen prompts for a login name and password. When the user fills in both fields, the password is encrypted by the application, and the software validates the user with a SQL statement like:

“Select count(*) from users where LoginName=’” & *text of login-name field* &
“and Password=’”& *encrypted-password-text* & “”

(Note: In SQL, string literals are enclosed within single quotes.) The intention of the code is that if the count of the users with the given user-name and password is equal to zero, access is denied.

The knowledge that SQL is composed this way (which can be determined by probing the system with a variety of inputs and inspecting the resultant error messages – see Friedl’s article for details – can be used to secure unauthorized access as follows. Instead of entering a user name, the attacker enters the text:

‘ or 1=1 --

The double-dashes are used in SQL as the symbol for inline comments – anything after the symbol is ignored. If this text were inserted verbatim into the above statement, the result would be

“Select count(*) from users where LoginName =” or 1=1 -- ...”

The comment symbol causes the part of the statement that checks the password to be skipped, and treated as a comment. Since the condition “or 1=1” must always be true, the condition “LoginName=” (the empty string) is simply ignored, and the count returned is non-zero.

11.8.1.2

Defenses against SQL Injection

Prevention of SQL Injection attacks involves several approaches, which must be employed *in combination*.

- Validating the input based on string length, data type, and regular expressions that identify permissible characters. This is the first line of defense: the EAV/CR metadata framework supports these validations, so you must utilize them maximally.

- Disallowing particular characters in specific fields: the software should disallow spaces in fields such as E-mail addresses, login-names and passwords.
- Escaping the single quote character: The single quote symbol should be replaced with two single quotes. Even if injection attacks did not exist, this transformation is necessary for all fields where this symbol is valid input. (Note: the doubling-of single quotes with your own symbol-substitution routine will not work for RDBMSs like MySQL, which allow alternative syntax for escaping single quotes such as `\``. MySQL, however, has a built-in function called `mysql_real_escape_string()` that safely escapes special characters such as single quotes and newlines.)
- Using parameterized stored procedures where applicable. Here, the user-name and password field contents would be passed as parameters to a routine that authenticated the user. Such routines *do not use string concatenation* to build a SQL string that is executed. The purpose of dynamic SQL is to eliminate the creation of large numbers of stored procedures: however, for tasks such as user authentication, the security that stored procedures afford makes them infinitely preferable.
- Limit the capabilities of the Service Account: The service account should not be allowed to do anything that an end-user would not be allowed to do (such as dropping a table). In other words, it operates with the least level of privilege necessary to function. This way, even if there is a breach, the potential damage is minimized.

11.8.2

Implementing the Delete Operation

Let's first consider the Delete operation. The primary-key-value for the current record is accessible, both in List and Detail view. Therefore, performing a delete simply requires issuing the SQL statement "delete from <tablename> where <primary-key-expression>" for a class represented conventionally in physical tables. For data represented in EAV structure, it is not that much more complicated. Remember that in EAV, a "record" is simulated by multiple physical rows, in multiple physical EAV tables, that share a common grouping – specifically, they share the same Object ID. If you have set up a Cascading Delete relationship between the Objects table and the individual EAV tables, the operation "Delete from Objects where Object_ID= <current-object-id>" will clear up all the dependent records.

You will generally maintain an audit trail table that tracks your deletions, as mentioned earlier, and you will also implement DELETE triggers on each of the involved tables. These triggers will first copy the contents of the records to be deleted to an audit table before deletion is attempted. All of these operations will be enclosed in a transaction.

11.8.3

Implementing the Update Operations

One important difference between EAV-structured data and conventional data is that, if you update a single record in a physical table, many RDBMSs (such as MS SQL Server)

implement the update as an insert followed by a delete. Therefore, from the disk-performance issue, updating multiple columns of a single row is not significantly less efficient than updating a single column. In other words, a framework that generates update code can be naïve and update every single column in a form, without needing to track which columns the user actually changed. For EAV-modeled data, however, since multiple physical records from multiple tables simulate a logical record, the naïve approach would be needlessly expensive.

Therefore tracking the user's changes is necessary so that the SQL updates only what actually needs to change. This can be done by one of two methods:

- Loading the initial contents of every field in a form into a hash-table in memory. When the user attempts to save the form's contents, you iterate through the fields, comparing each field's current value with the previously stored value to determine what has changed.
- Creating an "event handler" for each field's Change event (Web browsers support an event called *onChange*). Among other things, the handler flags the field as changed and stores its new value in a hash table that grows only with the number of fields that the user actually changes. This event handler must actually do several other tasks, including validating the changed content based on data type, range checks, regular expressions and so on. The list of changes – the "deltas" – is now used to generate the necessary SQL.

I prefer the first approach when using .NET. I use event handlers, but I don't try to make them handle deltas.

11.8.3.1

Practical Points

- Primary Key Fields: You must track changes to the Primary-key fields since the primary-key expression will be part of your SQL. In the case where the primary key field is a surrogate key (and therefore immutable), such tracking is not necessary.
- Segregating physical-table operations from EAV operations: In an update operation on a hybrid-class object, some fields on the Web form correspond to a single row of a physical table, while others correspond to separate EAV-modeled rows. Using the information in the attribute metadata, it is straightforward to separate the former from the latter. Operations on a single row can then be combined into a single UPDATE statement, while those on EAV modeled rows must be separate statements.
- Updates on EAV attributes are more complicated than they seem: There are three possibilities with an EAV-modeled attribute.
 - If the initial value was empty, and the new value is not empty, the SQL operation needed is really an INSERT for that row. (Recall that in EAV, we don't store empty values.)
 - If the old value was not blank, and the new value is empty, we need to generate a DELETE statement.
 - Only if old and new values are not blank (but different) is an UPDATE statement needed.

11.8.4

Implementing the Create Operation

The Detail View form is used both for data updates as well as data creation. To implement this, the form has a Mode variable, which can be one of Browse (if no edits are allowed), Insert, Update, or Query: I'll discuss the last mode in a subsequent chapter on data retrieval. The user switches into Insert mode by clicking a button (typically labeled New Record or Add), which causes all the form's fields to be either blank or set to default values. As individual fields are filled in, each field's event handler validates what is entered, and finally, when the user clicks on a Save button, cross-field (complex) validation kicks in.

The Primary Key field contents are either entered by the user, or more commonly, auto-generated from the database's surrogate-key mechanism. As in the case of Updates, inserts for hybrid classes must also segregate the operations on physical tables from EAV operations: the fields based on physical tables must be grouped so that a single INSERT statement is generated.

11.9

Selective User Access to Individual Classes or Attributes

Based on your needs, you may also need to implement a security model where access to individual classes, or individual attributes within a class, is *restricted* by user. I'll provide details in Chap. 15 but I'm mentioning it here because security support must be part of your metadata design.

Rather than defining such permissions at the user level directly, it saves significant labor to define such permissions based on *roles* rather than on individual users. In general, you want three levels of permission: None, Read-Only, Read-Write. To avoid specifying permissions *ad nauseam* for every role, you may also want to implement a default strategy, whereby if permissions are not explicitly specified for an individual role, they are assumed to be either granted or denied. Your choice of default will depend on the nature of your application and your level of paranoia. For attributes, however, it may be assumed by default that most roles that have access to a class will access all attributes: any attributes explicitly specified would therefore form an exception list.

The topic of subject-matter areas discussed earlier also bears on security, though this is as much a means of preventing cognitive overload as of preventing impermissible operations.

11.10

Case Study: Representation of Phenotype

The following case study is meant to illustrate the process of analyzing a problem and tailoring the metadata design appropriately, using EAV modeling as appropriate.

So far, when implementing metadata tables, I've used conventional structure consistently, because the information needed to be captured – typically concerning validation and presentation – has a regular structure.

While there are minor variations (e.g., validation specific to a particular data type, such as regular expressions), these can still be modeled within a single table and implemented in the user interface through skip logic. However, in uncommon circumstances EAV representation is appropriate for at least some metadata. This occurs when we model *phenotype* in a generalizable fashion.

First, some definitions for non-geneticists:

- **Genotype** refers to an individual's genetic constitution: this is normally studied in terms of the composition of the DNA at a specific position (or *locus*) in the genome that determines the expression of a specific trait. (When we refer to an "individual", it need not be a human being – any living creature can manifest a genotype.) Variation in genotype leads to variation in function as expressed at molecular, cellular, organ and system levels.

To represent genotype computationally, one starts with a DNA sequence at that locus (a series of letters belonging to the set A, T, G and C), and represents a given genotype as a variation from this sequence – substitutions, additions or deletions.

- **Phenotype** refers to "the outward, physical manifestation of internally coded, inheritable information" – something that is observed as the *consequence of genotype*. For example, the sickle-cell mutation in the Hemoglobin molecule (common in African Americans) can result in sickle-cell disease. The red blood cells become sickle-shaped and abnormally fragile in conditions of low oxygen availability: they are continually destroyed, resulting in an anemia; and the abnormal cells block small blood vessels all over the body, resulting in a large variety of symptoms. See the Wikipedia entry for this condition.

In practical terms, the phenotype for a given condition is defined by a set of parameters that can be characterized at the clinical, cellular or molecular level, which are *specific to the genetic condition being studied*. Across all genetic conditions of interest even in a single species, the total number of parameters would range in the hundreds of thousands.

Multiple genes are involved in the regulation of observed phenomena such as blood pressure and blood glucose. That is, abnormalities in a given phenotypic parameter may be due to multiple genotypes. Therefore further characterization of the individual becomes necessary. The workup of a new case of high blood pressure, for example, involves analysis of parameters such as endocrine function (e.g., adrenals, pituitary, thyroid), kidney function, and so on: as science progresses, the characterization of phenotype becomes progressively refined. Another challenge is to distinguish environmental effects from genetic: for example, chronic ingestion of particular environmental toxins (e.g., lead) and particular food (e.g., liquorice) would cause high blood pressure in any person irrespective of genetic predisposition.

11.10.1

Representing Phenotypic Data and Metadata in a Repository

The problem of representing phenotypic data in a database that needs to store information on an arbitrary number of phenotypes is somewhat similar to the problem of managing

clinical data: the extreme heterogeneity mandates an EAV representation for the data. However, the metadata problem is significantly more complicated as well, because we are storing not just clinical information but also preclinical information.

Therefore, the definition of the parameter in Chapter 5 must be significantly augmented, notably in the descriptive information. Rather than have the meaning of the parameter described only through its internal name, caption and a textual “Description” field, the individual aspects of the description must be atomized into separate attributes. The nature of the attributes depends on the category of parameter. For example:

- For parameters that are based on lab tests, one must record information such as the source of the biological sample (blood, urine), timing of the sample (random vs. post-lunch), units of measurement, range of normal values, and a bibliographic reference to the measurement method.
- For clinical observations, one must record the conditions of observations: body site, position of subject (sitting vs. standing), and the device used for measurement.
- For test-tube-based enzyme assays, the details of the assay method – chemicals used and their quantities, conditions of assay (temperature, pH etc.) – must be recorded.
- When one goes on to characterize molecules (such as sickle-cell hemoglobin) experimentally, each characterization technique comes with its own set of descriptors.

The reason for describing parameters in what may seem to be excruciating detail is that research on phenotype must be *reproducible*. Scientific journals insist on such details before research results are considered publishable: the challenge, from the database perspective, is to structure such information maximally rather than store it all as narrative text (though one may need to fall back on some text anyway).

The descriptive attributes (which I will call *Descriptors*) vary based on the category of the parameter: however, there is usually broad agreement about what Descriptors should be present for a given Category in terms of what order the descriptors should be presented, and which descriptors are mandatory or optional.

In terms of the metadata schema, one can support the above requirement through the following additional tables:

- **Categories** (Category ID, Name, Description)
- **Descriptors** (ID, Name, Description, Datatype, Choice_Set_ID). The last, which is permitted to be null, is used for enumerations.
- A bridge table, **Category_Descriptor** (Category ID, Descriptor ID, Presentation Order, Is Optional), records what descriptors apply to a given category.
- The **Parameters/Attributes** table has an additional column, Category ID, indicating the category that it belongs to.
- **Parameter_Descriptor** (Parameter ID, Descriptor ID, Value) records the descriptors of each attribute. For example, the parameter “Serum Sodium” could have the Descriptor-Value pairs: (Units of Measurement: “mMol/L”), (Lower Limit of Normal, 135), (Upper Limit of Normal, 145).

Note that the Parameter_Descriptor table is for practical purposes an EAV table, except that the “entity” is the Parameter, and the “Attribute” is the Descriptor. Given that the

values have different intrinsic data types, one could segregate this information into type-specific tables, but this is probably overkill (I haven't had the need to query on value so far), and the type information is best used for validation of data entry in the user interface only. The possible descriptors that are applicable for a given Parameter are constrained by that parameter's Category (using the information in Category_Descriptor), so as to eliminate the risk of inapplicable descriptors for a given parameter.

Appendix 11.1: Metadata Schema Details

The text in this section comprises a detailed annotation of Fig. 11.1.

Meta_Classes

- **Class ID:** Surrogate Integer Primary Key.
- **Class Name, Caption, User Description, Developer Description, Datetime_Created, DateTime_LastModified:** Mostly self-explanatory. User Description is presented in automatically generated end-user help. Developer Description contains developer-oriented documentation. For physical tables, the Class Name is the physical table name.
- **Class Type:** One of: View, Physical Table, EAV, Hybrid.
- **Row-level security required:** If true, every row in a class instance is stamped with the user who created it - only this user can alter a given record, others can only look at it.
- **Primary Key Expression:** The comma-separated list of fields that comprise the table's primary key.
- **Is Autonumber Primary Key:** If true, the primary key is based on an auto-numbered artificial key.
- **Current Version Number:** An integer incremented each time the class is modified.

The Meta_Attributes Table

Contains one entry for every field in every table in the TrialDB schema (including itself), so as to make the database self-describing. Note that many of the columns that describe an attribute are identical in function (and therefore, have the same name, as the validation/presentation-information columns in the Parameters table of chapter 5.

The columns are described below.

- **Attribute_ID:** Artificial Long Integer Primary Key.
- **Attribute Name, Caption, User Description, Developer Description, Datetime_Created, Datetime_LastModified:** As for Meta_Classes.
- **Attribute Representation:** This value is editable (and consulted by the software) only if the attribute's parent class type is Hybrid. Can be one of: Physical Column or EAV.

(If the parent class type is a Physical Table, then a representation of Physical Column is implied: if EAV, then an EAV representation is implied.)

- **Class Id:** Foreign key into the Meta_Classes table. Refers to the Class of which the Attribute is a member.
- **Data type:** The logical data type of the attribute (e.g., integer, decimal, string, class, etc.).
- **Schema_datatype:** For attributes that are physical columns, the data type as defined in the DBMS (e.g., varchar(255)).
- **Physical_Order_Sequence_Number:** For physical tables and views, the position of the column within the table.
- **Sequence_Number:** The order in which this attribute is presented when inspecting members of the class. This may be different from physical order.
- **Attribute_Class:** This is not null only if the Attribute's Data Type is Class (and the attribute's data is represented in EAV form): it records the ID of the Class to which the referenced object must belong. In such a case, data for this attribute is stored in the EAV_Objects table.
- **Attribute Group:** A string that serves as a grouping label. Attributes with the same label are placed on a page on a multi-tabbed section of the form: the string also doubles as the label on the tab. If no string is supplied, the attribute is placed in a non-tabbed section of the form, so that it is constantly visible.
- **Present_In_Listview:** If true, the attribute should be shown in List view.
- **URL Template:** a string used to generate a hyperlink to an external data source: see the discussion of PubMed ID earlier. The placeholder in the template (where the value of the attribute is inserted) is indicated by a vertical bar. If the URL Template is not null, when the attribute is in display mode, it is displayed as a hyperlink.
- **Class_Select_Method** How are instances of the class accessed- by pull-down menu or through a search form? Use the latter if the number of instances is large.
- **Part_Of_Search_Fields** If true, this attribute is shown if the class is displayed in a pull-down or list box if a class member is to be searched for and selected.
- **Searchable:** Is the field important enough to be commonly searched by? If so, its content is incorporated in a full-text index for that class.
- **Multi_Instance** In our own design, this is applicable only when the data type is Object. If true, this allows an array of objects. For multi-instance attributes, we also record the integer values of **Minimum Repeat Count** and **Maximum Repeat Count**, to specify the minimum and maximum number of times this particular attribute must occur. (If not specified, the number of occurrences is indefinite.)
- **Is_Stored_Encrypted** If true, the field is stored encrypted. Certain fields related to Personal Health Information are stored this way: they are decrypted/encrypted in the middle tier.

Choice_Set_ID, Mandatory, Default_Value, Upper_Bound, Lower_Bound, Upper_Bound_Warning, Lower_Bound_Warning, RegExp, RegExp_Error_Message, Allow_Future_Dates, Conditional_Choice_Set, Max_String_Length, BLOB_File_Extension, Max_BLOB_Size_KB, Computed_Formula, Display_Width, Display_Height, Orientation, ReadOnly, Invisible: Validation and Presentation-related columns. See the description of these columns in the narrative accompanying Fig. 5.2 of Chap. 5.

(Most of these are far more important than some of the fields described earlier: in the meta-data- browsing interface, several of these would be presented in list view. I've just placed them at the end to avoid repetition.)

Appendix 11.2: Alternative Means of Representing EAV Data

Using an XML Column for Sparse Attributes

For the special case of hybrid classes, the database/XML guru Dejan Sarka, in the very instructive text “Inside Microsoft SQL Server 2008: T-SQL Programming” (Itzik Ben-Gan et al., Microsoft Press) illustrates that XML can be used as an alternative to EAV. This is not surprising: XML is a kind of attribute-value data representation that is based on structured text rather than relational tables. The high-end RDBMSs support XML as a native data type (including the ability to associate a specific XML schema with a specific column for validation purposes).

Sarka's solution depends on first defining a temporary table with all the columns that you would consider using EAV for, associating each column with the necessary validation constraints (which are defined using your vendor's dialect of SQL) and then using your database's XML-schema-generating capability to define an XML equivalent of this table. Then this table is dropped, and the physical table that forms the core of your hybrid class is modified by adding an XML column, whose associated schema is then based on the one just generated.

This solution is undoubtedly much more CPU-efficient than the EAV approach for displaying the content of a single object – it is not scattered in multiple type-specific tables, but is right there in the same row of the physical table. While the contents of an XML column can be indexed, the indexes are nowhere as efficient or compact as those for traditional columns, so that in circumstances where cross-object query based on the sparse attributes would be important, it would arguably be less efficient.

However, the XML is *only an alternative storage mechanism*: this solution does *not* bypass the need to describe the sparse attributes in metadata. It does not solve the user-interface generation problem (including presenting the data in a friendly fashion and allowing its editing through forms). By the same token, without metadata, it does not address the issue of validating the user's input and providing end-user-comprehensible error messages (as opposed to the non-informative computer-ese that would be emitted by default if the XML content failed validation).

A programming issue that you need to be aware of is that, when you are retrieving heterogeneous data using a SQL generator that considers how individual classes are represented, you will have to generate XQuery code instead of SQL. Code generation may be considerably more complicated unless you use a standard design approach for structuring your XML. Further, individual vendors' implementations of XQuery vary with respect to which features of the XQuery 1.0 standard are/are not supported. Finally, while an implementation will allow you to access the contents of non-XML columns in your XQuery code, the functions that will let you do so will necessarily be vendor-specific.

Microsoft SQL Server Sparse Columns

Another approach to modeling hybrid classes is Microsoft SQL Server 2008s *sparse column* mechanism. Columns with a basic (string, numeric, datetime) data type (e.g., numeric, varchar or datetime columns) can be designated as sparse: up to 30,000 columns per table can be created this way. Sparse columns take up zero space for NULL values and only rows containing values are indexed. When listing the contents of a table containing sparse columns with a “SELECT * ” statement, all sparse columns’ contents are concatenated into a chunk of XML, where each value is sandwiched within open-and close-column-name tags.

Once again, as in the case of XML, sparse columns are *only a storage mechanism*. Do not delude yourself into thinking that they will save you the task of defining metadata: as a matter of fact, SQL constraints (i.e., any validation other than type checking) *cannot* be defined on sparse columns (simply because Microsoft says so). Unless you define such constraints in metadata, and implement them in software, using sparse columns “as is” is an invitation to trouble. In fact, I consider the current implementation of sparse columns as a poorly-conceived design trap for the naive.

References

1. PostgreSQL PostgreSQL documentation. Available from: www.postgresql.org/docs/. [cited 10/5/10], 2010.
2. World Wide Web Consortium. SPARQL query language for RDF. Available from: <http://www.w3.org/TR/rdf-sparql-query/>. [cited 11/1/10] 2008.
3. Nadkarni PM, Marengo L, Chen R, Skoufos E, Shepherd G, Miller P. Organization of heterogeneous scientific data using the EAV/CR representation. *J Am Med Inform Assoc*. 1999;6(6):478-493.
4. Nadkarni P, Marengo L *Easing the transition between attribute-value databases and conventional databases for scientific data: Proceedings of the AMIA Fall Symposium*, Washington, DC, Hanley & Belfus; 2001:483-487.
5. Marengo L, Tosches N, Crasto C, Shepherd G, Miller P, Nadkarni P. Achieving evolvable web-database bioscience applications using the EAV/CR framework: recent advances. *J Am Med Inform Assoc*. 2003;10(5):444-453.
6. Marengo L, Nadkarni P, Skoufos E, Shepherd G, Miller P. *Neuronal database integration: the Senselab EAV data model: Proceedings of the AMIA Symposium*; 1999:102-106.
7. Kimball R, Caserta J. *The Data Warehouse ETL Toolkit*. New York: Wiley Computer Publishing; 2008.
8. Slezak T, Branscomb E. *An integrated browser for chromosome 19 physical mapping data: The Cold Spring Harbor Meeting on Genome Mapping and Sequencing*. Cold Spring Harbor, New York; 1992:220.
9. Friedl S SQL injection attacks by example Available from: <http://unixwiz.net/techtips/sql-injection.html>. [cited 11/5/10] 2010

12.1 Introduction

A *Terminology* is a set of phrases (or terms) that are used to categorize or index (*tag*) content in a particular field of knowledge (for which I'll use the term *knowledge domain* or *domain* for short). This tagging facilitates its subsequent organization, categorization, retrieval and analysis.

The use of terminologies is ubiquitous in biomedicine, and certain biomedical terminologies have become very large, of the order of more than a million items. Their uses include the following:

1. *Documentation of clinical encounters or processes*: items from a terminology are used to record clinical diagnoses, procedures performed on a patient, or the adverse effects experienced by a patient who was treated with a therapeutic agent. Coding of diagnoses and procedures is a pre-requisite for reimbursement of clinical care delivery.
2. *Tagging of Prose*: Narrative text in the domain – this includes clinical text such as progress notes or discharge summaries, or scientific publications – can be tagged using phrases in the terminology. Such tagging was originally manual, being performed by trained human indexers, and in many processes, it still is. However, electronic methods that rely on text processing or *natural language processing* (NLP) are increasingly being employed, and I'll discuss them very briefly later.
3. *Exchange of Data between software systems* by serving as the basis for translation of coded data.

I will discuss the latter two uses in more detail shortly.

The content of a terminology is developed in one of two ways.

- The phrases are carefully selected in advance, usually by an official body comprising a committee of experts, in which case it is referred to as “controlled” to indicate the presence of an official process.
- Users assisted by computer software contribute the phrases to describe the content that they have created.

We'll see that these two processes represent two extremes of a continuum: in practice, the approaches are not mutually exclusive. For example, authors who submit a publication are asked to suggest phrases that would best categorize it, and such phrases may be added to the terminology later after curatorial review. However, in biomedicine, the first approach has predominated.

12.2
Types of Terminologies

At present, the terms Controlled Terminology, Controlled Vocabulary, Thesaurus, and Taxonomy tend to be used interchangeably. However, it is worth inspecting these terms from a historical perspective, so that we can see how the design of these classification systems has progressively evolved. In the terminology types described below, each succeeding terminology type is a superset of the previously described one in terms of feature set. The differences between the terminology types are summarized in Figs. 12.1–12.4, but you should probably inspect these figures in detail only after you have reached the end of this section.

Fig. 12.1 A basic controlled vocabulary. The terms that exist in the vocabulary (only a few are shown) are unconnected to each other with semantic relationships

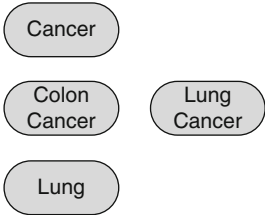
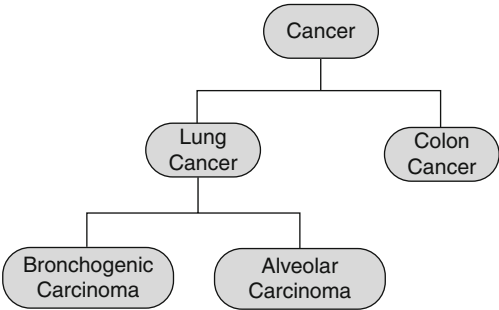


Fig. 12.2 A basic taxonomy. The lower-level (*child*) terms are connected to upper-level (*parent*) terms with “is-a” (general-specific) relationships. Thus, the children lung cancer and colon cancer are specific types of cancer, their parent, and bronchogenic and alveolar carcinoma are specific types of lung cancer. While not shown here, modern taxonomies allow terms to have multiple parents



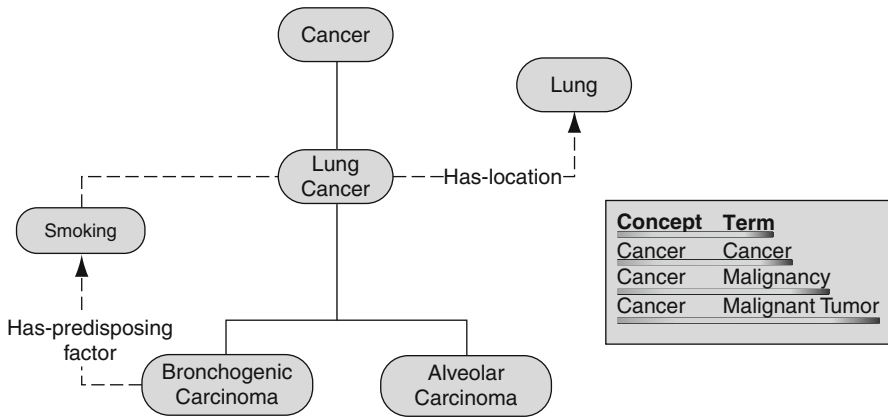


Fig. 12.3 A thesaurus. In addition to is-a links, other types of links exist. These link types are specific to the thesaurus's intended purpose, and vary with the knowledge domain being represented. In addition, there is a distinction between *concepts* (units of meaning) and *terms* (synonyms for a concept): synonyms of *cancer* are illustrated. While synonymous terms could be associated with concepts through "is-a-synonym of" relationships, it is better to place them in a separate table, as above, so that relationships only link concepts to each other

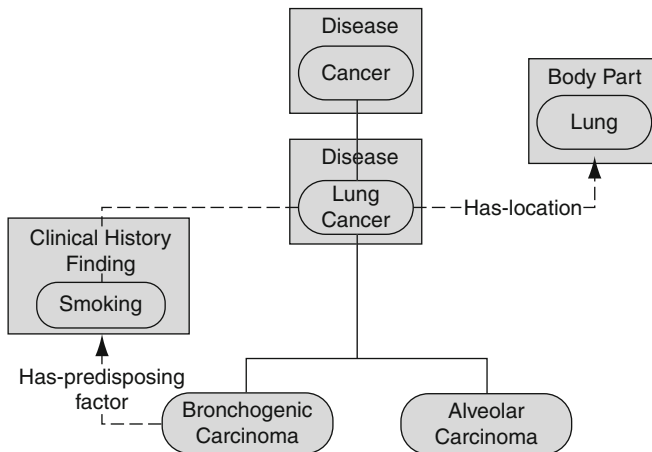


Fig. 12.4 An Ontology. Every concept is categorized into a Class. Descendants of a concept automatically *inherit* from the parent concept. Thus, bronchogenic and alveolar carcinoma must be diseases, and they must also have the location "Lung". Inheritance greatly increases the ability of software to generate inferences or perform "electronic reasoning"

12.2.1

Elementary Terminology Designs

12.2.1.1

Controlled Vocabularies

At the most elementary level, a committee decides to designate a set of key-phrases (terms), called a *Controlled Vocabulary*, that shall be used to annotate and index the narrative-text content in a particular knowledge domain (e.g., scientific publications). This was the approach originally used for the Index Medicus when it was initiated in 1879 by John Shaw Billings of the US Army's Surgeon General's Office.

In a basic controlled vocabulary, no relationships are specified between individual terms, even though such relationships may be inferred by those who are knowledgeable in the domain. For example, the term "infectious hepatitis" is a subcategory of the more general term "hepatitis", which itself is a sub-category of the still more general term "liver disease", but the basic controlled-vocabulary structure would not allow us to record the specific-general relationship between these terms.

12.2.1.2

Folksonomies

The basic controlled-vocabulary approach was intended for use in situations where publications were indexed by trained human beings who had to select terms from the terminology that, in their opinion, applied to a given document. In many new and rapidly evolving knowledge domains, it is unlikely that an expert committee will exist (and if one's needs are immediate, one may not have either the time or resources to invest the effort in convening such a committee). Also, manual indexing, as described above, is increasingly regarded as not cost-effective.

In this case, a software application that allows users to create content will simply allow the users to supply their own key-phrases to index their content. These key-phrases are then added to the terminology. In other cases, the text content will be scanned and words that are extremely common words in the language such as articles, prepositions and conjunctions (these are called "stop words") are removed, and the remaining words used as keywords to index the content.

The resulting "uncontrolled vocabulary" is called a *folksonomy*, a portmanteau word coined by Thomas Vander Wal¹ from the words "folk" and "taxonomy" – and inspired by the German word "Volkswagen" – "People's Car" because its contents are created by ordinary users rather than cognoscenti. (We discuss taxonomies in the next sub-section, but the only thing common to folksonomies and taxonomies, other than the name similarity, is their intended purpose of categorization.)

The folksonomy approach is used by social networking sites such as Facebook and MySpace to index user content. Some of these sites provide instant feedback: when a user provides a term that is already present in the terminology, the website software will instantly show other users whose sites that have been indexed using the same term. This feature provides a simple means of feedback to ensure that the term is not misspelled: if misspelled, the user would get back very few or zero matches for a term expected to be common.

As we'll see later, however, while folksonomies offer a convenient means of detecting the terms of interest in a domain (*vocabulary discovery*), they represent at best a first step that needs considerable refinement to be useful for most scientific domains.

12.2.2

Support of Hierarchies: Taxonomies

At the next level of sophistication, the *Taxonomy*, the individual terms are arranged in a hierarchical or tree structure, with more general terms ("ancestors") higher up in the tree and specific terms ("descendants") lower down. Thus, for example, the ancestor "heart disease" may have the descendant "coronary heart disease", which in turn has the descendant "myocardial infarction".

In the most restrictive kind of design, the structure is a pure (single) hierarchy: that is, a given descendant term may have only one ancestor. In many cases, the individual terms have associated quasi-numeric codes (*identifiers*) that are influenced by the hierarchical Dewey decimal system that is used in traditional libraries, where the descendant terms have the same partial numerical prefix as their ancestors. Many classifications are still stuck at this level, most notably the World Health Organization's International Classification of Diseases (ICD).

12.2.2.1

Drawbacks of the Pure-Hierarchy Design

While several taxonomies, especially the ones that have been long-established, use the pure-hierarchy approach, this design does not model most knowledge domains accurately, and does not evolve gracefully.

- Many terms in reality may descend from more than one ancestor: for example, tuberculous meningitis is both a disease of the brain as well as a type of tuberculosis, but the curators of the classification must arbitrarily designate only one of these terms as the ancestor. In the days when classifications were printed on paper and software that allowed search by partial phrase did not exist, locating a given term of interest was fairly challenging, involving a lot of guesswork and/or memorization. More important, the single-hierarchy design needlessly complicates certain operations on the data that has been encoded using the hierarchy, such as answering queries that involve aggregating descendants, e.g., finding all clinical cases of all types of tuberculosis or all cases of brain disease.
- Other complications involve the use of hierarchical codes. Often, knowledge related to a particular term (e.g., a specific disease) changes dramatically as a result of scientific research. One would like to reclassify the term by moving it to its new place in the taxonomy, but one is required to deal with older data where the old code (which now has incorrect semantics) has been used. The problem of hierarchical codes is a specific instance of the problem of *trying to assign codes with intrinsic meaning*. In numerous application domains, this approach has been repeatedly demonstrated to be extremely brittle. For example, in manufacturing applications where multi-part codes were designed (with each part having a specific meaning), the codes became useless when

the process associated with a particular code-part was rendered obsolete by technological advances or by changes in business needs – these parts now became legacy clutter.

Modern taxonomy design therefore supports *poly-hierarchy*: that is, one may designate as many ancestors for a given term as needed. It also abandons the use of hierarchical codes. In other words, the code, or identifier, associated with a term is *devoid of any intrinsic meaning*: the parent-child (i.e., immediate ancestor-immediate descendant) relationship between a pair of terms is *recorded explicitly* rather than implied through the use of hierarchical codes.

12.2.3

Support of Non-hierarchical Relationships: Thesauri

The next stage of terminology evolution is represented by the *Thesaurus*. Thesauri were originally used (by Roget) to describe the English language, but have been adapted to describe various scientific domains, including medicine. The distinguishing features of a thesaurus are support of the following features:

- In any terminology that has evolved over time, there are likely to be multiple, *synonymous* terms that refer to the same underlying idea. A thesaurus therefore makes a distinction between *concepts* – distinct units of meaning in the domain – and *terms*, which are synonymous forms of concepts. In medicine, for example, terms used by Greek and Roman physicians 2,000 years ago are still in use, as are more common Anglo-Saxon variants. Thus the terms vomiting and emesis refer to the same concept, as do “kidney disease” and “renal disease”. One or more terms for a given concept may be designated as *preferred*.
- The reverse situation may also apply: that is, the same term may sometimes refer to different concepts. This situation is called *homonymy*. Thus, “cold” may refer to a blocked nose (rhinitis) or to cold temperature. The origin of the former term is a vestige of the Hippocratic belief that that the runny nose was a manifestation of an excess of the cold principle (phlegm) in the body. (In ordinary English, homonyms are ubiquitous: – words like “set” and “jack” have more than 20 meanings each.) A special case of homonymy is *polysemy*: here the multiple meanings of a word or phrase are related. An example is *anesthesia*, which can refer to the lack of (general or local) sensation, or to the procedure used to induce lack of sensation.

A thesaurus must therefore record *ambiguous* terms.

- The relationships between a given pair of concepts are not merely hierarchical, as in taxonomies. Other kinds of relationships are also supported. Examples are “part of”: (e.g., the ankle is part of the lower limb), “caused-by” (e.g., AIDS is caused by the Human Immunodeficiency Virus), and so on. The list of relationships depends on the knowledge sub-domain: relationships that are useful in describing sub-cellular process and enzymatic pathways are quite different from relationships important in clinical medicine. In English-language thesauri, a special kind of relationship is the *antonym*, which links concept-pairs with the opposite meaning: these apply to adjectives (or nouns derived from them) as well as words that indicate superlatives (e.g., zenith-nadir).

12.3

Combination with Object-Oriented Modeling and Algorithmic Knowledge: Ontologies

The ultimate terminology is called an *Ontology*. Ontologies utilize the well-known ideas of object-orientation (OO), which was originally introduced into programming languages in 1968 in the language Simula 67.² The term was originally introduced by Thomas Gruber in the context of artificial-intelligence research as a computable representation of knowledge in a particular domain, and was not related to terminologies (which have been used much longer to describe knowledge domains) until much later.

Until very recently, “ontology” was a source of confusion because of the term’s much older definition in philosophy as “the study of the nature of existence”, which has little or nothing to do with computation. While Gruber possibly intended to use the suffix “-logy” as in “logos”, meaning “the story” or “the word”, most people who first encounter the term interpret it with the more common suffix “-ology”, meaning “study of”, as in psychology. This usage still trips up Microsoft Word, which consistently flags the phrase “an ontology” as a grammatical error.

An ontology builds on the scaffolding of a thesaurus as described in the account below. (This account below is possibly elementary to those with programming-language exposure, so for their benefit I will provide a summary definition of an ontology as a thesaurus on which an OO framework has been overlaid.)

1. Every concept belongs to (i.e., is a *member of*) a category or *class*. In clinical medicine, commonly used classes include clinical findings, diseases, procedures, medications and lab tests.
2. All members of a class have specific *properties* (also called *attributes*). For example, in a medical vocabulary that deals with lab tests, the properties of an individual lab test include the substance that is being measured and the measurement method.
3. *Constraints* specify permissible values for individual properties. Examples of constraints - the basis of Parameter definitions in Chapter 5 - are:
 - *Data type* (i.e., the value of a given property may be constrained to be a number, text, or date).
 - *Permissible range* (i.e., maximum and minimum values).
 - *Set membership* (i.e., a property value can be one of a set of enumerated values, or must be a member of another class).
 - *Mandatory constraint*: i.e., the value of a property must be specified and cannot be left blank.
 - There can also be more complex constraints based on a combination of properties. For example, for the lab test “differential WBC count”, the sum of the counts of individual white cell types must sum up to exactly 100 (since these represent percentages).

4. Classes are organized into a *class hierarchy*: the ancestral classes are more general, while the descendant classes are more specific: in medicine, for example, medications are a type of therapeutic intervention. A descendant class *inherits* from its ancestors; that is, some of its properties are derived from those of its ancestors, in addition to other properties that apply only to the descendant class.
5. Every concept now becomes a *class instance*, also called an “*object*”.

Optionally, *algorithmic knowledge* may be incorporated into the class definition. In programming languages, subroutines that apply to a specific class are called “methods”. Some frameworks in addition, also support the use of rules,³ (see Chapter 8) which represent algorithmic knowledge stored as data. In certain circumstances, rules allow greater ease of ontology modification.

I will discuss the design of ontologies in detail in the next chapter.

12.4

The Unified Medical Language System

The biomedical domain is fairly vast, and consequently terminologies have been developed for different sub-domain by different standards bodies. However, the boundary between one sub-domain and another is never distinct given any two biomedical terminologies supposedly addressing different sub-domains, there is likely to be some degree of overlap with respect to their content.

In the earlier discussion of data interchange between systems, we discussed earlier the need for a standard terminology to which individual systems could map their data elements. Standards exist for individual domains – e.g., diagnoses, procedures, laboratory tests, and so on, in addition to the terminologies developed for medical library science, but there is considerable overlap amongst these domains. The National Library of Medicine (NLM) identified the need for an overarching “super-standard” which would ideally encompass all of biomedicine and contain the union of all the individual standard terminologies.

This “super-standard”, which has undergone refinement over almost two decades, is the Unified Medical Language System (UMLS). The UMLS is a “meta-thesaurus”, i.e., a compendium of biomedical terminologies. Ideally, these would all be thesauri – hence the name – but many of these are simple taxonomies or controlled vocabularies. Each concept in the UMLS is given a unique identifier (the Concept Unique Identifier or CUI), which is mapped to the unique identifiers (and the corresponding term-phrases) in individual terminologies. Thus, one concept corresponds to multiple *terms*, and sometimes a term may have multiple lexical variants or *strings* (corresponding to variations in case or word order). As in a thesaurus, the individual concepts have relationships to each other, both hierarchical and non-hierarchical, but with a difference. Each relationship is tagged with the source terminology where it originated.

The UMLS is freely available for downloading from the UMLS Knowledge Server (<http://umlsks.nlm.nih.gov>). This requires a login, but a user account is freely offered to anyone who requests it. It is an invaluable resource for anyone who wants to get started with biomedical terminologies. Some of the component terminologies are not available for

commercial use without an arrangement with the terminologies' owners, but a significant proportion is unrestricted.

There is one important caveat that users of the UMLS must be aware of. The NLM is not in the business of curating the contents of the individual terminologies: it only accepts what a terminology's curators provide it. All terminologies are not created equal: some have a superior design compared to others (e.g., thesauri vs. single-hierarchy taxonomies), and even for terminologies with the same design approach, some terminologies are far better curated than others with respect to internal consistency.

In particular, if you try to use all the relationships in the UMLS, you will find that the contents of one terminology will often contradict the contents of another. Obviously, when you have two contradictory assertions, only one of them can be correct, but the UMLS will not tell you which: you have to use your knowledge of biomedicine, or do your research, to spot the erroneous one. UMLS simply tells you where a particular assertion came from: something on the lines of "according to terminology X, the Pope is Jewish".

Consequently, if you insist on trying to use all of the UMLS in the belief that "more is better", you will come to grief. The "one-bad-apple" phenomenon operates here: one bad terminology can vitiate the presence of several good ones by making the whole unusable. Using a subset of the terminologies within UMLS is a more reasonable decision. Often, you will use subsets anyway: for example, if you don't care about genomics, you will omit terminologies which have genomics or gene expression as the prime focus.

NLM will not tell you which terminologies are more reliable than others: you may have to ask your senior colleagues to get this information. I'll break the unofficial Omerta around this topic by telling you that the terminologies that I have found the most useful are the NLM's own Medical Subject Headings,⁴ SNOMED-CT,⁵ the NCBI Taxonomy⁶ (for living species) and Gene Ontology⁷ (for gene expression). In addition to these, you may choose specific vocabularies based on your needs, irrespective of intrinsic quality, such as ICD-9⁸ or the Medical Dictionary for Regulatory Activities (MedDRA).⁹

Even with the well-regarded terminologies, be warned: all parts of SNOMED-CT and Gene Ontology are not equally well curated, and the NCBI Taxonomy may omit the common synonymous terms for many well-known species (e.g., you won't find the terms "Komodo dragon" or "Komodo monitor" as synonyms for *Varanus komodensis*, the world's largest lizard). However, I don't want to be too critical: in a large body of data, some errors are unavoidable. The contents of the more useful vocabularies have been improving with each new release, because the curators listen to user feedback and incorporate their suggestions.

12.5

Requirements of a Terminology: Cimino's Criteria

Whether a given terminology type can be considered adequate depends to some extent on the uses to which you intend to put it. However, if you decide to use a very simple framework, such as a folksonomy or a pure hierarchy, as the foundation for a terminology, you will very quickly encounter constraints that limit what you will be able to do.

Consider the task of indexing textual content with a set of keywords. Since a simple folksonomy or controlled vocabulary is not aware of issues such as synonyms or homonyms, a search of text using specific words will miss certain relevant content (false negatives) as well as occasionally produce spurious matches (false positives). The folksonomy approach, while convenient, is less reliable than the controlled-vocabulary approach in this regard because individual users are also likely to introduce multiple lexical variants (or even misspelled terms) that refer to the same concept. Similarly, if a pure hierarchical taxonomy is used for purposes such as encoding medical diagnoses, automatic query expansion by including the descendants of a concept becomes error-prone, as stated earlier.

James Cimino¹⁰ has nicely summarized the requirements of a terminology in order to represent the biomedical domain adequately. His “desiderata”, i.e., necessary or desirable features, could, however, apply to any complex scientific domain. The reader is encouraged to read the original paper, which is extremely approachable, but we summarize these features here:

- At least thesaurus-level capability. That is, its design must model concepts versus terms, and also record synonymy/homonymy. The terminology must support poly-hierarchy, and associated concept identifiers must not have any intrinsic meaning.
- While the hierarchy will have more general/coarsely granular ancestral concepts with more specific/finer-grained descendant concepts, the supporting software should allow users to tailor the terminology to their needs by hiding concepts that are less useful to them: that is, it should support the use of subsets. (Thus specialists and general practitioners could use different subsets.)
- A given concept must have one and only one meaning, which must never change (*concept permanence*). From the computational perspective, concepts are never removed if they turn out to be obsolete: they are only retired.
- The meaning of a concept should be specified through the use of *formal definitions*. Such definitions are not merely textual descriptions (as in a dictionary), but also incorporate the use of *defining relationships*. The example Cimino uses is that of “pneumococcal pneumonia”, which is defined by being, not only a child of the more general concept “pneumonia”, but with the additional information that the causative organism is pneumococcus. The relationship to the pneumococcus organism becomes a defining relationship.
- As a corollary, the vocabulary will contain certain “primitive” concepts, which may have or may not have associated textual descriptions, but will lack defining relationships. The primitive concepts form, in principle, the building blocks of a knowledge domain. In a perfectly curated vocabulary, every concept in it is either primitive or non-primitive but *fully defined*.
- The meaning of a *concept* must be *unambiguous*. A term associated with a concept must avoid the use of modifying phrases such as “not elsewhere classified” (NEC), which requires the concept’s meaning to be determined only by looking at all other related concepts in the terminology. Also, the meaning of such a concept changes with time as knowledge advances, because related concepts may be added “elsewhere” within the terminology.

- One must be able to *compose* new concepts by combining existing concepts using relationships. The ability to compose new concepts is not limited to the terminology's curators: users of the vocabulary must also be able to do so when they need to. We discuss the issues related to composition shortly when we discuss the topic of pre- and post-coordinated concepts.
- Ideally, the terminology must be accompanied by computable knowledge that specifies *how* concepts are to be used or combined. (We discuss this theme later when we consider description logics, which are a means of representing computable knowledge.) Ideally, this knowledge must be specified in a sufficiently restricted and unambiguous manner that, as far as possible, there must be only one way to combine a pair of concepts. (If more than two concepts are to be combined into a composite concept, one proceeds to combine them a pair at a time, using the intermediate composite concepts created at a given step in the subsequent step.)
- The availability of computable knowledge implies that the software supporting the terminology will ideally be able to recognize redundancy in the form of equivalent concepts as the terminology evolves. (A terminology containing computable knowledge is an ontology, as we have stated earlier.) The example Cimino gives is that of pneumonia of the left lower lobe of the lung. At a particular point in time, it is possible that such a composite concept does not exist in the terminology, so that the user documenting a clinical encounter creates a child concept of pneumonia with a defining relationship that specifies the site of the pneumonia as the left lower lobe.

Later, let us suppose that a new version of the terminology incorporates "left lower lobe pneumonia" as a new concept. The software must ideally be smart enough to recognize that the concept previously created by the user is equivalent to the new concept.

From the practical perspective, the software can only be smart to the extent that the curators and the computable-knowledge designers have done a comprehensive job. If the new concept has been fully defined, and if the definition process used by the curator and the user happen to be identical (because there is only one way to define that new concept), this is a trivial operation. If the new concept is *not* fully defined, guaranteed recognition of equivalency by algorithmic means is impossible. In the intermediate situation – the new concept is fully defined, but the definition process is different because there is more than one way to define it – equivalency determination may be possible, but the computation involved can be significant.

- The terminology support the maintenance of an *audit trail* that records changes made by the terminology's curators, with detailed descriptions as to when and why a given change was made.

I will consider how the schema of a database can be designed to support such a design in the next chapter. For now, I will conclude this sub-section by stating a few caveats.

- What a vocabulary is called by its designers provides no inkling of what capabilities it actually has. You need to inspect its contents carefully to determine these.
 - The phrase "ontology", for example, has become a buzzword that is used to convey an impression of high quality, and is often used for content that, on closer inspection, turns out to be a simple taxonomy or less.

- Folksonomies are associated with another buzzword, “Web 2.0”, since they are employed with social-networking sites, and one occasionally encounter a paper that waxes poetically on their advantages (e.g.,¹¹). While definitely useful for bootstrapping content in knowledge domains where terminologies don’t currently exist, without the additional (and major) step of curation they are not significantly more likely to lead to high-quality knowledge when applied in a scientific domain than (to use the French mathematician Emile Borel’s famous example) a million monkeys armed with typewriters to produce all the works of Shakespeare.
- No existing vocabulary in any domain meets all of Cimino’s criteria.
 - The challenge is partly curatorial: there is a large amount of legacy content that must be dealt with: even something as obvious as providing textual definitions for every concept in the system takes significant effort. Among medical vocabularies, the only vocabulary whose curators are relatively compulsive about providing definitions is the National Library of Medicine’s Medical Subject Headings (MeSH).
 - The only vocabulary that is beginning to employ computable knowledge to constrain its content and specify how concepts can be combined is the Systematic Nomenclature of Medicine Clinical Terms (SNOMED-CT), which I will discuss later.

12.6

Terminologies as the Basis for Data Exchange

Standard terminologies can be valuable for exchange of data between software systems that are developed by different vendors but deal with the same area (or overlapping areas). An example is laboratory tests, which are recorded by software that supports the operation of clinical laboratories, as well as by electronic-medical-record systems that must record these results for individual patients along with clinical findings.

In each system, a given laboratory test is associated with an internal identifier that is meaningful only within that system. Each identifier is associated with a description (a test name), but there may be considerable variation in names between systems, because of the use of synonyms, abbreviations or word variants. For example, serum alanine aminotransferase and serum glutamate-pyruvate transaminase refer to the same test. The blood level of glycosylated hemoglobin, which is a measure of intermediate-term control of diabetes mellitus, may be referred to in some systems by its common abbreviation, “HbA1C”.

It can be seen that transferring data from the laboratory system to the EMR would require the creation of a mapping cross-table that records the correspondence between the identifiers in one system and the identifiers in another. Determining this correspondence can be a painstaking step that demands visual inspection of the descriptive names (and knowledge of the laboratory test area).

If there are multiple systems that need to exchange information with each other, one would have to create multiple cross-tables. If, however, one records, within each system, the correspondence between each identifier and the identifier in a standard terminology for lab tests, then the standard terminology becomes the bridge for data exchange between

systems. If the vendors of the individual systems have already done this (currently there is no guarantee of this), then the task of data exchange is dramatically simplified.

The simple scenario above describes a *one-to-one mapping* between an item in a system and an item in a source terminology. In many cases, notably subjective and objective clinical findings, such a situation may not hold: specifically, the standard terminology may not contain an item that exactly matches the meaning of what we wish to record. In this case, modern terminologies allow a means of *composing* new items using existing items in the standard terminology as building blocks. I'll discuss the issue of concept composition shortly.

12.7

Recognizing Concepts from Terminologies in Clinical and Scientific Text

I will discuss the topic of concept recognition in only as much depth as is needed to follow the subsequent sections of this chapter. This section is by no means intended to be comprehensive: for those who would like to learn more, we refer them to the books by Manning et al. (for Information Retrieval),¹² Jurafsky and Martin (for general Natural Language Processing)¹³ and Manning and Schuetze (for Statistical Natural Language Processing).¹⁴

The original difference between text processing and Natural Language Processing (NLP) was one of approach.

- Text processing, also called *Information Retrieval* (IR) is primarily a statistical approach oriented to processing large volumes of data rapidly, based on looking at words or phrases in isolation without attempting to extract the meaning of larger units of text such as sentences. Search engines such as Google and Bing are examples of IR technology.

The most basic search-engine approach is to index individual *words* that are recognized in text. If, however, one can employ an electronic thesaurus, then it is possible to recognize phrases in the text that match to thesaurus entries, and create an index based on the thesaurus's *concepts*.

- NLP tries to extract meaning, but its approaches are several orders of magnitude slower than IR approaches in terms of the volume of text they can process in a given time. It applies knowledge of sentence structure, particularly of grammar, and this in turn depends on determining the parts of speech of individual words or phrases. (If you decide to take up NLP seriously, you will have to relearn long-forgotten grammar concepts like gerunds and past participles, which you had probably done your best to forget.)
- *Information Extraction* is the general name for any process that, operating on unstructured text, aims to extract structured content – for example, data that may be imported into a database. It may use NLP technology, but this is hardly the only way. For example, if the set of documents one is working with has known regularity in their structure – for example, a sequence of specific section headings, captions and so on whose location is predictable – then the extraction process will utilize these. A well-known and well-regarded

commercial program, Monarch™,¹⁵ performs yeoman service in extracting data from textual reports produced by legacy systems running on mainframes whose programmers have long since retired: here the only feasible way to get information out of these system is through the reports that they generate. The developer or power user configures Monarch to recognize the structure of a particular report type, in particular where the desired information is located.

The practical problems that complicate concept recognition, whether using IR or NLP-based methods, are lexical variants (e.g., order of words in a phrase), ambiguous words with multiple possible meaning (i.e., homonyms) and abbreviations (which can also be ambiguous). NLP is also much more vulnerable than IR to malformed sentences or faulty grammar: thus, it has poor accuracy when dealing with the telegraphic text of medical progress notes.

In the past, IR and NLP were like oil and water, and the fact that there are still separate textbooks on each topic indicates that they haven't been fully synthesized. However, the differences between the two are gradually being blurred. Limited use of NLP techniques can improve the accuracy of IR for medical text in specific circumstances, allowing:

- Recognition of negation (the concept is absent or checked for and ruled out): for example, the vast majority of occurrences of the word “fracture” in clinical notes are cases of suspected rather than confirmed fracture.
- Gradation of severity, in cases where the parameter being analyzed can be graded (e.g., pain, obesity), or sub-categorization (e.g., smoking status can be one of non-smoker, ex-smoker or current smoker).
- Identification of temporal sequencing, e.g., determining the order in which clinical findings appeared in a patient.

Statistical NLP is a relatively new field that combines NLP with the techniques of IR and machine-learning. It has proved useful in the resolution of ambiguous phrases. Its approach here is based on detecting the presence of specific words, or word categories in the passage associated with the homonym. Thus, the word “anesthesia” in the context of a clinical examination is likely to refer to local loss of sensation, while it is more likely to refer to the procedure in the context of surgery. The machine-learning techniques use *supervised learning*: that is, the software is presented numerous examples of passages, each containing a different usage of the homonym, where the correct interpretation has been labeled by a human: it uses these examples to “learn”. Of course, this approach must be used for each homonym that we wish to disambiguate, so it requires human effort to prepare the input to the software, by identifying (or creating) passages and providing the correct interpretation for each instance of homonym use.

As I'll shortly discuss, the success of automated concept recognition also depends to a great extent on the terminology that phrases are matched to. Ideally (if we temporarily ignore the issue of homonyms) a phrase in the text should match to one and only one concept in the terminology. If one uses a terminology with a significant amount of redundancy – such as duplicated phrases, or multiple concepts with highly similar meanings whose differences are of little practical significance – the phrase will match not only to the concept desired but

also, inaccurately, to several other concepts. (The concept-matching process is probabilistic: while the concepts matched are ranked based on degree of closeness to the phrase – e.g., number of common words – it often happens that the match desired is not to the concept ranked highest.)

Consequently, the output of automated recognition needs to be checked by a human, especially if the concepts recognized have legal/reimbursement implications. The greater the number of one-to-many matches in the output, the greater the extent of human cleanup required. In the case of a concept-matching process using a terminology with very little discrimination (e.g., the full content of UMLS), the output of the matching process may have very little value when used later for search purposes. Problems with using the full UMLS partly stem from attempts to match text against the entire biomedical domain rather than the more circumscribed area of clinical medicine: a well-known concept recognition program, which did not use knowledge of grammar and sentence structure, was once notorious for matching the common word “his” (the possessive form of “he”) to the abbreviation for the amino acid Histidine.

12.8

Composing New Concepts from Existing Ones

12.8.1

Pre-coordinated Concepts vs. Post-coordinated Concepts: Concept Recognition

Compound concepts that are formed by composition operations are called *post-coordinated* concepts, as opposed to *pre-coordinated concepts*, which are formed by simply forming a long descriptive phrase that combines two or more concepts into one. To provide a concrete example, we could define the concept of hypertension due to renal disease as a post-coordinated concept thus: we could start with the concept “hypertension, systemic arterial”, use the relationship “caused by” and then specify the concept “renal disease”. If, on the other hand, we directly added the phrase “renal hypertension” to our vocabulary, with or without an accompanying textual explanation, this phrase would represent a pre-coordinated concept.

Pre-coordination is the “traditional” way of creating new concepts, while post-coordination is the “modern” way. The difference between the two is that the meaning of a pre-coordinated concept can only be determined by humans if all that we have to go by is the concept phrase itself. Only if one takes the effort to create defining relationships for this concept (“is-a” – “hypertension”, “caused by” – “renal disease”) is computation possible. Note that a purely post-coordinated concept consists only of the relationships: it is not assigned a descriptive name.

Pre- and post-coordination are illustrated in Fig. 12.5.

In general, some degree of pre-coordination is unavoidable in any evolving domain. For example, the concept “hypertension” itself is really a pre-coordinated concept that really means “systemic arterial blood pressure elevated”. The concept is just so common that medical shorthand has evolved for it. However, it is possible to go overboard with

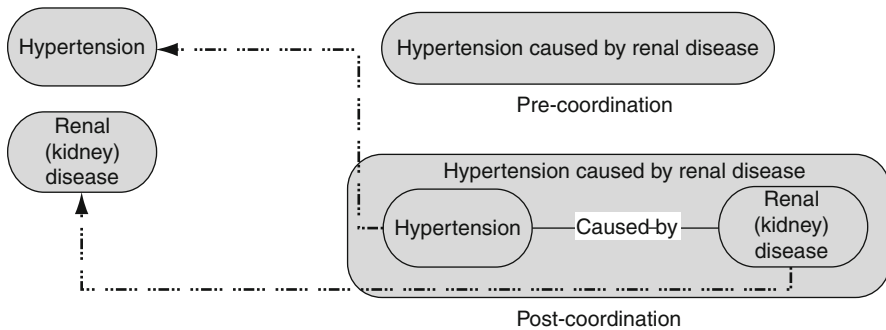


Fig. 12.5 Pre-coordination vs. post-coordination. The terminology has two existing concepts, “hypertension” and “renal disease”. We wish to represent the concept, “hypertension caused by renal disease”. In *pre-coordination*, the old-fashioned way of defining new concepts, one simply creates a new concept with this phrase. The new concept is not linked to the two existing concepts on which it was conceptually based: its association with these concepts must be inferred by the user using knowledge of English and Medicine. In *post-coordination*, in addition to creating a phrase to describe the new concept, its semantics is defined in relation to the two concepts on which it was based: its definition consists of the concepts *hypertension* and *renal disease* linked to each other using the relationship *caused-by*. (The dotted link of hypertension to hypertension, etc. is only used to indicate a *reference* to the concept: in practice, one uses only the machine-generated identifiers of the referenced concepts in the definition of the new concept.)

pre-coordination, greatly reducing the usability of the vocabulary. A poster child for out-of-control pre-coordination is seen in the ICD-9-CM (International Classification of Diseases, Version 9, Clinical Modification) terminology: “Chronic duodenal ulcer with hemorrhage AND with perforation but WITHOUT obstruction (disorder)”.

There are several reasons why such highly compound concepts are deprecated.

- Every clinical condition has associated concomitant findings, causative factors, type of onset, and so on. Mashing together all of these separate concepts (including their absence) into a giant composite concept serves no scientific benefit. Further, once you start this process, it’s hard to know where to draw the line. Excessive pre-coordination leads to combinatorial explosion, with dramatic growth in the terminology’s size: N concepts can be combined into $N * (N-1)/2$ pairs, $N * (N-1)*(N-2)/6$ triples, and so on.
- Currently, in situations where encoding is mandated (e.g., for billing purposes), it is still performed, or at least verified, by humans – who are either care providers, or specially trained coding staff. The larger the size of the terminology that a human has to work with, the slower (and the more potentially error-prone) the encoding process. Busy humans who have to encode an encounter within a fixed time are simply unlikely to pick highly compound concepts. Healthcare providers, for example, would rather spend the limited amount of time they have per patient in providing care.
- Consequently, there is a lot of research aimed at supporting encoding by software that scans clinical text, and recognizes concepts. The scan is performed within a particular window of text – typically, a sentence at a time.

Highly compound concepts are almost never stated explicitly in text as a continuous phrase. Instead, the component concepts would be likely to be stated across an entire paragraph (and sub-concepts that are missing, such as obstruction, would not be stated at all). Consequently, such concepts are unlikely to be recognized at all in any program that scans a sentence at a time: in most cases, the software can only recognize the individual component concepts. Scanning an entire paragraph at a time is not generally feasible, because the computation required to match concepts can go up exponentially. (Remember that a passage can contain numerous unrelated concepts: trying to permute them in the hope of finding a match to a compound concept like the above will fail far more often than not.)

- In general, it is hard to recognize even simple pre-coordinated concepts in text when they are implied but not explicitly stated by the text's creator. For example, renal hypertension is suggested by a patient who presents with hypertension and who has a history of chronic pyelonephritis for the last 15 years – the mention of the separate findings may be several sentences or even paragraphs apart. Only if the clinician has used a phrase like “hypertension due to renal disease” or “renal hypertension” will the concept be readily picked up.

However, post-coordination, which involves composing a new concept through software assistance, has its own challenges:

- One is not allowed to compose a new concept simply by throwing existing concepts together. There is a certain process to be followed. Firstly, one must select the parent concept. In the case of renal hypertension, the parent concept is hypertension, not renal disease (because renal hypertension is a type of hypertension). A relationship must be selected from a list, and in most cases, the set of permissible relationships is dependent on the broad category of the parent concept.

For example, “caused by” is a permissible relationship for hypertension, which belongs to the category “disease/disorder”, but it will not apply to a category like “pharmaceutical”. From the user-interface perspective, the “relationship” would need to be made available in a pull-down menu whose contents would change depending on the parent that was selected. Similarly, once a relationship has been picked, the second concept would have to be checked for validity: for example, a disease or chemical substance can be a cause of another disease, but a concept belonging to the category “normal body part” cannot.

It follows that, even though the idea of post-coordination is that users, rather than the terminology's original curators, are permitted to create new concepts, the process of concept creation is intricate enough that it should only be entrusted to *trained* super-users. With a naïve user, it is very easy to go wrong even with the most robust software support, because many errors will still escape software validation.

- Even with the most intuitive user interface, post-coordination takes time because the user must still search for the concepts of interest by keyword. Certainly one does not expect busy clinicians (or their administrative help) to try to compose highly complex concepts like the infamous duodenal ulcer example above during a time-constrained clinical encounter: they have more important things to do.

The only group we are aware of that uses locally developed post-coordinated concepts (of modest complexity) in production mode is James Campbell's group at the University of Nebraska Medical Center¹⁶. This group has anticipated and bypassed the issue of clinician resistance by the simple expedient of providing a local database that contains a reasonably large set of commonly used concepts that have been post-coordinated in advance. Essentially, as clinicians, in the course of their work, encounter concepts that they need to encode (but which are not present in the standard terminology used, SNOMED-CT), electronic requests are made for these concepts to be added. Curators service the requests by creating a post-coordinated expression corresponding to each requested concept.

The resulting set of post-coordinated concepts thus represents a locally developed *extension* to SNOMED-CT, which is accessed transparently along with the existing contents of this terminology: clinicians use these added concepts by the normal process of keyword search without even needing to know what post-coordination is. The locally developed content is licensed and redistributed by GE Healthcare to their other clients. Campbell emphasizes that clinicians are unlikely to tolerate time-consuming or confusing interfaces.

12.8.2

Description Logics

We introduce this topic because it is directly related to the challenges of trying to perform useful computations on concepts that have been composed. Description Logics (DLs) are a means of knowledge representation that are a particularly good fit for computation ("electronic reasoning") using the contents of terminologies: because of this, DLs were called "terminological systems" previously. The plural form is used because a given description logic (singular) is characterized by the operations and semantics that it supports for working with concepts. These operations are used to compose new concepts from existing ones, as stated earlier, as well as to define concepts. In addition to the use of knowledge-domain-specific relationships to compose concepts through defining relationships, one can use operations such as:

- Intersection (also called conjunction, using AND): e.g., "narcolepsy with (i.e., and) cataplexy".
- Union (also called disjunction, using OR) – e.g., "fracture of wrist or (fracture of) hand".
- Negation: e.g., "liver cirrhosis *without* history of alcoholism".

Some of the semantics that a DL can support include:

- Cardinality restrictions: e.g., a normal human hand or foot has five digits.
- Inverse properties/relationships: e.g., the relationship "caused by" has the inverse "causes".
- Enumerated restrictions: e.g., the Gender of a person can be either Male or Female.
- Universal restriction: e.g., *every* medication must have at least one therapeutic indication.

To support a DL, a given terminology must have at least a taxonomy structure, though obviously, a thesaurus structure is more desirable.

The commonest inference operation in DL is *subsumption* – determining, given a set of concepts and the relationships between them, whether a given concept is a more specific (or more general) form of another given concept (e.g., is myocardial infarction a type of heart disease?). Computationally, determining subsumption involves traversing is-a relationships from one concept in a given direction until the second concept is encountered. In the above example, we would identify all ancestors of “myocardial infarction” until the concept “heart disease” was encountered, or until one reached the top-level concept.

Note that a given DL may lack support for individual operations or semantics. The lack of support is simply a consequence of a design decision made by that DL’s implementers: DLs subsumption operations are intended to be reasonably efficient computationally (in DL lingo, the ideal operation is one that is *decidable*). It is known that with operations such as Union, decidability can be compromised. (Some implementers may decide that capability is more important, even at the risk of certain operations running slowly: other implementers may decide to forbid certain computationally expensive operations.)

Constraints such as we have described for concept composition, where the choice of relationship and second concept are limited by the category of the first concept, are also part of a given DL. Such constraints are understandably domain-specific: constraints that apply to concepts in physics are quite different from concepts applying to biomedicine.

12.8.2.1

Current Status of DL Applications

The application of DLs to large biomedical terminologies is currently in its infancy. Even for the only biomedical terminology to employ a DL, SNOMED-CT, the set of constraints that have currently been defined is incomplete. This is because the original SNOMED has been in existence much longer than the term description logic, which originated in the 1990s, and whose ideas became popular only about a decade ago. Retrofitting SNOMED to use a DL is challenging, and so a lot remains to be done. For example, users may occasionally find that, in particular medical sub-domains, the existing set of SNOMED relationships is incomplete, and they may need to create new relationship types. (For example, it is inadequate to represent sub-cellular processes: understandably so, since SNOMED was originally devised for clinical medicine. Even for clinical specialties, it has limited utility for psychiatric conditions or psychological processes: the psychiatrists use their own terminology, the Diagnostic and Statistical Manual of Mental Disorders.)

The International Health Terminology Standards Development Organization (IHTSDO), the body that currently controls SNOMED, is refining the SNOMED Machine-readable Concept Model (MRCM), a detailed specification that represents DL constraints that apply to SNOMED within a set of relational tables. The idea is that software that is intended to support SNOMED content curation or concept addition can consult the contents of these tables, so that as new rules are defined by IHTSDO, the software itself would require minimal modification. (In other words, the contents of the MRCM tables are metadata, which drives the operation of curation software.) I’ll briefly discuss the MRCM subsequently in the Ontologies chapter.

12.8.3

Practical Issues with SNOMED-CT

SNOMED-CT is currently regarded as the single most comprehensive single-source clinical terminology. (It is, of course, less comprehensive than UMLS, of which it is a part, but it has the advantage that, for terminology users intending to focus on the clinical domain, its curators commit to providing consistent data: UMLS does not offer this guarantee, as we have stated earlier.) Of the existing large biomedical terminologies, SNOMED-CT comes closer to the definition of “ontology” than any other.

While originally developed by the College of American Pathologists (CAP), it is now under the control of the IHTSDO: the US National Library of Medicine, the maintainer of UMLS, is a member of IHTSDO. Some years ago, NLM signed an agreement with CAP supporting its continued development in return for CAP making SNOMED-CT freely available within the USA. There is now a move to make SNOMED the basis for documenting the clinical encounter. However, there are significant challenges ahead in meeting this goal.

- SNOMED is huge, with a lot of legacy content. There is some degree of redundancy within it, and despite being intended to support post-coordination, a lot of the problematic ICD concepts (like the infamous duodenal ulcer example above) are also part of it. Clearly, there is a need to create a subset of concepts that are useful for documentation.
- Encoding an entire clinical note such as a discharge summary to record every concept of relevance – history and clinical findings, lab results and other investigations, therapeutic interventions – is a much longer process than merely encoding the diagnoses or the procedures performed for reimbursement purposes. One could well conceive that an encoding process that was not almost entirely automated through software would take much longer than the actual clinical encounter itself. No clinician would be willing to put in this kind of effort, and it would be hard to justify the hiring of coding personnel to do this since such encoding is not directly tied to issues of financial reimbursement or legal requirements.
- The accuracy of concept recognition depends to a large extent on the terminology from which the concepts must be recognized: large size and the presence of redundant concepts complicate the recognition process (whose output must ideally be a single concept per phrase), even if issues such as homonymy were ignored.

We are not yet at the stage where concept-recognition software can be trusted to run entirely autonomously, because its output still has significant false-positive and false-negative rates: especially where inaccurate coding can have legal consequences, its output still requires human review. This may be less of an issue for clinical-encounter encoding, whose purpose is primarily to facilitate subsequent retrieval of relevant records in response to queries: as in search-engine software such as Google, one accepts the possibility of some degree of error in return for speed of search.

Appendix: Working with the UMLS

In a production setting, you will require more than one vocabulary: ICD for billing, MedDRA (Medical Dictionary of Regulatory Activities) if you are dealing with drug and device safety, SNOMED if you need to encode the clinical encounter, and so on. You can

either go to the original vocabularies, or simply get the UMLS and extract the content of the vocabularies that you're interested in. Recent versions of UMLS come with a program called MetaMorphoSys that makes subset creation a menu-driven process.

Whether you wish to segregate the individual vocabularies into separate sets of physical tables is up to you – UMLS provides a means of doing so – but I find that using the UMLS design of keeping everything in one place scales nicely. Of course, you need to track which term comes from which vocabulary. One problem with trying to create separate subsets using UMLS is that, for each individual terminology, the resulting schema is much more complicated than the original. (For example, the original ICD-9 consists simply of a set of hierarchically related terms: it is not even concept-oriented.)

The text files in the UMLS distribution are not always in third normal form, i.e., some columns are redundant. By some very simple transformations to the raw data, along with the creation of a few extra tables that are not part of the UMLS distribution, you can improve performance and storage requirements. For example, UMLS uses 8-letter alphanumeric IDs for concepts, terms and so on, but because the alphanumerics simply consist of a single letter followed by a zero-padded number sequence, these can be readily converted to numbers by eliminating the leading letter and storing as long integers (which take only 4 bytes each) and also improves performance (machine operations on numbers run somewhat faster than on strings).

Similarly, to facilitate lookup of all terms containing a word, UMLS provides a “word index” consisting of the word, which can be up to 80 characters long in English, followed by the IDs of the term and concept it is associated with. By creating a table of unique words and assigning each word a unique ID using your DBMS's auto-number/identity-column feature, and then using this ID instead of the full word in the word index, the size of the table will shrink drastically.

The UMLS supplies data in two formats: the “original” and “rich” release formats. The former is meant for legacy applications that have been using UMLS for the last 20 years, but the latter format contains extra information that the former does not (especially for vocabularies like MedDRA, where a lot of information in the original terminology gets lost in the process of homogenizing). Today, there is no reason for not biting the bullet and switching to the rich release format.

References

1. Vanderwal T. Folksonomy coinage and definition. Available from: <http://vanderwal.net/folksonomy.html>. Cited 9/2/10.
2. Dahl O-J, Nygaard K. How object-oriented programming started. Available from: http://heim.ifi.uio.no/~kristen/FORSKNINGSOK_MAPPE/F_OO_start.html. Cited 6/1/09.
3. Date C. *What Not How: The Business Rules Approach to Application Development*. Upper Saddle River, NJ: Addison-Wesley Pub Co; 2000.
4. National Library of Medicine. Medical subject headings – home page. Available from: <http://www.nlm.nih.gov/mesh/>. Cited 11/25/04.
5. International Health Terminology Standards Development Organization. SNOMED clinical terms (SNOMED CT) technical implementation guide; Copenhagen, Denmark 2009.
6. National Center for Biotechnology Information. NCBI taxonomy homepage. Available from: www.ncbi.nlm.nih.gov/Taxonomy/taxonomyhome.html. Cited 7/8/10.

7. Gene Ontology Consortium. An introduction to gene ontology. Available from: <http://www.geneontology.org/>. Cited 11/26/10.
8. World Health Organization. *International Classification of Diseases*. 10th ed. Geneva: WHO; 1992.
9. MedDRA Maintenance and Support Organization. Medical dictionary of regulatory activities. Available from: <http://www.meddrasso.com/>. Cited September 10, 2009.
10. Cimino JJ. Desiderata for controlled medical vocabularies in the twenty-first century. *Methods Inf Med*. 1998;37(4–5):394–403.
11. Quintarelli E. Folksonomies: power to the people ISKO Italy-UniMIB meeting; 2005, Milan, Italy.
12. Manning C, Raghavan P, Schuetze H. *Introduction to Information Retrieval*. Cambridge: Cambridge University Press; 2008.
13. Jurafsky D, Martin JH. *Speech and Language Processing*. 2nd ed. Englewood Cliffs: Prentice-Hall; 2008.
14. Manning C, Schuetze H. *Foundations of Statistical Natural Language Processing*. Cambridge: MIT Press; 1999.
15. Datawatch Inc. Monarch: overview. Available from: http://www.datawatch.com/_products/monarch_overview.php. Cited 11/1/10.
16. Campbell J, Carpenter P, Sneiderman C, Cohn S, Chute C, Warren JJ. Phase II evaluation of clinical coding schemes: completeness, taxonomy, mapping, definitions and clarity. *J Am Med Inform Assoc*. 1997;4(3):238–251.

In the previous chapter, I've introduced the term "ontology", as the most evolved form of terminology. I've stated that no existing terminology meets every aspect of the definition, though some come closer to it than others. SNOMED CT, UMLS and LOINC meet the requirement that every concept be placed in at least one category. WordNet,¹ a widely used thesaurus of English that has also been explored to improve the quality of biomedical thesauri,^{2,3} has some aspects of an ontology: its authors, however, cognizant of the hype surrounding this term, modestly disavow such claims.

In this chapter, I'll focus on large-scale ontologies for the simple reason that comprehensive coverage of a non-trivial knowledge domain necessarily results in voluminous content. Such content, intended to be authoritative, can rarely be created by a single individual; when multiple people are involved and the content bulks up, the likelihood of mistakes goes up non-linearly. A software application that tries to perform inferences based on this content can misbehave if content treated as gospel turns out to have errors: one manifestation is where one bad inference can lead to others in a chain reaction.

Some of this material is aimed that those who have to support curation environments for ontologies. However, this chapter is also aimed at the consumers of large-scale vocabularies – typically vendors or healthcare organizations – who download periodic releases of one or more official terminologies/ontologies to use them in production settings. In doing so, they will need to convert the text-file formats of the source vocabularies into formats based on RDBMSs for better performance. In the course of long-term use, it is almost inevitable that new concepts that aren't in the official release will need to be incorporated: no vocabulary is perfect. When new releases come along, they will need to preserve their efforts, while identifying the changes in the new releases that may have made parts of their local effort obsolete. The need for local maintenance impacts on the vocabulary's publishers: if they are considerate to the software-maintenance needs of their user base, they need to provide ancillary information with each new release that will make maintenance chores much easier.

While the term "ontology" is a big part of Semantic-Web and "Web 2.0" jargon, I make no apology for advocating the use of present-day RDBMSs, which allow almost arbitrary extension with stored code created using robust modern programming languages. The issues of maintenance of voluminous content involve old-fashioned data-processing ideas such as transactions, concurrency support and data validation, and these can't be thrown away just because a new paradigm comes along. I've yet to be convinced that alternatives to RDBMSs offer the necessary robustness and scalability for the same investment in software-developer effort.

13.1

Functional Requirements of Large-Scale Ontologies

The essential requirements of an ontology relate to its being able to represent domain information. Historically, two different groups – the object-oriented programmers and AI researchers on the one hand, and the taxonomists and librarians on the other, devised different solutions. The former conceived of classes with their own attributes (“properties”) and embedded code (“methods”). The latter understood that if you want to organize information about a domain, you begin by listing the concepts that apply to the domain, the terms (synonymous forms), and then record how concepts are related. Both approaches were partially right – though, as the previous chapter has outlined, you would possibly start with the taxonomist’s approach first – and they happen to be complementary.

Interestingly, while AI work traces its roots to Marvin Minsky’s concept of “frames” – some software tools use frames terminology – Minsky’s original 1974 paper was criticized because of failure to acknowledge the prior 1966–1967 work of Dahl, Nygaard and Hoare, embodied in the programming language Simula 67, of which frames were largely a rediscovery. This criticism may have influenced the paper’s fate. Rather than appearing in a peer-reviewed journal, it was self-published as a 1974 MIT Technical Report,⁴ and later as a 1975 chapter⁵ in a book edited by Minsky’s student, Patrick Winston.

I’ll now list the requirements of ontologies that are related to scale.

- Their large size mandates continual maintenance by numerous, often geographically distributed curators as well as a software development team. This leads to the following requirements.
 - Differential access privileges and role-based restriction. Content-definition experts and software developers typically need read-only access to instance data: content curators need read-write access, but most of them should be restricted to classes in their area of expertise: only a few individuals will have broad access.
 - Concurrent multi-user access, with transaction support. Multiple curators may work on different parts of the ontology simultaneously. In addition, many change operations involve multiple steps: these steps must either run collectively to successful completion, and be “committed”, or the operation as a whole must be canceled (“rolled back”) if an error, which could include the introduction of inconsistency, is detected computationally.
- Internal data consistency must be maximized.
 - Certain categories of errors can be prevented through basic validation checks. However, several operations involved in truth maintenance in data collections can theoretically make the problem an “NP-hard” (NP= non-deterministic polynomial),

which means that its complexity increases exponentially or worse as the problem size increases linearly. The chain-reaction effect applies here: the longer an error stays undetected, the more it can propagate. One category of computationally detectable errors is cyclical relationships, where concept A is a child of concept B, which is a child of concept C, but concept C is recorded erroneously as a child of concept A (which is now its own great-grandparent).

- Other errors may pass all computational checks, and be apparent only to someone who knows the domain. Audit trails for both instance data and metadata must record what changes were made, when and by whom.
- Production ontologies must be capable of handling a diversity of data types.
 - If a knowledge domain requires, for example, the ability to represent, and manipulate, data such as numbers and dates, then your toolset must not prevent you from doing so.
- The design must facilitate synchronizing of local content with periodic new releases of reference ontology content.

13.2

Ontology Design Principles: Some Warnings

13.2.1

Ontologies vs. Databases

In planning a design for ontologies, it is first necessary to clear up the misconception that an ontology is somehow significantly different from a database. Several articles in the Artificial Intelligence (AI) literature emphasize the differences in various ways. For example, Noy and Klein's paper "Ontology Evolution: Not the Same as Schema Evolution"⁶ talks about change operations in databases vs. ontologies, while Gruber's paper "Ontology" in the Encyclopedia of Database Systems⁷ talks about ontology design being independent of the implementation and the similarity of ontology-modeling languages to first order logic.

While both papers make valid points, they seem to me to miss the bigger picture. These papers are in effect comparing Apple Pie to Granny Smith apples, or an automobile to an internal combustion engine. That is, they are comparing a specialized, finished product to a component that can be used in multiple products. An ontology, as far as I'm concerned, is nothing more than a database application intended for a special purpose. It has a large component of active developer/user-defined metadata (which ordinary databases don't care about – but if you didn't, you wouldn't be reading this book), and also incorporates computational knowledge of some kind. Given an ontology's large metadata component, there will naturally be a whole family of operations to consider (e.g., relating to metadata change, which in turn impacts the data that is interpreted through the metadata) that are not relevant to databases lacking this component.

Of course, you don't *have* to use an RDBMS to implement ontologies anymore than you have to use it for a database. As the contents that you have to manage get more voluminous and the design gets more complex, however, I think you'd be extremely foolish not to.

As far as computational knowledge goes, it need not be physically part of the ontology, as long as the software that manipulates the ontology can access it. Modern RDBMSs, however, allow physical co-location of *some* algorithmic knowledge with the data as stored procedures/functions: this, as I've stated earlier, can improve performance in client-server scenarios. As discussed in Chapter 8, Rules, an important means of representing non-procedural knowledge, are actually data that you can store in your database, and these rules can be executed under the control of a rule interpreter. It is possible to devise your own rule interpreter, but if you need real computational muscle and don't want to spend a lot of effort reinventing wheels, consider employing a robust open-source engine such as Drools. While you cannot make the Drools engine a physical part of your ontology, as long as you can call it from your application, it doesn't matter. After all, your ontology must also have a user interface, but user-interface code is also not intended to be stored in a database. All of the above doesn't mean that it is trivial to build a robust code base, but you will have to do a significant amount of work no matter what approach you use: you just don't want to be re-inventing solutions to the problems of multi-user data management that were solved 20 years ago.

13.2.1.1

"Knowledge Bases": A Dissenting Opinion

An ontology is claimed to store "knowledge" while a database stores mere "data": the home page for Protégé (see below) refers to the software as an "ontology-editor and knowledge-base framework".⁸ However, the term "knowledge", when used in the context of electronic representation, needs to be rigorously specified, or else readers may risk conceptualizing it in terms that resemble Supreme Court Justice Potter Stewart's famous characterization of hard-core pornography: "hard to define, but I know it when I see it".⁹

The clearest definitions of the differences between data, information, and knowledge that I've ever read are in Larry English's book on business data warehouse information quality.¹⁰ English starts with data as the raw material, and then defines information as *data in context*, which is usable and meaningful. English defines information with the "equation":

$$\text{Information} = \text{function of (Data + Definition + Presentation)}$$

That is, information quality depends on data correctness, clear definitions of what the data mean and understandable presentation of the data. Similarly, English defines Knowledge as

$$\text{Knowledge} = \text{function of (Information + People + Significance)}$$

That is, knowledge is *information in context*: the value added to information by people who have the experience and the acumen to understand its potential, so that when multiple units of information are available, they can determine the relevance of each unit to the particular problem at hand. (English also goes on to define “Wisdom” in the business context as applied knowledge, when people with access to knowledge are empowered, and given the resources, to act on what they know. Without such empowerment, an organization may still be dysfunctional.)

In terms of electronic representation, it is not particularly hard to represent information in a database: definitions can be stored in metadata, and if metadata also contains presentation information, we can present individual items of data in useful ways. Moving from information to knowledge, however, is a major leap: databases must also store procedural code, or employ non-procedural algorithms such as rules. Even such a system cannot function autonomously as humans might, unless it interfaces to software that can communicate with the external world through inputs and outputs. Most systems that call themselves “knowledge bases” fail this more rigorous definition, because they do not store algorithmic knowledge.

Finally, a much harder problem is *self-knowledge*: knowing what you don’t know, or can’t decide with the information and algorithmic knowledge that is available. I’ve seen too many instances of electronic naïveté, as embodied by ontology-based software applications with incomplete knowledge that don’t know when to quit. Such software is not merely an object of ridicule – as in the case of the infamous Microsoft Office Assistant, whose rule-based “knowledge” was next to useless.¹¹ In production settings such as healthcare, such simplistic thinking may be dangerous. While, for better or worse, the term “knowledge base” is here to stay, I tend to be uncomfortable with it: as software developers we need to be suitably humble and honest about what we are actually creating.

13.2.2

Prototypes vs. Production Systems

A tool that is popular in the AI world for modeling ontologies is Protégé,¹² developed at the Stanford department of Biomedical Informatics. Its major strength is its visual development environment, which allows ease of learning, easy creation/ modification of metadata (class and attribute definitions) and data (class instances). Like software originating in the AI world, it doesn’t support the thesaurus kernel of concepts, synonyms and relationships: you need to implement these as special-purpose classes. It supports algorithmic knowledge to a limited extent using “plug-ins” which are Java classes placed in a specially designated folder (some plug-ins relate to the user interface). See “Protégé Programming Development Kit” at <http://protege.stanford.edu/doc/dev.html>.

However, the Protégé architecture is not designed to scale to millions of class instances. Protégé’s data-storage mechanism uses a *single* de-normalized, modified entity-attribute-value table¹³ that holds both metadata (i.e., class definitions and hierarchy) and instance data. (The Protégé documentation uses the equivalent AI argot, “frame-property-value”.)

While Protégé does not use database technology by default, the interpretation of a row that constitutes data is based on other rows in the same table that record metadata, so in principle, almost every operation requires a self-join. For the modest-sized data sets that are typically used with it, this is not a problem as all operations are memory-based: Protégé loads an ontology's complete contents into RAM, changes are also made in memory and saved periodically (*en masse*) by the user.

13.2.2.1

The Challenges of Scaling up

The advantage of this approach is that data does not need to be migrated between tables if the logical data model changes – something that happens frequently during iterative prototype refinement. However, this strength becomes a limitation for large-scale ontologies edited by multiple users.

1. Role-based privileges (in particular, metadata editing, which should be restricted to only a few expert users) are much harder to implement when metadata and data are combined into one table.
2. As I'll discuss shortly, the metadata sub-schema of an ontology is stable and domain-independent. Therefore its structure is naturally modeled using *conventional* design: if you use EAV, you have to manually program features such as referential integrity and constraint support, and needlessly so.
3. Protégé's approach to metadata and data edits is simply not geared to multi-user scenarios such as record locking or transactions. More important, there is no support for modeling separate users: all users must trust each other completely. Audit-trail support consequently becomes challenging (more precisely, impossible).
4. Finally, in large scale ontologies where a particular class has a *homogenous, non-sparse* structure, you would like the freedom to model it using a conventional table with appropriate indexes. You sacrifice this control – a minor issue for a prototype, but not for a production system with a hundred thousand instances. In particular, it is woefully inefficient to model the basic terminology components – concepts, terms and relationships – using an EAV approach. Relationships, in particular, are already modeled using a kind of special-purpose EAV (two entities/concepts, one attribute, zero or more values) and an additional layer of indirection doesn't make sense.

For the above reasons, while I still strongly advocate using Protégé for prototyping an ontology that will ultimately become large, once your design stabilizes you should port your design to an RDBMS. However, I know some hardened developers who prefer to use Unified Modeling Language for the design phase, coupled with any of several commercial tools that translate the UML into an RDBMS schema and skeletal code.

Given, however, that Protégé uses a modified EAV design, the question is whether the EAV/CR design approach would work for the data management aspect of large-scale ontologies. The answer is unequivocally “yes”. Of course, as I've stated earlier, you have to provide an interface to your data, and you have to write code (or utilize existing

frameworks) that will perform the inferencing you need. I believe that this is much easier to do than to reinvent multi-user RDBMS functionality.

13.2.2.2

The Challenge of Preventing Misuse

One caveat about Protégé-based modeling applies to any tool that is sufficiently easy to use: it can be abused by naïve users. It is possible to create ontology designs that are not grounded in software-engineering reality: as one of my colleagues puts it, such designs can assume a parallel universe where Al Gore still serves as President. For example, an animal-species ontology has been modeled that consists exclusively of a class hierarchy without instances. Every entry in the taxonomy – dog, penguin, tiger – is a class.¹⁴ Software designers, on the other hand, will tell you that a class is worth defining only if its properties and property values distinguish it from other classes. Further, a class is not just data: it also contains code unique to that class. If all that you care to record about a class is its name and its immediate parent – e.g., tigers belong to the genus *Panthera* – than you should be modeling these as concepts, so that you use *inter-concept hierarchical relationships* rather than a *class hierarchy*. A hierarchy with 300 classes may possibly work: one with 200,000 won't.

Protégé won't stop you from creating designs like this, any more than most RDBMSs (or EAV/CR for that matter, which relies on an RDBMS) will red-flag a totally non-normalized, or otherwise unsound, design. (MS-Access has tried to solve the non-normalization problem with a "table analyzer wizard", which makes suggestions based on the frequency distribution of values it encounters in the table's columns, but it is easy to fool with a sufficiently idiotic design, and understandably doesn't work if you have little or no data in your tables, as is expected during prototyping; to use it productively, as when importing data from flat text files, you have to know about normalization in the first place.)

13.2.3

The Closed-World vs. The Open-World Assumption

Inferencing operations can use one of two assumptions: closed-world and open-world.¹⁵ In the former, what is not known to be true is assumed to be false. In the latter, not-true=unknown. A case where open-world is more applicable is medical history of a patient: absence of a record indicating a past illness does not mean that the illness never occurred – unless we have recorded the negation explicitly. With respect to facts in a large-scale ontology that is intended to be authoritative, however, closed-world is reasonable.

The language OWL (see below) uses the open-world assumption, while Prolog uses closed-world. It is said that SQL uses closed-world, but that is an over-simplification: SQL must ultimately operate on a database. In the design of medical databases, and as part of good medical practice, it is imperative to record significant negative findings, as I've stated earlier in my introduction to the clinical record. It is thus a trivial matter to distinguish between *no record of* past hepatitis from *no* past hepatitis.

13.2.4

Semantic Web Technologies: OWL and SWRL

The Web Ontology Language,¹⁶ named OWL in a transposition of letters to suggest wisdom, uses XML syntax. OWL syntax supports limited declarative computational knowledge in the form of “axioms”, which also include definitions of classes and their properties, and the representation of data (“facts”) that relate to these classes and properties. Again, in keeping with its AI roots, concepts and synonyms and relationships are not intrinsic, though relationships can be treated as a kind of property/attribute. OWL’s inferencing operations rely on description logics (DL), introduced in the previous chapter. OWL comes in three versions: OWL-Full is the most expressive, but certain operations may be NP-hard in the worst case; OWL-DL (DL=Description Logic) limits the operations to those guaranteed to be computational tractable, while OWL Lite is so stripped-down as to be a joke: any self-respecting OWL implementation would aim to support at least OWL-DL.

OWL version 1.x is possibly adequate for modest-sized ontologies that deal exclusively with symbols. The drawbacks of employing it as the *primary* means of representing large-scale ontologies are:

- OWL isn’t computationally complete. Some time in the future there may be means of allowing developers to extending it by implementing classes/properties using their favorite language, but that hasn’t happened yet. For example, if the inferencing operations that you need to perform include numeric manipulation, working with OWL 1.x offers less productivity than working in assembly language, because numeric data types aren’t supported.
- Most implementations load the entire OWL ontology – facts and axioms – into RAM, but access to individual elements isn’t particularly optimized. If there are millions of facts, and they all have a regular structure, the only reason to be operating directly on XML (basically glorified text) is ideological purity: handling large data volumes with disk-based indexes (and smart caching in RAM) is what DBMSs did well 30 years ago. (Proprietary approaches like IBM’s Ontology Development Toolkit use an RDBMS rather than raw XML, and have reported a 100-fold gain in performance.¹⁷)
- Don’t even think of trying to operate on a large OWL 1.x ontology in the multi-user situation. The few “native XML database” engines¹⁸ that support transactions and concurrency do so only *across* XML documents, not within the single giant XML document that comprises the ontology.
- OWL’s DL-based inferencing is very limited in what it can do – primarily classification. Rules are more expressive, because they allow a limited form of programming. The best attempt at a Rule language that integrates with OWL is Semantic Web Rule Language¹⁹ (SWRL, pronounced “swirl”). Underneath the hood, it is based on XML, but fortunately, to protect the sanity of rule creators, allows rules to be written using human-friendly syntax. In terms of expressive muscle, however, SWRL is a toy compared to implementations such as Drools, which is far ahead of the rest of the field. This is despite the very creditable creation of SWRLTab,²⁰ a Protégé plug-in that among other things, allows developers to write their own Java functions (“predicates”) that can be called from SWRL.

In October 2009, cognizant of the numerous limitations that were encountered when applying OWL to nontrivial problems, the W3C OWL Committee released the specifications for a new version of OWL, 2.0,²¹ with intended built-in support for a variety of data types (specifically for those supported by XML schema, such as numbers and date-times), as well as assertions of negations (e.g., “Disease X is not caused by organism Y”). There is also a new dialect of OWL, OWL QL (=Query Language), intended to query large sets of facts with regular structure that would be stored in an RDBMS: in theory, OWL constructs could be converted to SQL. However, I’ve yet to hear of any operational software that actually demonstrates such functionality: a presentation by a team at Oxford University describes some of the challenges in achieving this with even a simple construct.²²

Even when OWL 2.0 becomes a reality, it is not intended to address the issue that this chapter deals with, that is, *managing* your ontology content.

13.3

The Elements of a Large-Scale Ontology Schema

The essential components of this schema are the thesaurus kernel, the EAV/CR Metadata tables, and tables for history and audit trails. You will of course have data tables, and depending on whether your data is homogeneous or not, and the number of instances of each class, you will either use the type-specific EAV data tables, or physical tables, or a mixture. This sub-schema is illustrated in Fig. 13.1, with some tables compressed because they are identical to the EAV/CR tables of Chapter 11.

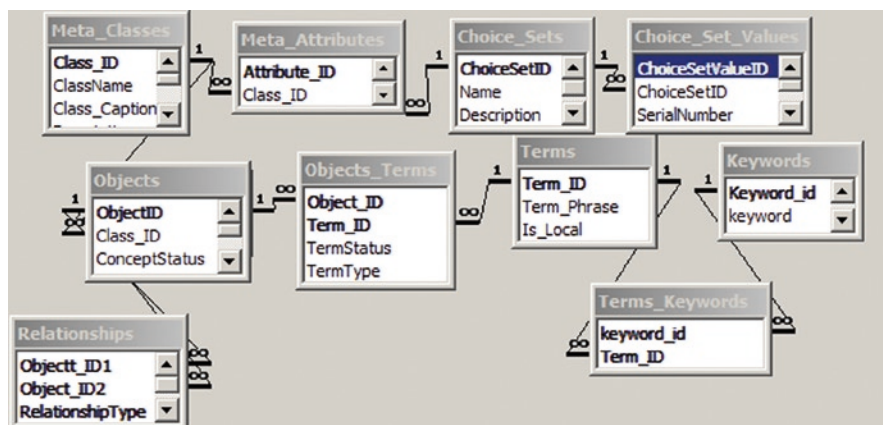


Fig. 13.1 Core Thesaurus and EAV/CR Metadata Schema. The tables that record details of each Object, whether as data-type specific EAV tables or special-purpose tables, have not been shown. The tables Terms and Keywords (and the bridge tables Objects_Terms and Terms_Keywords) are used to record synonym/term information and used to speed up search for individual terms based on keywords that they may contain

The Objects table of EAV/CR is functionally identical to the Concepts table in the thesaurus kernel – with a modest difference I’ll discuss shortly. In ontologies, Objects are nothing more than Concepts that have been categorized into classes, with attributes (properties), and values for those attributes, associated with them. Therefore there is no purpose in having two tables do the work of one: whether you call your table Objects (with an Object ID) or Concepts (with a Concept ID) simply depends on which phrase you are partial to. In the schema diagram, I’ve used “Objects”: you can mentally substitute “Concepts” if you are used to working with controlled terminologies.

I now discuss each component in turn.

13.3.1

The Thesaurus Kernel

The thesaurus sub-schema consists of core tables – concepts/objects, terms, and relationships – and tables that facilitate fast search. In every core table, you will also have columns that indicate the date a record was created and the date it was last modified. (Tom Rettig (1941–1996), database developer extraordinaire and former child actor in “Lassie”, was the first to emphasize that almost every table in a production database needs to have these two columns.²³) As a matter of fact, the curators of most source terminologies use such columns internally: the columns just don’t make it into the public releases.

All the core tables include a Boolean column, `Is_Local` (Boolean). To understand its purpose, consider that the biomedical field is fortunate in having numerous high-quality thesauri available (as well as UMLS, the mother of all thesauri). You might as well bootstrap your ontology with such existing content, and if you need the contents of multiple sources, UMLS is the best place to start.

However, no matter how comprehensive your imported (“reference”) terminologies are, you are likely to need local concepts that haven’t made it into their current versions as yet. Rather than maintain these in separate tables that are clones of the thesaurus core tables, it’s simplest just to add a Boolean flag, `Is_local` (default False) to each of the core tables to indicate that the concept, term, or relationship is of local origin rather than originating in the imported terminologies.

13.3.1.1

Distinguishing Local IDs from External Content IDs

Most of the core tables use long-integer surrogate primary keys, which are typically named using the convention `<table-name>_ID`. When you combine local content with reference content in the same tables, and the latter will be re-imported with each new release, you must ensure that the local content’s key values do not clash with those of newly imported reference content. One approach used in the past was to use large numbers expected to be well beyond the range of the reference-content IDs: unfortunately, this approach is fragile – for terminologies such as SNOMED, IDs are large-non-sequential and unpredictable because they include a check digit. (SNOMED began using 64-bit “big-integer” IDs relatively recently.)

A more robust approach is to use *negative IDs* for local content. SQL Server can create negative auto-numbers if you seed both the seed and increment of a table's "identity" (auto-number) to -1. When bulk importing reference content, you must turn the auto-incrementing off using `SET IDENTITY_INSERT <tablename> ON` (meaning that you will insert the column's values explicitly). Oracle's "sequences", which achieve the same result, can also have negative increment values.

13.3.1.2

The Concepts/Objects Table

This table is identical to the Objects table described in the EAV/CR chapter, except that it contains an `Is_Local` column.

13.3.1.3

The Terms/Synonyms Table

The Synonyms table of EAV/CR could do double-duty as the Terms table with the addition of the `Is_Local` column. However, in huge biomedical terminologies, *homonymy* (same term=many concepts) is quite common, as I've stated in the previous chapter. Therefore, the relationship between Terms/Synonyms table to Objects is actually many-to-many, not many-to-one. Therefore, we need a bridge table, `Terms_Objects` between the Terms and Objects tables. Because of the bridge, each unique should be associated with a surrogate key, the Term ID, and it is this ID that is used (along with Object ID) in the bridge.

Supporting Multiple Languages: It's not likely that your large-scale ontology will have to support terms in multiple languages, but if you do, UMLS (and SNOMED) contain terms in several languages besides English (contributed by terminologists in those countries), and from the design perspective, all you need to do is add a language ID column to the Terms table.

13.3.1.4

Assisting Rapid Search of Terms by Keyword

The tables for supporting search are a normalized counterpart of those provided in the UMLS. Specifically, there is a table of unique **Keywords**, and a bridge table (**Terms_Keywords**) indicating which keyword occurs in which term. You may wish to implement a routine that transforms a word supplied by the user into a root form, so that your keywords table can be smaller and store only root forms. Alternatively, you can store words as they occur in the term, and broaden the user's query dynamically. To transform a word into a root form, you can use either stemming or lemmatization (normalization, in UMLS terminology): the difference between the two is that

- Stemming is more drastic, often yielding a phrase that is not a word in the language. There is a controversy about the benefits versus drawbacks of stemming for technical subject areas.²⁴
- Lemmatization may occasionally yield more than one root, because of the homonymy in English: thus “leaves” can be the plural of “leaf”, or the present tense of “left” (as in went away). A utility supplied with the UMLS²⁵ will perform lemmatization, but it is most efficient to run this in batch mode against every word in every term in your database, store the computed results along with the original word, and then use table lookup at runtime.

If you wanted to, you could run this algorithm against every word in the UMLS, from which you can readily extract a Keywords table. Cases where the root is different from the word itself occur mostly for common words rather than technical terms, so you can save space by storing only such instances.

Using the Terms_Keywords table it is possible to support simple Boolean searches – e.g., terms containing one or more keywords. You can also implement relevancy-ranked (Google-style) search, where the user can simply supply an arbitrary set of keywords, and the results are ranked closest-match first. The original algorithm was described by Keith Sparck-Jones in 1972²⁶: several open-source implementations are available, but to implement the algorithm efficiently, you need to maintain against each Keyword a number called the Inverse Document Frequency – here, it is the logarithm of (total number of terms in your database / number of terms containing that keyword). This number can (and should) be pre-computed: you can use any base for the logarithm, as long as you use the same base in your relevance-ranking computation.

13.3.1.5

Inter-concept Relationships

The basic structure of a Relationship record is: Object-1, Object-2, Relationship-Type, (Additional Details....). While I’ve designated “Relationship-Type” as a single column, in some designs (notably UMLS), two or more columns acting as a composite may serve this role. There are so many practical ramifications of this content, which is by far the most voluminous in the standard controlled vocabularies, that I’ll discuss them in the next section.

13.3.2

The Metadata and Data Subschema Schemas

If you are using an EAV/CR design (and I’ve already stated why you should), this is unchanged from the EAV/CR chapter, so I won’t re-describe it here. There are some points to note, however, if you are relying on imported UMLS or SNOMED content, which I now discuss.

13.4

Special Issues: Dealing with Imported Content

13.4.1

Class and Attribute Structure in UMLS

In UMLS, the closest thing to “classes” is the *semantic types*. Every concept belongs to at least one semantic type (for example, a Clinical Drug or a Pharmacological Substance), and some concepts, as in the real world, can belong to more than one. However, the types are extremely *ad hoc*: for example, “eicosanoid” is a type but “alkaloid” is not. The class hierarchy has evolved only modestly over the duration of UMLS’s existence, and in most cases you are better served simply by identifying a concept’s hierarchical parent/s.

The UMLS’s curators have the very difficult task of combining data from multiple formats in multiple source vocabularies into a common format that is heavily influenced by EAV modeling. This homogenization, which is partly driven by the need to keep the schema relatively simple and uninfluenced by the special aspects of individual source vocabularies, works well for some vocabularies. For others, such as the LOINC vocabulary (=Logical Observations, Identifiers, Names and Codes), which is used to describe laboratory tests and enumerated clinical observations, the process is something of a sausage grinder – it mostly results in loss of individually identifiable characteristics. Further, as I’ll subsequently discuss, enumerated information is poorly modeled.

If your work needs to concentrate on an individual vocabulary, you may be better served by trying to access it independently of UMLS, the exception being when the vocabulary is a commercial one that you can’t afford. (UMLS still allows you to use such content for non-commercial – e.g., research – purposes, which effectively lets you try before buying.)

A lot of the information associated with individual concepts that can’t be put elsewhere is recorded as attribute-value pairs in a specially designated table. However, the rigid association of permissible attributes for specific classes hasn’t been formalized, though you could no doubt discover patterns if you spent enough time doing it. The existence of a particular attribute depends more on the source terminology where a concept originated – though, if a concept is present in multiple terminologies, all those attributes get included. Consequently, the attributes are highly sparse – values of most attributes don’t exist for the majority of concepts. In fact, if you want to still use an EAV/CR framework for such concepts, the best you can do for starters is create a single class called “UMLS Concept”, and associate the entire grab-bag of attributes with this class.

13.4.2

Challenges in Dealing with Types and Enumerations

I’ve mentioned earlier that the terminologists and the object-oriented/AI folks arrived independently at two different approaches for modeling knowledge. While the approaches mostly harmonize well when used together, one area where the latter got it right and the

former wrong is in the treatment of enumerations. This is reflected in the design of vocabularies that are a compendium of content merged from elsewhere – notably SNOMED and UMLS.

For example, the Apgar score (devised by American anesthesiologist Virginia Apgar) evaluates a newborn’s health through a quick clinical examination of five parameters (with the mnemonic Appearance, Pulse, Grimace, Activity, Respiration), each graded 0–2 (2 = normal), so that the final score (the sum) is 0–10, which can be treated as an ordinal entity. Each parameter is anchored: a grade is defined unambiguously to minimize observer subjectivity. Thus, a respiration grade of zero means that breathing is absent, while one indicates weak, irregular gasping. The Apgar test is generally performed at 1 and 5 min after birth (sometimes at 10 min as well). Appropriate modeling would involve separating the timing of the test from the parameters themselves. One would also record the summated nature of the overall score as an arithmetic formula.

In the object-oriented/AI modeling approach, the Apgar score would be defined as the sum of five ordinal parameters. In a pure terminological approach, however, because there is no concept of enumerations or relative magnitude (everything is just a string), there are a glut of pre-coordinated concepts that permute the parameter, the timing and the score. Thus, for the overall score, you have 10×4 concepts, of the pattern “Apgar at 1 min = 0 (finding)” etc.

Problems arise when data in the clinical encounter is indexed using the IDs for concepts such as these. A query such as “show me the patients with Apgar Score at 5 min or later that is less than or equal to 4” cannot be answered. This is an example of a trivial computational task becoming unsolvable because of the incorrect choice of data structure and knowledge representation. (A solution would require individual user sites to implement enumerations: in the object-oriented/AI modeling approach, the publicly released ontology would already have this information.)

Certain controlled terminologies, such as the National Cancer Institute’s Common Toxicity Criteria for Adverse Effects (CTC AE),²⁷ which is used to grade severity of individual adverse events to cancer therapy, are based entirely on anchored enumerations (in this case, a 0–5 scale). The structure of UMLS and SNOMED make it currently impossible to import this content without major information loss. CTC AE, a very valuable and high-quality anchored vocabulary, is unfortunately still not part of the UMLS despite being developed by another branch of the National Institutes of Health.

If you need to import such terminologies into your ontology, you will need to associate a sparse attribute called “enumeration” with the Object, and define this attribute as having sub-structure: an ordered list of code/definition pairs.

13.4.3

Separating Lower-Level from Higher-Level Metadata

In principle, *everything* in a terminology is metadata, because it is used to annotate external content such as clinical text and non-terminology metadata content such as parameter definitions. Within a terminology, however, some of the content helps to organize, validate and define the other content. I’ll avoid introducing the term “meta-metadata”, but you should get the general idea: some metadata are higher-level than others.

For example, in SNOMED, every concept belongs to a class: SNOMED calls classes “hierarchies”. However, the definition of a class in a terminology is somewhat looser than in a typical EAV/CR design: because concepts themselves form a (mostly) hierarchical structure – remember that many concepts can descend from two or more parents – a hierarchy is nothing more than a top-level concept and all its descendants.

SNOMED adheres to the principle that every class must have a defined set of attributes. Attribute-value information for a given concept, however, is stored in the general-purpose Relationships table, and such rows are differentiated from the others by using the categorical tag “Defining Relationship”, which was introduced in the previous chapter. Attribute-value data is represented using the approach that the “Entity” is the Object -1 column, the “attribute” is the Relationship-Type column, and the “value” is the Object -2 column. For such data, SNOMED refers to Object -1/Entity as the “Domain” concept and Object -2/Value as the “Range” concept.

SNOMED’s content adheres to the organizing principle of ontologies that for a given class, only certain attributes/Relationship Types are permissible. A similar principle applies to Value/Range concepts. In an EAV/CR design the value for a given attribute must belong to a specific data type: in terminologies, data types are not generally employed (numbers, dates etc., don’t generally appear as values), but the permissible Range concepts for a given Relationship Type must be similarly constrained. For example, a concept belonging to the class “Procedure” (such as a type of surgical operation) has several permissible relationship-types, such as the body site, the device used, the method, and so on. For the relationship-type “body site”, the range concept must be a concept that belongs to the class “body structure”.

A distinguishing feature of terminologies is that *attributes tend to be sparse*: this is a consequence of their current state of evolution. That is, we would like to have all the applicable attributes (and their values) supplied for a given concept, but that information may not be currently available. Recently, however, there is an acknowledgment that certain attribute/value pairs must be present for certain concept classes in order to render the vocabulary usable.

In the design approach used by SNOMED and UMLS *everything*, whether a class definition or a relationship type, is a concept. As such, there is an entry for it in the Concepts/Objects table. From the perspective of building a system that supports robust data validation, however, it is desirable to segregate such high-level metadata into specially structured tables where they can allow lookup-based validation.

Two important categories of higher-level metadata that should be implemented in these terminologies but is currently recorded only as prose in the documentation are:

- The unique values of various columns based on enumerations: however, you can get these easily by performing the query `select distinct (column-name) from tablename`. If you are building an editing interface for local content that must go into the same tables, these enumerations form the basis of pull-downs in the user interface.
- The constraints on attributes/Relationship Types and values/Range concepts based on class in SNOMED, discussed above. A brief 2009 paper by Nadkarni and Marengo²⁸ describes a simple table-driven approach to implementing these constraints. IHTSDO, the maintainers of SNOMED, have been working for a while on a *machine-readable concept model*,²⁹ which is a metadata model that records such constraints explicitly. I’ll discuss this in the case study at the chapter’s end.

13.4.3.1

Relationship Types and Semantic Inverses

When designing a special-purpose ontology, you can certainly take advantage of the existing relationship types available in reference terminologies: however, you will almost certainly need to introduce your own. A special category of metadata that applies to relationship types is the idea of recording the “semantic inverse” of a relationship type. For example, it is well known that the absorption of Iron in the intestine is increased by Vitamin C, which chelates the Iron ion. You could express this fact computationally in the Relationships table as follows: “Vitamin C” (object-1), “increases-absorption-of”, (relationship type), “Iron” (object-2). If you switched Iron and Vitamin C around and made Iron object-1, you would have to use a relationship type “absorption-is-increased-by”.

Which representation would you use? The truth is that you could use either. However, the user doesn’t know which way the fact is stored, and may query the system in one of two different ways. You have two choices: you can either store the same fact in two different ways, doubling the storage requirements but simplifying your search algorithm, or you can maintain a table of semantic inverses, which records pairs of inverse relationship types, so that, no matter how a user queries the data, the software will do the right thing. The use of semantic inverses thus allows a simple form of automatic query broadening. (Some concepts are their own semantic inverse, as when two drugs mutually reduce the absorption of each other.)

The concept of semantic inverses is fairly old and used extensively in multiple knowledge domains. In E-R diagrams and UML Class diagram, for instance, a relationship between two tables (or classes, in UML) is typically given two labels, one at either end of the relationship, describing the meaning of the connection: the second label is the semantic inverse of the first. Note that inverses simply offer a means of switching from subject to object or vice versa: the convention of devising a name for an inverse is to switch from active voice to passive, as in the above example. This is different from opposites/antonyms: thus, the opposite of “decreased” would be “increased”.

Opposites are much thornier to handle, in part because certain words are polysemic – they have multiple closely related but different meanings, and each meaning has a distinct antonym. Also, when trying to deal with them, you encounter the problem of negation and the open/closed world assumptions: absence of a fact in a database does not indicate that it is false. In the above example, “not increased” is not the same as “decreased”: something could be unchanged.

The Relationships table can be used to record semantic inverses using the relationship type “Inverse Of”. In practice, however, you might want to segregate semantic-inverse information into its own separate table.

13.4.4

Challenges in Dealing with Imported Inter-concept Relationships

The homogenization process mentioned earlier is possibly most problematic in the case of inter-concept relationships. Several functionally distinct types of data are merged into a

single table, with the use of additional columns to distinguish individual relationship categories. These include:

- *Attribute (“property”) values that also happen to be Objects/Concepts.* SNOMED’s defining relationships, discussed above, are examples of these. In an object-oriented design, the “domain” concept would be modeled as the entity, the relationship-type as an attribute, and the “range” concept as the value. (In the EAV/CR design, this information would go into EAV_Objects.)
- *Hierarchical Relationships* (Parent/Child, General/Specific), which are used, as described in the previous chapter, to answer queries related to classification.
- *Equivalence and other historical information.* Equivalence means that two concepts, one of which is now obsolete, have identical meaning. This information properly belongs to the audit-trail category, discussed shortly. While all tables must have time-stamp information, they are particularly important for this category of data, because they allow recovery from curator errors, a point made by Ceusters, Spackman and Smith.³⁰ Further, when reference content is publicly released, release of change history with associated time-stamps for individual rows facilitates synchronization of local content with reference content.
- Information that refines the meaning of concepts when used in post-coordinated expressions (“qualifying relationships”) such as severity and clinical course.
- *Additional relationships*, such “part-of” associations between anatomical structures.

In a production system, these categories of information should be segregated into different table, for several reasons. Queries regarding classification can limit themselves to the hierarchy. Segregation also simplifies security. For example, hierarchical/additional relationships are added by curators: on the other hand, historical information is really created by software applications that prompt the curators for some of the input (e.g., reason for making a change), but most users should be prevented from even accidentally modifying change history.

I’m almost certain that internally, the SNOMED and UMLS teams use a segregation strategy, despite the merged content that the public sees. If you are supporting a team that has to maintain reference content, however, you would do well to segregate the above data from the beginning.

13.5 Scaling Up: Security and Audit Trails

13.5.1 Enforcing Selective Access

I’ve touched briefly on this in the EAV/CR chapter. To avoid repeating detailed content from Chapter 15, which deals with this theme in depth, I’ll simply state here that in a large-scale ontology, you have to maintain a table of *users*, a table of *roles* (e.g., software developer, content curator, etc.), and a cross-table matching users to *roles*: certain users may have multiple roles.

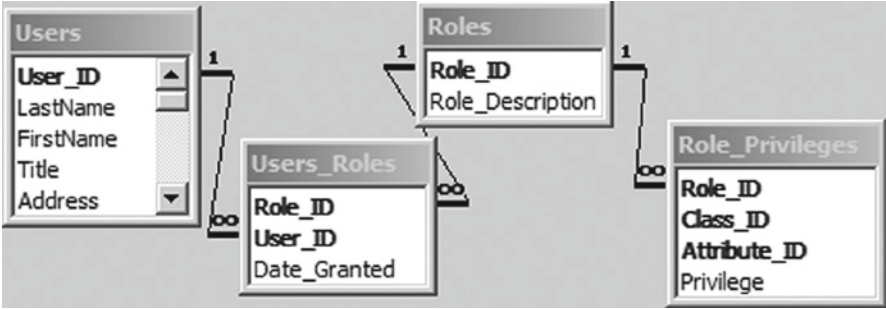


Fig. 13.2 The subschema used for enforcing selective Role-based access to individual Classes and Attributes

In most selective-access scenarios, privileges are granted by role. In large ontologies a unique issue arises with respect to curation: different curators also specialize in different kinds of subject matter, and so the ability to edit data may be restricted by subject area. One way to implement this is to make the roles more granular by adding an extra column, Subject Area ID, in the User_Roles table of fig. 13.2, which points to a master table of subject areas.

The subschema for selective access is illustrated in Fig. 13.2. The structure of the tables **Users**, **Roles**, **Users_Roles** (a bridge table) and **Role_Privileges** are fairly self-explanatory. Connections to Meta_Classes and Meta_Attributes from the last table have not been illustrated.

13.5.2
Maintaining Audit Trail Information: Change Management

I’ve described the representation of audit trail data in Chapter 11. In an ontology – indeed, in any metadata-driven system—it is necessary to keep audit trails for both metadata changes as well as data changes. It is best to separate the two physically in separate tables (**Audit_Data** and **Audit_Metadata**) whose structure is shown in Fig. 13.3. (The connections to the rest of the schema – through Object ID, Class ID, Attribute ID and User ID – are not

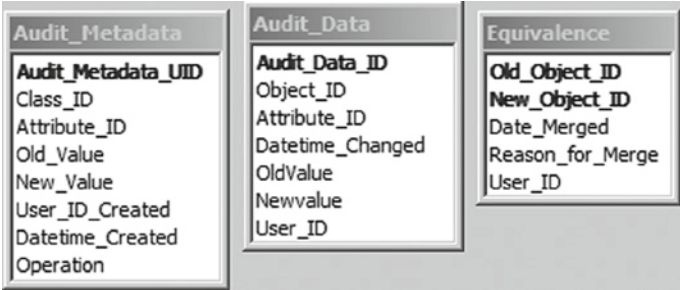


Fig. 13.3 Tables used to track changes in both Metadata and Data. The table equivalences is used to track concept merges. Relationships to previously described tables are not shown

shown.) I've discussed the general structure of a change table in Chapter 11. Thus, for objects, you record the Object ID, the attribute that was changed, when it was changed, the old and new values, and who changed.

The two audit tables are nearly identical. The `Audit_Data` table additionally records the Object ID of the changed Object/concept: it does not need to contain the Class ID (it can be looked up from the `Objects` table). Metadata changes, while much less frequent than data changes, are more far-reaching in their consequences. In particular, certain changes in metadata impact data. The `Audit_Metadata` table's `Operator` column identifies the action that caused the change (see 13.5.2.1).

The reason for having a near-identical structure for both tables is that the EAV/CR schema records information on every Class and Attribute in the system, irrespective of whether a class represents data or metadata. To recapitulate, even the `Meta_Classes` and `Meta_Attributes` tables (and their attributes) contain information about themselves. Therefore the same mechanism can be used for change tracking.

Noy and Klein discuss ontology structure changes (i.e., metadata changes, in the approach used here) in their previously cited paper on schema evolution. However, they only consider the limited family of metadata modifications that are automatically followed by cascading data modifications. Some of these operations, such as removal of a class or attribute, may result in data loss, and introduction of a new constraint may lead to inconsistency because existing data may violate that constraint.

In a production system, as opposed to a prototyping environment, such changes must not be allowed to happen without first checking for the possibility of data loss or inconsistency. The danger of corrupting the system through thoughtlessness is one reason for separating metadata-editing privileges from data editing privileges, a point repeatedly emphasized throughout this book.

If the developer decides that such changes are necessary, the challenge is in devising a robust strategy to deal with it. I now describe the elements of such a strategy. This strategy has the following aspects:

1. Detecting an attempted change.
2. Testing the potential consequences of the affected change, and reporting these to the user.
3. Allowing the change to take place automatically if it is safe to do so, and preventing such changes if it is not.

Let's explore each of these.

13.5.2.1

Detecting Metadata Changes

For classes that are simulated – i.e., modeled through EAV, you will be allowing modifications through a custom-developed user interface, and therefore your interface, can detect any action that can change metadata, no matter whether this action is safe (e.g., addition of a new attribute) or dangerous, such as class removal, attribute removal, type change or constraint addition.

Certain changes to the system are structural, such as addition of tables or columns, or changes in column definitions. You can detect these by implementing Data-Definition-Language triggers, which is not particularly simple, as I've stated earlier, because of the need to write a SQL parser. The alternative is to build a GUI accessible only to developers, which provides a GUI that lets the developer perform certain actions such as Class/Attribute deletion or Type Change through a GUI (when can emit DDL-SQL when the developer actually confirms the intended action.) Here, the button clicks can be trapped.

13.5.2.2

Testing and Reporting Consequences of Intended Change

The simplest way to do this is to generate a count of the data elements that would be affected by the change. For a class or sparse-attribute deletion, the number of instances would be reported. For a constraint change (e.g., change of an attribute's data type), the count would be less useful than identifying *what* data would be affected.

Since the quantity of affected data might be significant, it is necessary to create a report that can serve as a semi-permanent record. Here, I recommend keeping the esthetics to a minimum: if instead, the report can be generated in a basic tabular structure that can be stored in a spreadsheet or a temporary relational table. Especially if stored in the latter, it can be the basis for bulk edits that can be made through scripts: however, many edits will have to be made individually (and manually).

13.5.2.3

Allowing or Disabling Changes

If the test reveals that there are no data elements that would be affected, the change can proceed. If existing data would be affected, however, it cannot be allowed. This is in keeping with the design of most production systems, where actions such as cascading deletes or updates are rarely, if at all, enabled. If disallowed, a report would be generated after the system informed the user about its refusal to proceed.

The normal workflow at this stage would involve edits (by developers, users/curators, or both), and reattempting of the action until the report finds no affected data. (These edits would, obviously, be recorded in the audit trail for data.) Conscientious developers expecting to make a series of major changes might even perform a full backup of the system before starting data modification.

For safety, one could implement the testing/reporting and the change action within a transaction, which would be committed if all was well and rolled back if potential problems were discovered.

13.5.3

Reporting Metadata and Data Changes

Allowing data changes to be undone is not the only reason for creating the audit trail: it can also be the basis for reports. In situations where the ontology /terminology being maintained

is one that is periodically released publicly, with users making changes to their local copies, such reports tell the users and the developers associated with a given installation what changes have been made. The UMLS, for example, provides tables that states which concepts and terms have been deleted.

13.5.3.1

Merged Concepts

An important category of historical information is *merged concepts*, when multiple concepts are combined into one (so that all concept IDs in a set except one are removed). Merging occurs either because a concept is *superseded*, or when *duplicate concepts* are discovered. Sites that use a standard terminology but maintain their local content will also have to perform their own merging when they discover that some newly added concepts in the public release happen to match semantically to concepts that they had previously created locally.

Merging is an operation unique to terminologies. In terms of database design, while one could record a merge record as “obsolete-concept *is-equivalent-to* current-concept” in the Relationships table (as SNOMED does), I prefer to record equivalence information in a special-purpose table separately from other historical information, because the equivalence record has a different structure: the person who merged the concepts must provide a reason, if only in prose, as to why the merge was performed. In Fig. 13.3, this is shown as the table Equivalence. (I have placed it in Fig. 13.3 since merge information is part of the audit trail.)

Maintenance of merges can be fairly intricate, because one needs to replace all instances of the superseded concept with the new. The extent of change may go far beyond the ontology itself. For example, indexes to medical text that were concept-ID-annotated must be updated.

13.5.3.2

Metadata Changes That Invalidate Local Content

Wade and Rosenbloom’s case study³¹ reports how a major overhaul to SNOMED – replacement of one class with another (and reclassification of concepts), addition and removal of attributes – affected an installation which had created a significant volume of local content according to the previous definitions, and had to laboriously discover the local data elements that were affected by all these changes and fix them.

In general, the kinds of reporting that individual sites will want to perform on a metadata/data audit trail may be far more extensive than those envisaged by the terminology’s maintainers. As such, it may be desirable to make the trail (or at least a subset of the columns in it) available to those who need it. The trail is generally quite voluminous, and so it may be better to make it available for downloading for those who need it rather than make it part of the standard public release.

13.6

Case Study: The SNOMED CT Machine-Readable Concept Model

One of the drawbacks of the current version of SNOMED CT, as I’ve stated earlier, is that the constraints that define valid content are currently specified only as narrative text in the

technical documentation. Unless these constraints were implemented in a database that stored SNOMED CT content, local implementations could very easily introduce errors when creating local content.

An IHTSDO initiative called the SNOMED CT machine-readable concept model (MRCM) aims to define a metadata model that will be part of new SNOMED CT content releases, and will facilitate electronic validation of content created both by the SNOMED CT curatorial team as well as local implementations. The chair of this initiative is Dr. Jeremy Rogers of the British National Health Service.

A prototype version of the MRCM is available as a Microsoft Access database that defines the constraints using a relational model. An interesting aspect of the prototype is that because constraints may be created by multiple individuals independently, the primary key is a Global Unique Identifier (GUID), discussed in [Chap. 2](#).

The prototype requires the basic SNOMED CT tables to be imported in order to be operational, since the constraints must ultimately refer to Concept IDs; my personal preference is to upsize the contents to MS SQL Server; SNOMED CT content currently does not hit the 2 GB limit of Access databases, but if you add value by breaking up the content of textual descriptions into keywords, incorporate synonyms from UMLS and index the content, you may well hit this limit. (In any cases, Access databases become fragile and prone to corruption well short of the 2 GB limit.)

The constraints supported by the MRCM are of the following types:

- **Relationship:** These constrain permissible relationship types (i.e., attributes) for a given class (domain) of concept, as well as the permissible values for a given attribute. The major difference between the independently developed approach of Nadkarni and Marengo, cited earlier, which also implements these, and the MRCM, is that the MRCM uses a more normalized design: as the number of constraints increase, a normalized design is obviously preferable.
- **Cardinality:** These specify restrictions on the *number* of attributes possible for a concept, in terms of minimum and maximum number of occurrences. A *mandatory* attribute is thus one where the minimum number of occurrences is one: an *optional* attribute is one where the minimum number is zero.
- **Dependency:** These may be of three subtypes. From a set of possible relationship types for a class:
 - *only* one must be present ;
 - *at least* one must be present ;
 - If one is present another must also be present.

Constraints themselves may be of two different levels of authority: mandated or recommended. The difference is that violations of the former should be prevented, while violations of the latter are possible but should be warned against. The example given in the MRCM documentation of a recommended constraint is that for concepts belonging to descendants of the concept “endoscopic procedure”, if the relationship type is “access instrument”, then the value should belong to concepts that are descendants of the concept “endoscope”.

The MRCM also formalizes the content of enumerations using a table called SV_Lookup, which is a denormalized combination of the Choice_Sets and Choice_Set_Values tables of the EAV/CR design; this facilitates the design of editing interfaces.

References

1. Miller GA, Fellbaum C, Tengi R, Langone H, Ernst A, Jose L. Wordnet: a lexical database for English. cited 4/1/2011, <http://wordnet.princeton.edu>.
2. Mougín F, Burgun A, Bodenreider O. Using WordNet to improve the mapping of data elements to UMLS for data sources integration. *AMIA Annu Symp Proc.* 2006;2006:574-578.
3. Bodenreider O, Burgun A, Mitchell JA. Evaluation of WordNet as a source of lay knowledge for molecular biology and genetic diseases: a feasibility study. *Stud Health Technol Inform.* 2003;95:379-384.
4. Minsky M. Frames: a general representation of knowledge (MIT AI Lab Memo 306). 1974 [cited 6/1/09]; Available from: <http://web.media.mit.edu/~minsky/papers/Frames/frames.html>
5. Minsky M. A framework for representing knowledge. In: Winston PH, ed. *The Psychology of Computer Vision*. New York: McGraw-Hill; 1975.
6. Noy NF, Klein M. Ontology evolution: not the same as schema evolution. *Knowl Inf Syst.* 2004;6:428-440.
7. Gruber T. *Ontology. Encyclopedia of Database Systems*. Berlin: Springer; 2009.
8. Stanford Center for Biomedical Informatics Research. Protege Home Page. 2010 [cited 12/1/10]; Available from: <http://protege.stanford.edu/index.html>.
9. Wikipedia. Potter Stewart. 2010 [cited 4/10/10]; Available from: en.wikipedia.org/wiki/Potter_Stewart.
10. English L. *Improving Data Warehouse and Business Information Quality: Methods for reducing costs and increasing profits*. New York: Wiley; 1999.
11. Conniff R. What's Behind a Smile. *Smithsonian Magazine*; 2007: p. 51-52.
12. Noy NF, Crubezy M, Ferguson RW, et al. Protege-2000: an open-source ontology-development and knowledge-acquisition environment. *AMIA Annu Symp Proc.* 2003;2003:953.
13. Stanford Center for Biomedical Informatics Research. Protege: JDBC Back-end design rationale. 2005 [cited 6/4/09]; Available from: http://protege.stanford.edu/doc/design/jdbc_back-end.html.
14. Zhou M, Geng G, Huang S. Ontology development for insect morphology and taxonomy system. In: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology; 2006.
15. Reiter R. On closed world data bases. In: Gallaire H, Minker J, eds. *Logic and Data Bases*. New York: Plenum; 1978.
16. World Wide Web Consortium. OWL web ontology language: overview. 2004 [cited 9/1/05]; Available from: <http://www.w3.org/TR/owl-features/>.
17. IBM Corporation. IBM integrated ontology development toolkit. 2004 [cited 3/2/09]; Available from: <http://www.alphaworks.ibm.com/tech/semanticstk/faq>.
18. Wikipedia. XML databases. 2009 [cited 9/1/09]; Available from: [The URL is: en.wikipedia.org/wiki/XML_database](http://en.wikipedia.org/wiki/XML_database).
19. World Wide Web Consortium. SWRL – A semantic web rules language combining OWL and RuleML. 2009 [cited Aug 1, 2009]; Available from: www.w3.org/Submission/SWRL/.
20. Das A, O'Connor M. SWRLTab. 2009 [cited Aug 16, 2009]; Available from: protege.cim3.net/cgi-bin/wiki.pl?SWRLTab/.
21. World Wide Web Consortium. OWL 2 web ontology language document overview. 2009 [cited; Available from: <http://www.w3.org/TR/owl2-overview/>.
22. Cregan A, Mochol M, Vrandečić D, Bechhofer S. Pushing the limits of OWL, Rules and Protege: A simple example. 2005 [cited; Available from: <http://www.aifb.uni-karlsruhe.de/WBS/dvr/publications/rove.pdf>.
23. Rettig T. *Tom Rettig's Foxpro Handbook*. New York: Random House; 1990.

24. Harman D. How effective is suffixing? *J Am Soc Inf Sci.* 1991;42(1):7-15.
25. Divita G, Browne AC, Rindflesch TC. Evaluating lexical variant generation to improve information retrieval. *Proc AMIA Annu Symp.* 1998;1998:775-779.
26. Sparck Jones K. A statistical interpretation of term specificity and its application in retrieval. *J Doc.* 1972;28(1):11-21.
27. National Cancer Institute. Common Terminology Criteria for Adverse Events (CTCAE) and Common Toxicity Criteria (CTC). 2009 [cited 9/1/09]; Available from: http://ctep.cancer.gov/protocolDevelopment/electronic_applications/ctc.htm.
28. Nadkarni PM, Marenco LA. Implementing description-logic rules for SNOMED-CT attributes through a table-driven approach. *J Am Med Inform Assoc.* 2009;17(2):182-184.
29. International Health Terminology Standards Development Organization. SNOMED CT machine-readable concept model. 2010 [cited 3/1/10]; Available from: https://thecap.basecampHQ.com/projects/388747/file/35466849/xres_MachineReadableConceptModel_Core-Full_INT_20091001.zip (this URL requires IHTSDO membership to access; membership, however is freely granted).
30. Ceusters WM, Spackman KA, Smith B. Would SNOMED CT benefit from realism-based ontology evolution? *AMIA Annu Symp Proc.* 2007;2007:105-109.
31. Wade G, Rosenbloom S. The impact of SNOMED CT revisions on a mapped interface terminology: terminology development and implementation issues. *J Biomed Inform.* 2009;42:490-493.

14.1

Introduction

Clinical Study Data Management Systems (CSDMSs) support the process of managing data gathered during clinical research. Clinical research involves much more than clinical data management – for example, research-grant tracking and reporting to the sponsor and to institutional review boards (IRBs, also called Human Investigations Committees) as well as financial management. I'll focus on the data-capture, reporting and query aspects, as these are the components that benefit from metadata.

Commercial CSDMS, like EMRs, can be priced in the millions of dollars. Even after the up-front investment, they need a team to maintain. Further, because they need to interface to other systems that exist in the institutions, custom development is still necessary after purchase. In general, CSDMSs are much less mature than EMRs – incredible as it may seem, until a few years ago, a major vendor was selling a system whose GUI interface was a direct port of a VT-100 terminal interface, with almost no consideration given to usability. Even for the usable packages, system-level documentation tends to lag: the development team needs to spend significant effort figuring out the software's internals.

14.2

Differences Between CSDMSs and EMRs

While CSDMSs and EMRs have many similarities that are related to their shared function of clinical data capture, their designs and the metadata sub-schemas differs significantly enough that, despite certain shared elements such as parameter or data-entry-screen definitions, it is not desirable to attempt to use an EMR to perform the functions of a CSDMS. While at least one EMR vendor has made such a suggestion in their documentation, I would liken such a use as akin to trying to eat clear soup with a fork and knife. There are crucial differences, which I now discuss, between the conduct of clinical studies and the management of patients as part of the clinical care process.

The concept of a *study* is fundamental to CSDMSs. The metadata must therefore include numerous tables that capture various aspects of the definition of a study, also called the *study protocol*, in a normalized design. Currently, no EMR vendor has found it economically viable to make the effort to implement these tables, because only a modest fraction of the customer base is involved actively in clinical research.

Let's now go into the details of the protocol definition.

14.2.1

Patients vs. Subjects

Unlike patient-care processes, the individuals on whom data is being gathered are not necessarily sick – for example, they may be healthy volunteers: therefore they are typically called “subjects” rather than patients. The process of data gathering is not necessarily longitudinal, but may be a *one-time effort* without subject follow up. For example the study may be designed as a simple survey, or involve *retrospective* gathering of data culled from some other source (e.g., an EMR).

In the retrospective study, some of the subjects may not even be alive when their data is first entered into the CSDMS. In extreme cases, we may not know much about them other than that they existed. For example, in study of genetic disorders, one may gather data not only on individual subjects with the disorder but on long-dead ancestors, who only serve the purpose of connecting distantly-related individuals (who are still alive and on whom detailed data is gathered) into a larger family tree so as to increase the power of data analysis.

14.2.2

Recruitment and Eligibility

In a care process, patients come to care providers with problems. In a clinical study, the research team reaches out to subjects matching a particular profile of gender, age and disease through advertisements, phone calls, or in-person meetings, and tries to persuade them to participate in a study. The persuasion process (*recruitment*) must involve explaining, in simple language, the risks of the study, not just the benefits. Often, such benefits may accrue only to science and posterity, not to the subject. Subjects are usually compensated financially for their involvement: while the level of compensation is not supposed to influence a subject's decision to participate, it often does, as Robert Abadie reports.¹ Subjects may refuse to participate if they decide the risks are unacceptable to them: further, they have the right to withdraw from the study at any time, without needing to provide a reason for doing so.

Understandably, not all persons will be persuaded to enroll in a study even after multiple encounters with research staff. However, the CSDMS must record contact information about potential subjects, and also keep a record of all encounters, because the recruiting staff, who often work part-time, must be paid for their efforts.

14.2.2.1

Eligibility Criteria

A given clinical study has specific *eligibility criteria* that a potential subject must pass in order to be recruited. For example, if a study involves a disease condition, the subject must have that condition: this is an *inclusion* criterion. Certain studies, e.g., investigation of new drugs for cancer chemotherapy that have been shown to be mutagenic in animal studies, will exclude subjects who are pregnant: this is an *exclusion* criterion. Other studies involve selecting patients in whom standard therapies have failed.

The EMR (or software integrated with it) can be valuable for pre-screening patients who are potential subjects, because many of the parameters that comprise eligibility criteria (e.g., diagnosis codes, normal renal function) may be determined from the record. However, many eligibility criteria are fairly complex, and will be determined only after additional tests are performed through screening procedures. For example, for a particular investigational chemotherapeutic agent, the presence of brain metastasis may exclude them: subjects with neurological symptoms must undergo a CT scan/MRI of the brain to exclude brain metastasis.

The study metadata must record the eligibility criteria for the study. The subject data will record whether a given subject passed or failed individual criteria: again, even if a subject turns out to be ineligible, a record of the screening is required for reimbursement by the study's sponsor. In simple study designs, each eligibility criterion can be recorded as a Yes/No question: for a subject to be considered eligible, the responses to all inclusion criteria must be Yes, and the responses to exclusion criteria must be No.

In more complex designs, especially those driven by data that originates in the EMR, it is preferable to record the primary data (e.g., renal function test values) in a specially designated *case report form* (CRF), and then transform the result into a Yes/No response recorded separately. For example, if an inclusion criteria states that the serum creatinine (a renal function test) must be less than 1.5 times the upper limit of normal, and the lab's upper-limit value is 1.2 mg/dL, a value of 1.7 mg/dL would be transformed to Yes.

14.2.3

The Study Calendar

In clinical care, a patient visits a provider when a problem manifests, and this is inherently unpredictable (though follow-up visits may be scheduled predictably if the problem is not too serious and can be managed on an ambulatory basis). Even then, the patient may visit the provider before the scheduled appointment if a complication, or a new problem, emerges. If the patient is sick enough to require admission to a hospital, the length of stay is variable, and during the stay many interactions with the provider/s are also not predictable.

By contrast, most research studies are conducted in ambulatory (out-patient) settings: most conditions of research interest do not mandate the expense of continuous subject monitoring through admission to a hospital or research center. (Even in major illnesses like cancer, patients are typically in a hospital only for the short duration of chemotherapy,

unless the study involves terminal patients.) Apart from expense, an ambulatory setting is generally more convenient for the subject as well.

Consequently, in clinical research the vast majority of visits are *scheduled*. The schedule of visits (called the *Study Calendar*) does not exist for logistic purposes only: in studies that involve observing change in a disease process (with or without therapy), data analysis is greatly facilitated if all subjects that can be compared at identical time-points. The time-points, which are recorded relative to a “time zero” such as date of baseline screening, are specific to a given study’s design. CSDMS terminology typically employs the term “event” rather than “visit”, because certain critical time-points that drive workflow may not involve visits. For example, 1 week before the scheduled visit date, a “Pre-Visit Reminder” event will involve generation and mailing (or E-mailing) of visit reminders.

In very short-term study designs such as pharmacokinetic studies (which study what happens to a drug administered to a subject), the entire study may last for a single day, and the individual time-points may be minutes or hours apart. In most cases, however, the visits are several days to several months apart. Finally, studies such as one-time Web-based surveys have only a single “event” – the date/time the survey is filled out.

14.2.3.1

Event Scheduling: The Subject Calendar

For a given subject, the actual time-zero date and time (and consequently, the rest of the schedule) depends on that subject’s date of enrollment. A *Subject Calendar* refers to the schedule, as computed for an individual subject based on the Study Calendar. This schedule can be handed to the subject as an appointment calendar, with added information about what will be done at each visit, and anything required of the subject prior to a visit. The extra text can simply be formatted boilerplate that can be associated with the event in the metadata, or the developer may add intelligence by way of conditional printing of certain phrases or sections based on clinical or demographics parameters.

There is normally some leeway with respect to scheduling the actual dates, because certain computed days may fall on holidays. Also, for long-term studies, a given time point has a researcher-designated slack (*window*) that varies with the study and the event. For example, the event that corresponds to the 1-year follow-up may be allowed to occur between 11 and 13 months. For a given event, a subject may schedule any convenient date as long as it falls within the window.

In certain studies that involve significant subject risk, such as evaluation of a toxic medication, the subject may need to visit the care provider or hospital if significant adverse events develop suddenly. CSDMSs must therefore allow for “unscheduled” visits that do not fall on calendar time-points. In this circumstance, the CSDMS behaves somewhat like an EMR.

14.2.3.2

The Event-CRF Cross-Table

At each event, specific actions are performed – e.g., administration of therapy – and units of information gathered in individual CRFs, e.g., for questionnaires, physical examinations,

laboratory tests and special investigations. The association of individual events with individual case-report forms is called the “Event-CRF Cross-Table” For reasons of expense, as well as risk to the patient from invasive tests, all investigations are not carried out at all events, or with equal frequency: expensive and/or tests posing risk of physical harm are much fewer than cheaper or routine tests.

In such experimental designs, the supporting software must typically *enforce the Study’s Event-CRF cross-table constraints*. That is, it should not be possible for a member of the research team, when entering data in real-time or off-line, to accidentally create a CRF for an event where, according to the cross-table, it does not apply. Cross-table constraint enforcement ensures that the values of individual parameters gathered at a specific time point on the Study Calendar for individual patients can be pooled together into summary statistics (and compared across groups of patients, where the study design uses more than one group), because these values reflect the state of that cohort or sub-cohort at that point in time.

The CSDMS should also provide alerts for the research staff about which subjects are due for a visit, and what event that visit corresponds to, so that the appropriate workflow (e.g., scheduling of use of a scarce resource) can be planned. The CSDMS should ideally also support provision of *advance reminders* to subjects either through form letters, phone messages or E-mail. (Reminders are one feature that today’s EMRs support very well: missed office visits translate into lost revenue because scheduled services reserved for a given patient are not utilized.) Timely alerts about *missed visits* are particularly critical, because even if a subject is persuaded to show up for an appointment later, the data for the delayed visit may not be usable if it falls outside that event’s time window.

14.2.3.3

Importance of the Study Calendar and Event-CRF Cross-Table for Analysis

In clinical studies, it is important to determine temporal trends of one or more clinical parameters in a group of patients. Since patients are enrolled in a long-term study at different times (often years apart) absolute date-times are less useful than the relative time-intervals from the baseline time-point. I’ve already stated that time-intervals between any two events will vary slightly from patient to patient. As long as they are still within the permissible window, however, one can still aggregate the values of a parameter for a single event across all patients, using the default time-intervals as specified in the study calendar). The modest error that might result in interpreting the aggregated results is more than offset by the dramatic simplification of analysis.

Similarly, the cross-table can also be used to simplify query and analysis. For a given study, even if two parameters that we want to compare lie in different CRFs, if we know from the cross-table that they are always collected together – e.g., physical examination parameters are always collected along with a basic hematology panel – then they can be analyzed together as if they were part of a single CRF.

In EMR data, where such regularity of data collection is not guaranteed – patients come in whenever they fall sick, and the follow-up intervals also vary greatly because appointments are not always kept — aggregate analysis, as well as comparison of parameters, is much more complicated.

14.2.4 Projects

Sometimes, a *project* may involve multiple related studies performed by a research group or consortium, typically involving a shared pool of subjects – so that certain common data on these subjects, such as demographics or screening data, is shared between studies within the same project. Projects imply several features: a group of users has access to a set of studies within the project, which therefore includes read access to the CRF-related data within each study. However, edit access may be more restricted, because different individuals are responsible for different studies and CRF content.

Most CSDMSs, after log in, will ask the user to select from among the studies s/he is authorized to access. Once a particular study is selected, the user then selects a patient to work with. The user stays in this study until s/he switches to another: this confinement makes it cumbersome to access particular data for the current patient from another study. One way that we've got around this problem in TrialDB is to designate, within the group of studies constituting a project, a **core study**. Read-only access to the CRFs in the core study of a project is always available from any other study in the project. The project team, after designating a core study, decides which CRFs are associated with it.

14.2.5 Differences in Data Capture Between CSDMSs and EMRs

14.2.5.1 CSDMSs Often Need to Support Real-Time Self-Reporting of Subject Data

To ease the research staff's data-entry burden some CSDMSs may support *self-entry* by subjects, or can accept data via bulk import from external systems that support such self-entry. Self-entry is a natural fit for Web-based surveys, and is appropriate for certain CRFs in studies involving self-rating. For example, pain intensity is typically self-reported on an ordinal or analog scale, and certain instruments, such as the Center for Epidemiological Studies Depression Scale (CES-D)² have been used to self-assess intensity of depressive symptoms following radiotherapy for head-neck cancers.

Many subjects are more than capable of using Web-based computer applications for work or personal purposes, so it is reasonable to allow such patients to fill up such CRFs via the Web at a time and location (e.g., home) convenient to them rather than have mandate a visit, or have a staff member interview them over the phone. CSDMSs that support self-entry by subjects allow informatics staff to provide a limited login to subjects, and also to specify which forms are subject-enterable. When the subject logs in, only such forms will be presented for data entry. In a longitudinal study, only the form instances appropriate to the given patient's current event (based on the patient calendar) should be presented.

14.2.5.2

Clinical Research Data Capture is More Structured and Fine-Grained than in Patient Care

In clinical care, a patient may present with any disease: even in most clinical specialties, a broad range of conditions are possible. The only way to capture most information other than vital signs or lab tests is through the narrative text of clinical notes. The flexibility of text explains its preponderance in emergency medicine or primary care. Structured data only arises when a patient is undergoing a specific protocol where the required data elements are known in advance, e.g., for coronary bypass, cataract surgery, or when partial structure can be imposed (e.g., for a chest X-Ray examination).

I've discussed the challenges of trying to process unstructured text in an earlier chapter, so I'll cut to the chase and state that it has little or no role in data capture in a prospective study. In most clinical research, the patients are pre-selected for a specific clinical condition or conditions, so one knows in advance exactly what data elements will be captured. Therefore data capture should be structured right from the beginning. CRFs, for example, maximize the use of elements that require numeric or discrete responses, such as yes/no responses or values selected from a list of choices.

Occasionally, in studies that have dual objectives – i.e., research combined with clinical care – CRFs may contain narrative-text elements like “Additional Comments”, “If Other, Please Specify”, but such elements are relatively modest in number. A good research team will monitor the contents of such fields continuously, looking for frequently occurring textual responses. These provide an opportunity to revise the CRF by increasing its structure through specific prompts for such responses. Apart from making the CRF faster to fill (entering narrative text is always slower than clicking check boxes or selecting items from lists), they improve the data's subsequent analyzability by making more of the questions structured.

14.2.5.3

CSDMS Electronic Data Capture Needs to Be Robust and Flexible, and Efficient to Set up

Data capture in many research settings (notably psychiatry/psychology) is typically far more extensive than in EMRs. Numerous questionnaires have been designed specifically for research problems, and are too lengthy for convenient use by busy caregivers, or by patients who are not compensated for their time in a research study. Because many CRFs are so lengthy, there is a greater risk of the data capture process introducing inconsistency. Consequently, CSDMSs must provide extensive support for real-time data validation. I've discussed most of these issues in depth in Chapter 5. While EMRs increasingly allow sophisticated data capture, it would be safe to say that CSDMSs have defined the state of the art in this regard.

Finally, certain research designs, such as those involving psychometrics, may require the order of questions in a particular electronic CRF to be changed randomly. In computerized adaptive testing,³ even the questions themselves are not fixed: depending on how the subject has responded to previous questions, different new questions will appear.

Note that a given CSDMS may not support every possible feature. For example, I'm not aware of any commercial CSDMS that supports adaptive designs. Such CRFs are developed from first principles, as in the PROMIS consortium.⁴ Also, certain experimental designs, as described in^{5,6} require more than one research team member to evaluate the same subject (or the same tissue from the same subject) for the same logical encounter. Each team member performs an evaluation or rating, and this design intends to estimate inter-observer variability or agreement in an attempt to increase reliability.

Issues of privileges specific to individual user roles arise here too. Some users may only be allowed to view the data in forms, others may also edit their contents, while some with administrator-level privileges may be permitted to lock CRF data for individual forms or subjects to prevent retrospective data alteration. Certain designated forms may be editable only by those responsible for creating their data.

Similarly, the same CRF can be used across multiple studies dealing with the same clinical domain: standard CRFs, such as laboratory panels, can be used in a variety of research domains. When reused, it is desirable to allow *study-level customization*. Here, when a CRF is presented in a given study, only the subset of all parameters that are relevant to that study's objectives will be shown to the user, while non-relevant parameters are hidden: this is possible if the developer employs a strategy of rendering the page dynamically, so that the rendering of a given element and its accompanying caption is conditional to being designated as relevant for that study.

14.2.5.4

Data Entry in Clinical Research May Not Always Be Performed in Real-Time: Quality Control Is Critical

EMRs capture patient-encounter data in real time or near-real-time: CSDMSs are more adaptable to individual needs, supporting off-line data entry with transcription from a source document if real-time capture is not possible, or bulk import of data such as laboratory values from external systems.

To ensure highest quality in terms of minimal missing or unusable data – a major issue in clinical research⁷ – off-line transcription should be as little delayed from the original encounter as possible – e.g., not more than 4 days later. Missing data can occur because source documents can be misplaced or damaged. Bad data-element values are much more likely with paper source documents than with electronic CRFs that support robust interactive validation. Bad-data errors can be corrected only by querying the source document's human originator, and only if the originator remembers the encounter, which is likely only if the encounter is very recent.

Double-Data Entry: Limitations

Double data entry (DDE) is a quality-control method based on the principle of comparing identical input created by two different operators who transcribe the same document separately: input that matches exactly is likely to be correct (unless both operators made the same mistake). Originating during the punched-card era, DDE, in our opinion, has outlived its usefulness.

One practical problem is that many clinical studies are performed primarily in clinical settings where the care-giving objectives are as important as the project's research goals. Here, other than the fact that the data-capture process is driven by a study calendar, the CSDMS behaves as though it were part of an EMR (it may even interface to the latter). The clinicians and nurses are used to the superior ergonomics of an EMR: you can't force them to use paper which is then double-entered off-line by data-entry clerks. (Suggesting that individuals involved in care-giving, who earn high hourly wages, should double-enter data would justifiably get you fired or institutionalized.)

More important, the vast majority of errors in data capture occur at the source: that is why electronic systems provide detailed interactive validation, skips and the rest – you lose all these benefits by moving to paper. As pointed out by in a seminal article by Day et al.⁸, DDE is neither necessary nor sufficient for good data quality, because it does not catch bad-source-data errors.

Today, best quality-control (QC) practices involve close-to-real-time data entry with CRFs maximally using interactive validation, followed by very timely **random audits** of a statistical sample of CRFs against the source documents. The proportion of CRFs is based on criteria such as the criticality of a particular CRF for the study's aims and clinical decision-making, the study's stage (early on, the sampling percentage is higher so as to get an idea of the error rate), and site in a multi-site study (some sites may be more lackadaisical). All questions/parameters on a single CRF are not equally important, and therefore only some (typically critical items used for analysis or decision-making) are audited. This approach, based on QC guru W. Edwards Deming's approach, allows concentration of limited resources in the areas of most potential benefit, as opposed to DDE, which indiscriminately weights every single question on every single CRF equally.

Because timeliness of data entry is so important, a useful CSDMS report will list which CRFs have not yet been entered for scheduled patient visits, or which have been created after a delay longer than that determined to be acceptable.

14.2.5.5

Certain Low-Risk Clinical Studies May Not Store Personal Health Information

In EMR-supported processes involving patient care, the Joint Commission on the Accreditation of Hospital Organizations recommends the use of at least two personal identifiers⁹ to ensure that errors due to treatment of the wrong patient are minimized. In contrast, in certain multi-site clinical studies that involve minimal risk to the patient (such as retrospectively accumulated data), Institutional Review Boards will not permit PHI entry into a CSDMS. Instead, patients are often identified only by a machine-generated "Patient Study ID", and the correspondence between a Patient Study ID and an actual patient is stored in a separate system.

Note that using IDs this way interposes extra manual processes, and magnifies the risks of entering/editing data for the wrong patient. Personal Health Information should *always* be used in clinical studies where even modest physical risk is involved. The researcher may make a faulty clinical decision, such as stopping chemotherapy dose escalation because the white blood cell count has dropped critically – unfortunately, the WBC count

on which the decision was made may have really belonged to another patient, but entered wrongly in the system because every patient was simply an ID, and a fatigued research assistant mentally transposed a digit in the ID. Also, in situations where patients must be followed up, printing of form letters with names and addresses becomes impossible if these cannot be accessed and included except through a very roundabout process.

It is almost compulsory for the Patient Study ID to incorporate extra check digits to prevent an invalid, e.g., digit-substituted or digit-transposed, ID from being accepted. This is necessary even in retrospective data gathering, to prevent errors in analysis. In studies that involve clinical care, this is not satisfactory: far too much reliance is placed on individual human diligence, and the work of the research staff is needlessly multiplied by having to fight with the system. If the airlines' safety checks operated this way, we'd have an airplane crash every second day.

I've witnessed multi-site clinical studies involving patient hazard (such as selecting patients for exploratory laparotomy based on lab values), conducted without PHI use and without check-digit safeguards, simply because of IRB or investigator paranoia. IRBs need to be gently but persistently educated that permanent physical injury and death have somewhat more impact on a patient's welfare than inadvertent privacy loss.

14.2.6

CSDMSs Implement User Roles That Are Specific to Research Designs

14.2.6.1

Supporting Differential Access to Individual Studies

For an institutional CSDMS that supports multiple studies only a handful of individuals – typically, administrators and developers – will have access to all studies. Unlike in the EMR setting, where a patient can be seen by almost any healthcare provider in the organization, access to research subjects' data must be limited to those individuals involved in the conduct of the study or studies in which that subject is participating. The vast majority of users, after logging on, will therefore see only the studies or projects to which they have been given access.

Even here, their *privileges* – the actions they can perform once they are within a study – will vary. For example, an investigator may be a principal investigator in one study, but only a co-investigator in another: therefore certain administrative-type privileges may be denied in the latter study.

14.2.6.2

Representing Experimental Designs

Clinical research often involves an experimental design. In some designs, two or more groups of subjects are given different therapeutic agents (including placebo) or procedures. The designs are typically *double-blinded*. That is, neither the patient nor the caregiver/s dealing with the patient (nor even the chief investigator) knows what a given

patient is receiving: the patient simply receives a custom-formulated medication with one's name on the container.

It is occasionally necessary to break the blinding for a given patient, e.g., if serious adverse effects develop and the patient needs specific therapy to counteract it. Therefore some individuals (typically pharmacists who dispense the medication) are aware of the blinding scheme. CSDMS software is aware of the study-specific privileges of the currently logged on user with respect to blinded data: EMR software lacks this capability.

14.2.7

The Scope of a CSDMS May Cross Institutional or National Boundaries

A given clinical study may often be conducted by a research consortium that crosses institutional boundaries, with multiple geographically distributed sites. Very often, certain investigators in the consortium happen to be professional rivals who are collaborating only because a federal agency initiates and finances the consortium, selecting members through competitive review. Individual investigators would not care to have investigators from other sites access their own patients' data. However, putatively "neutral" individuals, such as the informatics and biostatistics team members and designated individuals affiliated with the sponsor, would have access to all patients.

Even if all consortium investigators trusted each other fully, regulations such as those related to the Health Insurance Portability and Accountability Act (HIPAA) limit unnecessary access of personal health information (PHI) to individuals not directly involved in a patient's care. So biostatisticians intending to analyze the data would generally not care to access PHI. However, PHI such as patient address might be analytically important if one is studying the fine-grained geographical distribution of the condition of interest.

The concept of enforcement of selective access to individual patients' data (*site restriction*) as well as *selective access to part of a patient's data* (PHI) based on the user's role and affiliation is again a critical issue that EMRs do not address.

14.2.7.1

Technological Considerations in Multi-site Scenarios

For trans-institutional studies, CSDMS solutions must increasingly use Web technology to provide access across individual institutional firewalls. By contrast, EMRs, even when used in a geographically distributed setting (as for a network of community-based physicians) are still institutional in scope. Therefore, EMR vendors have been relatively slow to provide access this way: most still employ two-tier (traditional "fat" client-to-database server) access, or access using remote login (through mechanisms such as Citrix).

When a multi-site study is conducted across countries with different languages, the informatics challenges can be significant, as well-described in.¹⁰ The low-tech approach is for every site to develop their own data-capture system. Other than providing full employment for informatics personnel, this duplication of effort has little to recommend it: when CRFs are to be functionally identical across all sites, the likelihood of coordination failure is far too great.

A technologically superior approach is to deploy a single application across all sites, so that all CRF and CSDMS development is centrally coordinated even if actual deployment is geographically dispersed. If Internet connectivity performance is a major limiting factor, then deployment of multiple copies of the same application at different sites is unavoidable in the interests of satisfactory response time. If this is not an issue, then a single Website, hosted in the country where the main informatics team is located, is feasible.

If the latter approach is attempted, one issue that arises is that users at individual sites do not all speak and write the same language. One can be dictatorial and mandate the use of English, but in circumstances where near-real-time data entry is being performed, and the on-screen prompts are intended to facilitate data entry, such a decision may impose an unacceptable handicap on non-native speakers.

14.2.7.2

Software Localization

The process of tailoring the user interface of a single body of code (e.g., a program like Microsoft Word) to a specific language and culture is called *localization*. Localization involves more than just presenting the appropriate phrases of text: the layout of prompts and input fields must take into account that some cultures (Arabic, Hebrew) read and write text right-to-left. The fundamental issue in localization – separating the interface from the code as cleanly as possible, so that the former can be modified without changing the latter – was first addressed on the Apple Macintosh platform, whose system developers recognized the problem and provided operating-system-level software support to deal with it. Every GUI-based environment, including Windows, now facilitates localization.

The most essential step in localization is the creation of “*resources*”: text strings, graphical icons and the like, which are stored in files and assigned symbolic names. When you create your program, any part of it that needs to present text to the user – in the form of menus, dialog boxes, prompts and error messages – avoids hard-coding English phrases in the program code. Instead the code refers to a given resource only by its symbolic name. When the program runs, the resources referred to are loaded into memory. To localize an existing program for a different language, a bilingual individual (who is not necessarily a programmer) translates each phrase and creates a language-specific resource file. Sometimes, even icons that contain textual elements (such as the red STOP sign octagon, a common metaphor) tend to be culture-specific.

In *static localization*, the approach used for software that is physically distributed on magnetic or optical media (e.g., the Dutch version of MS-Excel), the program code is combined with the language-specific resources to form a language-specific executable program. In *dynamic localization*, which can be used for Web applications, multiple language-specific resource files are kept on the same machine, and one of them is active based on the current user. Some systems can prompt the user for their choice of language initially: however, it is also possible for a Web server to sense the default language of a connecting Web browser (this information is part of the handshake), and load the appropriate resource file. Some Web technologies, notably Microsoft ASP.NET, make dynamic localization considerably easier than others.

One practical issue in CSDMSs is that, because of the rapid development of CRFs to meet research needs, the text strings that are presented in the CRF are stored as metadata within the database rather than as string resources. These include the captions of parameters, parameter groups and CRFs, section headings, scripts/ instructions, nutshell help, and even the descriptive phrases associated with enumerated data-type attributes that are presented as pull-downs, radio buttons or list boxes. It is straightforward to generalize your schema design for these elements by introducing many-to-one relationships and adding an extra “language” column in the “many” table. This way, based on the desired language, the appropriate elements can be presented. Of course, you should also designate one of your languages (typically English in the US/UK) as the default – you can hardly be expected to support every language ever spoken by humans.

While several commercial Web sites implement dynamic localization, to the best of our knowledge no existing commercial CSDMS has employed it. The programming is not particularly hard, but the design approach requires discipline: existing code will need considerable overhaul. Code-development environments like Visual Studio will assist you greatly in terms of porting non-metadata-based user interface elements: if you turn localization warnings on, hard-coded strings in your code will be flagged.

14.3

Data Storage in CSDMSs

There are two types of approaches in the organization of CSDMS data. The more widely employed approach is to use EAV. High-end systems such as Oracle Clinical and ClinTrial (now also a recent Oracle acquisition) do this. From the perspective of the clinical data sub-schema, the changes required are very modest compared to the design for an EMR. The top-level of the data table (Forms_Data of the User Interface chapter) also includes a Study (Protocol) ID, identifying the Study where the form’s data was collected.

The other approach is to design a conventionally modeled database schema for each study. Designing schemas from scratch each time a new study comes along is needlessly labor intensive, and the software packages that use this approach therefore also use metadata. After definitions of parameters and CRFs are created, a program generates a conventional database schema as well as a user interface for each CRF. Each CRF typically corresponds to a separate table, though if it includes repeating-group data, then these will go into their own tables, which are linked many-to-one to the main CRF table.

The difference between the metadata design in this approach and the previous one is that the metadata is stored, not in a subschema (as with EMRs) but in a separate database.

14.3.1

Drawbacks of the Separate-Database Approach

In either approach, the metadata definitions are also reusable. However, in the separate-database design, the “metadata” database may not necessarily be accessible to the

customer: typically, the it is the *vendor* who operates the metadata schema, and generates a database-plus-application based on the customer's specification: the customer only gets access to the latter.

For customers with little in-house software expertise, this approach may, paradoxically, be preferable. I've said that metadata are the crown jewels of a system, and it may be corrupted through accidental introduction of errors by insufficiently knowledgeable users. With the metadata operated only by trained vendor staff, this risk is reduced. However, for large institutions with their own IT infrastructure, who might wish to use their data as a resource for analysis and knowledge discovery across multiple studies dealing with the same family of disease conditions, such an approach has some drawbacks, as now discussed.

14.3.1.1

Difficulty in Reusing Parameters

The more you can reuse parameter definitions across multiple studies, the more valuable your data becomes if you eventually decide to perform meta-analysis. However, *it takes more effort (and discipline) to reuse parameters than to simply define new ones*. Reuse means searching the metadata by keywords to see if a parameter has previously been defined. If you find likely candidates, you must inspect their definitions – the long description, data type, enumerated values, range checks, etc. – to make sure that their semantics are indeed identical to what you need. If the semantics are not what you need, you have to define a new parameter.

While you may assume that the vendor staff knows the product better than you do, you can't assume that their knowledge of the diseases that you are studying, or concerns for meta-analyzability, will match yours. If your vendor has multiple study designers whose productivity is measured by forms designed per week, and/or significant staff turnover, potential problems exist. Remember that the vendor has an economic incentive to design your study with the least possible investment in time and effort that is compatible with CRFs that work: long-term concerns like reuse and meta-analysis get short shrift.

As the metadata grows more voluminous as the number of studies grows, it tends to follow the second law of thermodynamics: the entropy (disorder) tends to grow. This problem manifests as multiple definitions that have identical semantics but different parameter names (i.e., undetected redundancy), or as parameters that are nearly identical but differ in subtle ways that are not documented clearly. Treating subtly different parameters as semantically identical in a future meta-analysis can often be unjustifiable scientifically.

Messy metadata is a risk no matter whether you use the EAV, single-database, or separate-database approach, and it is not limited to CSDMSs: it occurs in the EMR context as well. Naïve and/or time-stressed study designers, who do not understand the long-term importance of rigorous data element definition and reuse, cannot always be trusted to use metadata-browsing tools even if you provide them. However, if you have control over the metadata repository, you can take responsibility for its quality, by training, monitoring and controlling your own team. It is much harder to ensure metadata quality when the metadata's creators are not directly answerable to you. In operating TrialDB, after having to clean up the messes created by power users we had mistakenly trusted to create element

definitions, we learned very soon to limit metadata-creation privileges only to those individuals who reported to us. Without continued vigilance, you may only discover problems years after they were originally introduced.

14.3.1.2

Extra Effort in Creating Data Warehouses and for Meta-analytic Data Extraction

Data that is housed in a large number of separate physical databases is also more laborious to integrate into a warehouse. Because the individual studies' schemas bear little resemblance to each other, there is no obvious recipe for merging the data in a manner that might facilitate reanalysis. You must either throw all the tables together in packrat fashion, or homogenize the data into EAV form prior to extracting data subsets of interest. Either case results in a state of affairs that no one besides a job-security-conscious contract programmer would find satisfactory.

14.4

CSDMS-Related Processes During Different Stages of a Study

After discussing the special needs that CSDMSs meet, we now consider CSDMS-related matters that arise in the different stages of a study. In chronological sequence, these stages are:

1. Study Planning and Protocol Authoring
2. Recruitment/eligibility determination (screening)
3. Protocol management and study conduct (including Patient-monitoring and safety)
4. Analysis and reporting.

Most of the foregoing text has dealt with issues relating to Stage 3, though CRF design and Calendar setup are part of Stage 1. We now discuss issues related to planning.

14.4.1

Study Planning and Protocol Authoring

Ideally, the protocol is developed by a team consisting of one or more of each of the following: a clinical investigator, a biostatistician and a data management expert/ informatian. The biostatistician works with the investigator to conduct a study of the relevant literature to determine previous research, availability of research subjects, relative incidence in the population of the condition(s) of interest, epidemiology of the outcome, the time course of the condition, risk factors and vulnerable populations. Knowledge of these factors will determine an appropriate experimental design. If the design involves two or more groups of subjects, knowledge of the risk factors and co-morbidities will suggest

strata for randomization. A power analysis can determine how many subjects need to be recruited for the study to have a reasonable chance of being able to prove its main hypothesis. If data is available on the annual number of cases presenting at the institution, sample size determine will provide an idea as to how long the study must remain open for enrollment of new subjects, or even if it is possible to enroll all subjects from a single institution: sometimes, multiple sites are needed to get sufficient power. A useful freeware package for power analysis is PS, developed at Vanderbilt University by Dupont and Plummer.¹¹

14.4.1.1

The Informatician's Role

In my experience, one of the greatest dangers in investigator-initiated clinical studies is under-budgeting for the data management effort. I wish I had a dollar for every occasion where I've been approached by an investigator who's obtained funding of some sorts for a clinical study, and who then asks, after the fact, for data-management support that can be tailored to fit a pittance of a budget. Unfortunately, CRF and report design, administrative chores, end-user training, documentation and help-desk functions take up human resources that have to be paid for even if existing CSDMS software can be used. For an informatics team, participation in a study where they are very likely to find themselves expending more resources than they are being compensated for becomes, in the undying words of Walt Kelly's Pogo, an insurmountable opportunity that is best declined.

Electronic protocol design involves the following tasks:

- Setting up the Study Calendar.
- Designing the CRFs for the study (or reusing other CRFs that have been previously created for other studies).
- Designating which CRFs apply to which event on the calendar.
- Designating user roles and the privileges associated with each.
- Specifying the options required for a given experimental design, such as blinding, hiding of PHI.
- Specifying eligibility criteria. (More on this shortly.)
- Identifying the types of reports that will be needed, and designing these, as well as devising a data analysis plan. (More on this later.)
- Determining QC parameters for timeliness and accuracy of CRF entry.
- Creating a manual of operations. The package I'm partial to for documentation creation is Adobe RoboHelp™ (which creates a Website, hard copy and downloadable content – e.g., CHM files – from the same source), but I expect there are probably cheaper alternatives around. As your operation gets more streamlined, a lot of the manual's content may be taken from standard text that is study-independent, but which reflects the way things work in your institution. Other content, such as help for individual CRFs, is of course study-specific.
- Devising and documenting a data safety monitoring plan (DSMP), which ensures adequate oversight and monitoring of study conduct, to ensure participant safety and study integrity. At the least, the DSMP should include a plan for adverse event reporting (see later) and a Data Safety Monitoring Board if the intervention has the potential of significant risk to the patient.

- Iterative testing and refinement of the CRFs.
- Role-based User training and certification. Expect this to be ongoing process: research jobs are often considered stop-gaps that don't pay particularly well, and there tends to exist a revolving door with respect to the lower-level team members.

14.4.1.2

Recruitment and Eligibility Determination: Pre-screening Using the EMR

The most effective approach to recruitment for subjects with a clinical condition (as opposed to healthy volunteers) involves close integration with the institutional EMR. If your development team has access to the EMR's API – with the balkanization of informatics responsibilities in large institutions, such access is by no means guaranteed – consider building an ad hoc query framework for the latter, because the queries that identify patients, based on diagnosis codes or laboratory values, will have to be run, not once, but repeatedly even for the same study. You will almost always need to operate against a warehouse rather than the production system. Of course, certain other criteria (such as whether the patient is currently pregnant) need to be determined through subject interviews or testing.

14.4.1.3

Scheduling Enrolled Subjects

CSDMS software may often provide its own scheduler, but should ideally be well integrated with an EMR's scheduling system if the subjects are patients and the hospital (as opposed to a clinical research center) is primarily responsible for providing care.

Robust software generates reminders for both staff and subjects, and also allows rescheduling within an event's window. The period of time prior to a visit date for which changes to the visit date are allowed depend on the nature of the visit: if the visit involves access to a relatively scarce and heavily used resource such as a Positron Emission Tomography scanner, changes to the schedule must be made well in advance.

One report driven off the patient calendar is so critical for longitudinal studies that in its absence the research team would find it hard to function. This is a "to-do" list of patients whose visits are scheduled in a particular period in the future – e.g., from tomorrow to the next two weeks – along with what event/visit applies to each patient, and any special instructions for the research staff (i.e., a nutshell description of the workup at that visit). In some academic institutions, you have research coordinators who manage multiple studies, and this list must span all the studies that the coordinator is responsible for.

14.4.2

Protocol Management and Study Conduct

All patients never enroll in a study at the same time. Tasks related to ongoing study conduct include:

- Tracking the overall enrollment status by study group, demographic criteria, and randomization strata.

- Importing external source data into the CSDMS, using electronic rather than manual processes where possible.
- Monitoring and reporting of protocol deviations, which are changes from the originally-approved protocol, such as off-schedule visits. Protocol violations are deviations that have not been approved by the IRB. Major violations affect patient safety/rights, or the study's integrity. Certain major violations, such as failure to document informed consent in the CSDMS, or enrolling subjects who fail to meet all eligibility criteria, can also be forestalled by the CSDMS software refusing to proceed with data capture for that patient until these issues are fixed.

Other protocol deviations relate to workflow or wholesale CRF revisions that have not been vetted by the IRB. I have encountered investigators who, after the protocol was supposedly finalized, asked for study design and CRF changes that would have doubled the number of blood draws per patient, or greatly increased the patient's interview time. To prevent these violations, the informatics team must refuse changes without documentation of official approval.

- Supporting occasional revisions to the protocol to meet scientific needs, including a *modest* degree of CRF modification.
- Creating new reports to answer specific scientific questions.
- Monitoring the completeness, timeliness and accuracy of data entry.

14.4.2.1

Patient-Monitoring and Safety

In clinical studies involving therapeutic interventions, monitoring for adverse effects (AEs) is critical. The NCI's CTC AE anchored vocabulary for side effects, mentioned earlier, has been used in areas beyond cancer, though it is less useful for subjective symptoms. CTC AE's use is tied to workflow: a grade-3 or greater AE *must* be reported to the sponsor and other collaborating sites as well as to the local IRB.

The process of grading AEs manually is laborious enough that it can benefit from electronic help. AEs based on quantitative laboratory values can be graded based on simple numeric range checks, as reported by Richeson et al.¹² Dealing with subjective findings is more challenging, but Wang et al.¹³ describe a prototype approach for pharmaco-vigilance based on NLP of clinical text.

14.4.3

Analysis and Reporting

Most CSDMSs implement a variety of standard reports. Among these are:

- Reports related to enrollment of subjects, subcategorized by demographics or randomization strata. Reporting details of subjects screened vs. subjects actually enrolled.
- Reports of screened subjects who failed individual eligibility criteria.
- Reports related to Adverse Events.

- Reports related to completeness, accuracy and timeliness of data capture/entry.
- Reports summarizing the numbers of patients in different stages of the study (based on events).
- Reports of patients who terminated from the trial abnormally – e.g., because of refusal to continue, adverse events, etc.
- Workflow reports related to the study calendar – to-do list, missed visits, data-entry-timeliness reports.

In addition, each study will generally require specific, custom-designed reports related to its scientific objectives.

For the purposes of *analysis*, a CSDMS must provide bulk-export capabilities. This topic has numerous ramifications, which we discuss in a subsequent chapter.

14.5

Summary

CSDMSs may use the same underlying storage approach (EAV) that EMRs do, but their metadata structure, and the requisite functionality, is so different from that of EMRs that attempts to meet both needs in a single design are unlikely to be successful. An important category of functionality is that related to the Study Calendar. From the implementation perspective, the real-time load may be less than that an EMR needs to support, but form-design capabilities, as well as localization for multi-language scenarios, is more challenging. User-role management is also different because not every study needs to store personal identifying information, and in those that do, not all users may need to see such information.

References

1. Abadie R. *The Professional Guinea Pig: Big Pharma and the Risky World of Human Subjects*. Durham: Duke University Press; 2010.
2. Raloff L. The CES-D scale: a self-report depression scale for research in the general population. *Appl Psychol Meas*. 1977;1(3):385-401.
3. Wikipedia. Computerized adaptive testing. 2010 [cited 4/10/10]; Available from: en.wikipedia.org/wiki/Computerized_adaptive_testing.
4. Cella D, Riley W, Stone A, et al. The Patient-Reported Outcomes Measurement Information System (PROMIS) developed and tested its first wave of adult self-reported health outcome item banks: 2005–2008. *J Clin Epidemiol*. 2010;63(11):1179-1194.
5. Thwin SS, Clough-Gorr KM, McCarty MC, et al. Automated inter-rater reliability assessment and electronic data collection in a multi-center breast cancer study. *BMC Med Res Methodol*. 2007;7:23.
6. Van den Broeck J, Mackay M, Mpontshane N, Kany Kany LA, Chhagan M, Bennish M. Maintaining data integrity in a rural clinical trial. *Clin Trials*. 2007;4(5):572-582.

7. Wisniewski SR, Leon AC, Otto MW, Trivedi MH. Prevention of missing data in clinical research studies. *Biol Psychiatry*. 2006;59(11):997-1000.
8. Day S, Fayers P, Harvey DR. Double data entry: what value, what price? *Control Clin Trials*. 1998;19(1):15-24.
9. Joint Commission on Accreditation of Hospital Organizations. National Patient Safety Goals. 2008 [cited 11/03/10]; Available from: http://www.jointcommission.org/PatientSafety/NationalPatientSafetyGoals/08_hap_npsgs.htm.
10. Frank E, Cassano GB, Rucci P, et al. Addressing the challenges of a cross-national investigation: lessons from the Pittsburgh-Pisa study of treatment-relevant phenotypes of unipolar depression. *Clin Trials*. 2008;5(3):253-261.
11. Dupont W, Plummer W. Power and sample size calculations for studies involving linear regression. *Control Clin Trials*. 1998;19:589-601.
12. Richesson RL, Malloy JF, Paulus K, Cuthbertson D, Krischer JP. An automated standardized system for managing adverse events in clinical research networks. *Drug Saf*. 2008;31(10):807-822.
13. Wang X, Hripcsak G, Markatou M, Friedman C. Active computerized pharmacovigilance using natural language processing, statistics, and electronic health records: a feasibility study. *J Am Med Inform Assoc*. 2009;16(3):328-337.

Retrieving data from a heterogeneously modeled database is understandably complex because the same conceptual operation requires different types of queries depending on how the class of data that you wish to retrieve is physically organized. Thus:

1. For conventional (columnar)-structured data, the conceptual operation and the query mechanism are identical. A query that fetches only attributes of interest (i.e., without aggregates), even across multiple tables, can usually be effected with a single SQL statement that has a simple structure – that is, it has no subqueries.
2. For EAV-structured data, query of data based on combinations of values of attributes takes many more steps than for columnar (traditionally modeled) data: the AND, OR and NOT operations must be substituted respectively by the set operations of intersection, union and difference.
3. For hybrid classes, a combination of approaches would be necessary, based on whether the attributes you are fetching are columnar or EAV-modeled.
4. Querying a mixture of conventional, EAV, and hybrid classes is the most difficult. It requires decomposing the query task into individual operations on each class, and then combining the results.

If you try to manually compose queries against any data where at least some of it is represented in EAV form, you will find yourself continually searching the data dictionary tables to locate the class and attribute IDs of interest so you can use these in your query. To improve productivity, it is desirable to build a framework to at least partially automate operations and minimize repetitive and error-prone hand-coding.

In this chapter, I first classify the types of data retrieval operations that are needed in a production system. I then consider the various challenges of framework-building for data retrieval. I then consider the individual types of retrieval operations and the special considerations that apply to each.

15.1

Types of Retrieval Operations

In general, there are three kinds of query capability that you need to implement: I list these in order of frequency of use:

1. *Simple searches*, generally restricted to a single class, or classes closely related to this class, for the purpose of being able to browse details of the records of interest, edit or delete them. While one can create shortcuts for very commonly used operations – such as searching for a patient by last name – in the general case, what is desired is a *Query by Form* (QBF), where one can specify a *combination* of criteria to reduce the number of records returned – e.g., search by last name, first name and date of birth.
2. Standard “canned” *reports* that allow various users of the system to do their jobs. Such reports allow a limited degree of flexibility, by allowing parameterization – for example, allowing the user to specify date ranges or filters that make the report’s output as specific as possible. Occasionally, one may allow the person who views records in a report to jump to a Detail View of a record of interest to allow inspection and editing. There are two aspects to designing such reports: composing the SQL that fetches the desired data after accepting the user’s parameters, and formatting the returned data for display.
3. Complex *ad hoc query* (AHQ) involving multiple classes. By definition, this is completely unpredictable in nature. The people performing AHQ are technically savvy users, typically those who perform analytical roles. Commercial query tools are remarkably useful in supporting query composition for conventionally modeled data through a GUI – I’m a fan of Microsoft Access’s Visual Query By Example (VQBE) – but understandably these have not been designed with EAV or heterogeneous data models in mind. While you can still query EAV data with such tools if you first get the Class and Attribute IDs of interest and then use these as query criteria, a framework that helps the user perform limited AHQ through a GUI is desirable.
4. *Bulk extraction of data*: Bulk extraction operations pull data from the entire database or from large chunks of it, typically a class at a time, converting it into conventional columnar form if necessary so that one can apply standard analytical tools (such as spreadsheets or statistical packages) to it. Bulk extraction may be regarded as similar to reporting operations, except that the issue of data formatting for presentation purposes doesn’t apply. Also, under certain circumstances, one can take advantage of various kinds of regularity in the data so as to improve extraction performance.

15.2

General Principles of Framework Creation for Data Retrieval

15.2.1

Complex Problems Defy Simple Solutions

The problem of creating a single approach that will allow arbitrarily complex query to be composed and yet be fully intuitive to naïve users – e.g., by operating through a GUI – is one that has not been solved, even in the context of schemas that consist entirely of conventionally structure tables. There is no reason to believe that it will ever be solved, because it is something of a contradiction in terms. I’ll now explain my reasons for this assertion.

In general, an interface that is optimal for simple tasks breaks down for more complex ones, and alternatives need to be provided. For example, the Apple Macintosh, introduced in 1984, was widely praised for ease of learning and use, but it frustrated power users who needed to create simple programs that coordinated existing applications, but were unable to do so. Many such users were programmers who were comfortable writing UNIX shell scripts: the flip side of WYSIWYG turned out to be “what you see is all you’ve got”. Finally in 1991, Apple provided AppleScript, a programming language somewhat COBOL-esque in its wordiness, for such tasks.

The problem of a universal query interface is as intractable as that of creating a universally optimal user interface for computer operation.

15.2.1.1

Approaches to Solving the Data Retrieval Problem

SQL stands for Structured *Query* Language. It was originally intended only to fetch data; the ability to create, delete and change data was added later, and the ability to define the structure of your database was added still later. The part of SQL that developed first has evolved the most: if you study a textbook on SQL, you will find that the part devoted to the SELECT statement, with all its various nuances, such as aggregates, grouping, sub-queries and set operations is larger and more complex than the rest of SQL combined. Even create, delete and update operations require knowledge of the SELECT statement if you need to operate on a table based on criteria specified on other tables.

The graphical query tool I rate as the overall best for its blend of power and ease of use is Microsoft Access’s Visual Query By Example (VQBE), when used to query high-end RDBMSs. Even VQBE, however, has no graphical metaphors for composing expressions (using arithmetic operators or function calls), correlated sub-queries and set operations. Users who have not studied advanced SQL are not even aware of the last two. The developers who need these capabilities must specify them by typing SQL. Once you enter typing mode, however, VQBE cannot prevent syntax errors: when such errors are reported, the diagnostics aren’t particularly helpful to novices. Microsoft has deliberately limited their product with good reason: I can’t think of a natural visual metaphor to represent sub-queries, for example.

The problem of querying heterogeneous data – one that VQBE’s graphical metaphor was never intended to address directly – adds another dimension of complexity to the query problem. You should therefore consider yourself fortunate if any tool that you create meets 80% of the needs of your user base. The challenge is to identify that 80%.

15.2.2

Focusing Your Approach

Given enough time and human resources, it may be possible to implement your own version of VQBE which is fully heterogeneous-schema-aware. You don’t have such a luxury, however, so you should limit your framework. The first step is to avoid the trap of

conceiving of a single “user”: users are not homogeneous. In other words, you can’t have a single user interface. Consider the following:

- The needs of expert analysts are different from those of more casual users. In my experience, less than 5% of your users will be able to use ad hoc query capabilities productively, or even care to: most will simply ask for a report that can be run with a click, or on a regular schedule. No matter how responsive your AHQ interface may be, using its capabilities requires the user to formulate a specific question or hypothesis, and this in turn requires considerable knowledge about the data and the way it is organized. For example, even if the user is not required to know what EAV is, s/he must still know what parameters are being collected and how these are defined: this often requires exploration of the metadata, something that casual users have neither the time nor inclination to do.
- Though most needs of casual users are met through canned reports, the developers who need to create such reports must work with the same query tool that the analysts use: this will reduce manual effort in composing the SQL that fetches the required data.
- Everyone, however, needs to be able to execute simple searches efficaciously in multiple circumstances, and therefore implementing Query by Form capability takes high priority.

15.2.3

Ad Hoc Query: Implementation Considerations

Implementing Query by Form is far less challenging than reporting, while Ad hoc query is the most challenging of all. This is because the first step in data retrieval is to let the user expeditiously identify attributes of interest. In QBF, the attributes of interest are already presented in a form: the user merely needs to specify criteria. The problem is that EAV-modeled attributes’ definitions are stored in metadata, and are therefore inaccessible to a non-EAV-aware interface like VQBE, which can only locate attributes that are represented as physical columns. Therefore, the user must browse the metadata continually (in a separate window) to locate the desired attributes, and then enter their IDs into the query interface. Of course, because the data may be stored in data-type-specific EAV tables, the appropriate EAV table-names also need to be added into manually composed SQL.

Therefore, implementing an AHQ interface for heterogeneous data requires that your interface should blur the distinction, for the user, between row (EAV) and column-modeled attributes, treating all attributes as though they were the latter. For EAV-modeled attributes, it should silently pick up the attributes’ IDs (and associated information, such as data type and class to which it belongs), and compose segments of EAV-aware SQL into which these are then inserted automatically.

15.2.3.1

Utilizing Your Problem’s Constraints

The SQL that needs to be generated depends on the problem you are addressing, which in turn influences the variant of the EAV model that you implement in your schema: it is different for the EAV/CR and clinical data scenarios, because in the former attributes have

only one level of grouping (classes) while in the latter there are two levels (groups and forms). Also clinical-data attributes are sparse, but attributes in EAV/CR scenarios are not.

The optimal user interface also varies with the kind of data that your system is managing. From my own experience, an overly general query interface will be too tedious for users: try to take advantage of the constraints of your problem to provide a good user experience. I'll illustrate later how in the CSDMS situation, the common situation where query is limited to a single study can change the way the user will prefer to identify attributes of interest.

You will have to develop iteratively in rapid-prototyping mode, and revisit your design frequently based on how your users use what you have created, and what they begin asking for. One tip about implementation strategy: Your ad hoc query users are few, and are likely to use high-end desktop machines. So consider the use of two-tier, fat-client designs that, while not scaling well, allow more rapid iterative refinement, and take advantage of the desktop's capabilities.

No matter what your query framework does, it should provide a means of storing the SQL that it generates. This SQL can be used as a building-block in reports, as well as in more complex queries that the user or developer can compose using a commercial tool. For example, VQBE allows queries to be used as components of higher-level queries – to the latter, the former are indistinguishable, for practical purposes, from a table. Such a combined “divide and conquer” strategy will make the user more productive than the use of a commercial query tool (or your query tool) alone.

15.2.3.2

A Word of Advice for Academic Informaticians in Service Settings

Query approaches for EAV data seem to be a fertile topic for PhD dissertations, and I've seen more than one academic publication describing a new dialect of SQL that will address the problem – if only vendors would implement it. Few projects go beyond the proof-of-concept stage. Unfortunately, the creation of production-quality code and graduate-student responsibilities do not mix well. The former involves a lot of effort (version control, regression testing) that grad students may consider a diversion from the dissertation or publication perspective, and what is interesting from a research viewpoint may not be practicable in production settings.

The most infamous example of grad-student database code was PostGRES, developed by Michael Stonebraker's group at UC Berkeley, which became Illustra, which was eventually acquired by Informix, in turn acquired by IBM. PostGRES (and Illustra) were so notoriously fragile – the joke was that PostGRES would crash on you every half an hour – that version 6.0 (PostgreSQL, which became open-source) was essentially a complete rewrite, a point that its new developers took pains to emphasize.

15.2.4

Infrastructure Considerations

The operations of reporting and bulk data extraction are typically resource-intensive enough that it is preferable to run them, not against a production system whose contents

are being updated interactively by numerous users, but against a data warehouse that runs on separate hardware, and whose contents are not more than 24 h older, say, than the production data. Complex ad hoc query can also be resource-intensive, and by default, should also be run against a warehouse: only in exceptional circumstances, and only with a very few privileged users, can it be permitted to run against a production system.

The warehouse is typically generated from the production system on a batch basis – ideally, at least nightly, so that the warehouse data is not more than 24 h out of date. With modern hardware and database technologies such as transactional replication, the process of pulling data out of a production system can be greatly streamlined and even real-time if you need such a capability. As the excellent series of books and articles by Ralph Kimball and his colleagues will tell you, however, the process of creating a warehouse is much more than simple physical cloning: the extraction, transformation and load (ETL) process must also check for inconsistencies or errors in the production system, and implement strategies for temporarily dealing with these – though ultimately such problems must be fixed in the source. I’ll deal with the metadata-related issues of data warehousing in a later chapter.

15.2.4.1

Warehouse Schema Design

The difference between the production schema and the warehouse schema is that the latter’s columns are much more extensively indexed, and certain data are *denormalized* by merging columns from one or more tables into a single table, to avoid having the RDBMS repeatedly join these tables.

It is also desirable to create views that simplify access to individual attributes. Modern RDBMSs allow such views to exist as physical tables: these are called *materialized* or *indexed* views – the former is an Oracle term, the latter is used by MS SQL Server.

In our own warehouse schema, we denormalize the metadata tables extensively, so that when you fetch information on an attribute, you can directly access all the information that you would ever need to know about it, including higher-level groupings. Metadata is modest in volume, but it is consulted so often that such denormalization helps significantly.

For our CSDMS data-subschema, we combine the `Form_Data` and `Group_Data` tables in the data-subschema diagram of [Chap. 5](#) into a physical table, also incorporating essential Study Calendar metadata (specifically, the Event name and Event Serial Number). For the data-type specific EAV tables, we create an equivalent, non-indexed view for each table that combines the EAV table with the denormalized table above. (We originally tried combining the EAV tables with the higher level tables, but the storage requirements ballooned without a corresponding improvement in performance.)

Dealing with Enumerated Data Types

Enumerated data types are used so extensively in both clinical data capture and EAV/CR schemas and so access to such data must be facilitated.

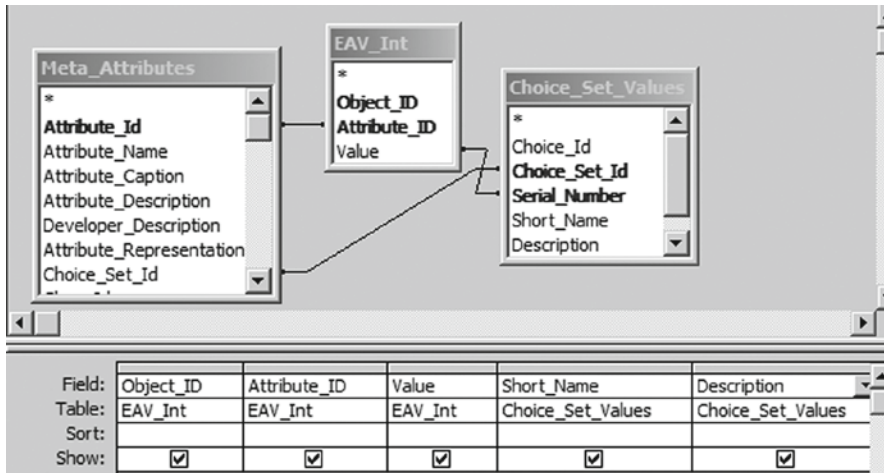


Fig. 15.1 Creating a view to simplify access to enumerated data. The SQL corresponding to this view is: Create view Enum_View as Select Object_ID, I. Attribute_ID, Value, Short_Name, Description from EAV_Int I, Meta_Attributes Att, Choice_Set_Values CV where I. Attribute_ID = Att. Attribute_ID and I. Value = CV. Serial_Number and Att. Choice_Set_ID = CV. Choice_Set_ID

We create an indexed view that deals specially with enumerated data: this merges the modified EAV_Int table with the **Choice_Values** table that holds the textual descriptions of all enumerations, joining on the shared *Choice_Set_ID* column. For example, for the schema of the EAV/CR chapter, we reproduce the requisite view's design both pictorially and as SQL in Fig. 15.1.

Simplifying Access to Object Data Types

In the EAV/CR schema, the EAV_Objects Table has the purpose of recording details of attributes whose values happen to be Objects. This table stores two Object IDs (for Entity and Value) and an Attribute ID. Since Object IDs are essentially meaningless to the user, who thinks in terms of Object Names (which are stored in the Objects table), it is desirable to simplify presentation of the details of a row in the EAV_Objects table. The design of a view for this purpose is shown in Fig. 15.2. The Objects table is used twice in the view under separate aliases, Object_Parent and Object_Child. The SQL for the view is shown below.

Create view Objects_View as

```
SELECT Object_Parent.Class_ID AS Parent_Class, Object_Parent.Object_Name AS Parent_Object_name, EAV_Objects.Object_ID, EAV_Objects.Attribute_ID, EAV_Objects.Value, Object_Child.Class_ID AS Child_Class, Object_Child.Object_Name AS Child_Object_Name
```

```
FROM (Objects AS Object_Parent INNER JOIN EAV_Objects ON Object_Parent.Object_ID = EAV_Objects.Object_ID) INNER JOIN Objects AS Object_Child ON EAV_Objects.Value = Object_Child.Object_ID;
```

I will later demonstrate the use of this view (which can be materialized in a warehouse) for Ad Hoc query on EAV/CR-modeled data.

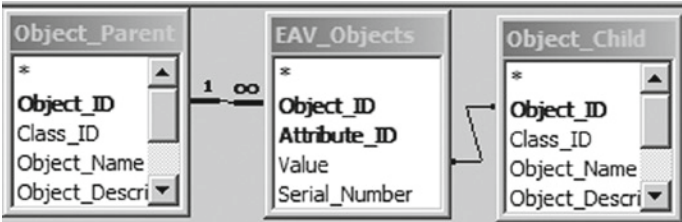


Fig. 15.2 Creating a view to simplify access to the details of hierarchical object information

15.2.4.2
Caching Metadata

Your warehouse database server should be fully loaded with respect to RAM. Since practically all operations consult the metadata tables, it is worth considering ways to improve access to these tables. As I’ve stated several times, while metadata that defines data elements has a complex structure, its total volume, in terms of number of rows, is relatively modest compared to the data – even for a large EMR, it would be highly unlikely to exceed 100 MB.

Therefore it is a good idea to cache metadata in RAM – all of it, if possible. Some DBMSs, notably Oracle, let you configure individual tables (in the CREATE TABLE statement) that let you specify how much of the table’s contents should be in RAM. MS SQL Server once had a command that let you “pin” a table’s contents into memory, but this was disabled after SQL Server 2008.

15.2.5
Restricting the Output Based on User Access Permissions

No matter what query mechanism is being used, users must see only the information that they are authorized to access. In the CSDMS situation, the most important barrier is site restriction: users can see only see the data of patients from the site/s with which they are associated: this is *row-level* access control. I’ve already discussed earlier the issue of restricting permissions to classes, and attributes within a class in the EAV/CR scenario.

Yet another type of restriction for clinical data is the ability to view personal health information (PHI) – attributes that can identify an individual uniquely, such as full name, exact date of birth, social security number, phone number, street address and so on. People who need to actually interact with the patient cannot function without such information, but statisticians and analysts do not generally have a good reason to access such data. PHI, class and attribute restrictions are examples of *attribute-level* access control. From the software-design perspective, PHI restriction can be implemented as a macro that operates on a set of attributes that have been flagged in the metadata as PHI.

There are two means of implementing such restrictions.

1. Attribute-level Control: When a user is composing an ad hoc query, classes that are off-limits are simply not presented to the user for selection, and if the user tries to search

for a class or attribute by keyword, matching classes/attributes must be filtered to those that are accessible.

This also applies to query-by-form: here, the user should not have been allowed to access an off-limit class, and for a class that is accessible, the attributes that are off limits, if any, must be hidden. If the Web page is rendered dynamically at the server end, then not rendering off-limit attributes is a more robust strategy than hiding them at the browser level: hidden attributes can still be peeked at by viewing the page's source code.

2. Row-level Control: Where conditions such as site restriction must be enforced, the SQL that is generated during ad hoc query or query-by-form must be modified by addition of a site-restriction filter to the WHERE clause. (In CSDMSs, every patient is associated with a site, and therefore the site_ID of the patient, matched to the Site_IDs associated with the user, acts as a filter.)
3. Situations like PHI are best handled through role-based restrictions as in point 1 above. Note that analysts will use ad hoc query or bulk extraction, not query by form.

Reports, which contain hard-coded SQL, are the hardest to deal with in terms of hiding individual output columns. It is very hard to compute which columns – real or simulated through EAV – are off-limits to whom, without writing an elaborate SQL parser. The best you can do is to restrict access to individual reports based on roles, a topic I'll consider shortly.

I'll now deal with the details of individual modes of data retrieval.

15.3

Query by Form (QBF)

QBF is limited to operating on a single class of data (alone or with related classes), and utilizes a form containing multiple fields, corresponding to attributes that the class may need to be searched by. The user enters textual patterns into one or more fields that indicate the desired criteria. The criteria are used to generate the filter (WHERE clause) of a SQL SELECT statement. When multiple criteria are entered, they are combined with an AND operation.

15.3.1

Patterns for Criterion Specification

Microsoft Access has a reasonably elegant implementation of QBF, and its operation is worth studying for anyone intending to create similar capabilities. In general, the following types of patterns should be supported:

- The simplest case is to enter a value of interest – e.g., NY in a State field for records relating to New York State.

The value may be preceded by a *relational operator* (<, >, <=, etc.): if none is specified, “=” is implied. The set of permissible operators is constrained by the attribute's

data type: thus, for *nominal* enumerated data types, only the operators = and $\diamond$ are permissible, since two nominal values cannot be compared for relative magnitude.

- **Wildcards and Regular Expressions:** For fields that are represent string/text columns in the database, wildcard patterns and regular expressions must be supported. (The wildcard symbol most users are familiar with is the asterisk (*), though in ANSI SQL this is replaced with the percent symbol.) Note that regular-expression capability (of which wildcard capability is a subset) must be tailored to the back-end database engine in use: no SQL dialect supports the full range of regular expressions that most programming languages can handle.

Wildcards may be combined with the relational operator $\diamond$ (this would generate the SQL clause “NOT LIKE”). Other relational operators would be ignored if a wildcard pattern is detected.

MS-Access also allows wildcard patterns for dates: thus, 1/*/* would match January dates. Generating the appropriate SQL for other databases involves using the date-part functions (day, month, year) of ODBC (discussed in section 15.4.1).

- **Ranges and Set Membership:** For numeric and date fields, range searches must be supported. SQL supports range checks using the syntax BETWEEN low-value AND high-value. For fields based on a limited set of values, the user should be allowed to specify a set. SQL allows a set to be specified thus: IN (value-1, value-2, value-3 ...).

While there is nothing wrong in making the user type in SQL syntax, alternatives that require slightly less typing are generally preferable. In TrialDB, we allow the user to specify a range thus: low-value~high-value (e.g., 3~6). For sets, we allow the user to enter a comma-separated list without the IN and the parentheses. The $\diamond$ operator may be allowed to precede either of these patterns.

- **Dealing with enumerated-type attributes:** Such attributes are also set-based, but the user normally sees only the phrases associated with each enumerated value in the interface, and does not know or care to remember the internal codes. The approach that we’ve taken, with its rationale, is described below.

On the Windows platform, list boxes on a Web page allow multiple selections by letting the user hold the Ctrl key down while making additional non-contiguous choices, or the Shift key to make contiguous choices. This is not only confusing to some users, but even experienced users may occasionally forget to hold a Ctrl or Shift key down and lose their selection. This is one reason why *a single multiple-select list box should not be used by itself for data entry*, especially if the list is lengthier than the number of rows used to present it.

The alternative that I prefer is to shown in Fig. 15.3. It uses *two* list boxes, one showing the items available, and the other the items that have been selected. The user selects one or more items at a time, based on skill level, and after clicking a “Choose” button, the list of available items shrinks while the “Chosen” list grows. A “Done” button concludes the selection process. This is easier to use because it allows easier removal of accidentally selected choices. This approach has the drawback of taking up more screen real estate, but if the screen is brought up only on demand as a pop-up window, this is not an issue.

Note that both the internal code and its associated description are displayed. When the selection is finalized, the text-box that records the user’s selection is specified as a comma-separated list of codes. Your subroutine library must have a routine that, given

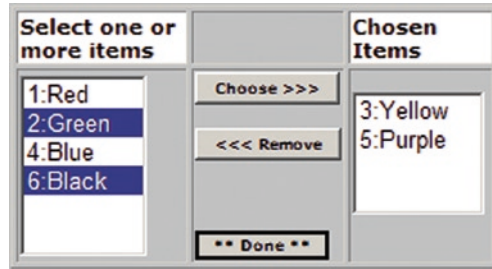


Fig. 15.3 Selecting items from lists. The list on the *left* shows what is available, while the list on the *right* shows already selected items. Both lists are multi-selectable: the current selection on the *left* may be added to the “Chosen Items” list by clicking “Choose”, and chosen items on the *right* similarly be removed (and restored to the *left* list) by multi-selection and choosing “Remove”. The “Done” button is clicked when the selection process is complete

such a list as well as the ID of the choice set, brings up the interface of Fig. 15.3, with available items on the left and chosen items on the right, so that the selection may be modified if needed.

- **Providing Online Help:** Supporting all of the patterns above makes QBF more powerful, but the list of features can be intimidating and hard to remember. It is therefore desirable to provide a standard help page (preferably containing examples) that is available with a button click, which explains the features that you have implemented, with at least one example of each feature.

15.3.2

Presenting Results to the User

We now consider the form/Web page that allows the user to specify search criteria. The question is whether to reuse the Detail View form, which is normally used to edit a record belonging to that class, or to create a special search form.

- A Detail View of a class presents all attributes of that class, except those that are hidden either because they are used for internal housekeeping, or because a given user does not have the privilege to view them. There is no harm in reusing the Detail View for a QBF interface, but in general, not all of these attributes are worth searching by: many attributes may be used rarely for searching, if at all. In the interest of reducing user interface clutter, it may be desirable to hide such attributes. There are two possible approaches here:
 - A custom search form containing only the attributes that have been identified as likely candidates for searching.
 - If the Detail View performs double duty, you can dynamically hide the non-searchable attributes. This is possible to do quite easily with Web-based forms, but if your layout is complicated, you may end up with unsightly gaps in your form.
 If the Detail View does double duty, as in MS-Access, then switching to QBF mode must clear the contents of individual fields. The validation behavior of a numeric or

date field depends on whether you are in Search mode or normal Editing mode. For example, commas and range operators would not be allowed in editing mode, but must be permitted in search mode.

One advantage of utilizing the Detail View is that it not only contains fields for the main class, but also simulated sub-forms containing fields for related many-to-one classes, which might also need to be searched by. For example, one might want to locate the details of patients who had visits between a range of dates: the class Visits is related many-to-one to Patients.

MS-Access QBF has the limitation that it does not allow search by criteria based on related tables. This is not too hard to implement in your own framework— technically, the SQL that you generate utilizes a sub-query based on the related class's link to the main class, as defined by a primary-key/foreign-key relationship. This information is available in the Relationships table of EAV/CR metadata. One point to note here is that Detail View generally presents only a few fields from the related tables – typically the most important ones – and therefore there is a limit to the sophistication of this mechanism.

15.3.3

Controlling the Number of Records Returned

If the criterion that the user specifies is not selective enough – for example, the user simply enters a wildcard symbol in a text field – the database would return most or all of the records in a class. Since QBF is only a prelude to browsing the returned records and editing one or more of them, an unrestricted query would be unacceptable in terms of database and network loads if the class contained a large number of records.

It is therefore desirable to limit the number of returned records to a number, which may either be hardcoded for all users (e.g., 500) or for specific users (e.g., analysts may be able to return more records than the average user). This means that the SQL generation really occurs in two steps. In the first step, the framework generates the WHERE clause, but the query fetches only the count of records that match the criterion. If this count exceeds the threshold, the count is presented to the user, who is prompted to make the criteria more specific. If not, the WHERE clause is reused, but this time the records of interest are fetched as well.

Even the latter step is divided into two phases. When multiple records are returned, the standard approach is to present summary information – only the most essential fields – in a list view. The SQL therefore selectively fetches these attributes only along with the primary-key-value information. When the user selects a record of interest for detailed inspection, another SQL statement now fetches all the attributes for that single record only, based on the primary-key-value expression.

15.4

Reporting

I now discuss the issues related to creating standard reports.

15.4.1

Composing SQL Using Querying Tools: ODBC-Related Issues

The TrialDB report framework utilizes certain standard reports, and must run on multiple RDBMSs. We don't want to create one body of code per RDBMS, so we use ODBC (Open Database Connectivity), a technology originally developed by Microsoft as a way to connect to multiple RDBMSs using a single dialect of SQL. Microsoft, however, made the specification freely available, and so the creators of Java, for example, built database access (JDBC – J for Java) using an approach almost identical to ODBC. I'll provide some background on the need for ODBC and the limitations of existing technology.

Due to the semi-Balkanization of SQL, different RDBMSs do the same thing differently. Thus string concatenation operator is “||” in Oracle and “+” in MS SQL Server; dates are handled differently; and each engine has a different set of utility functions. ODBC tries to hide the differences between individual RDBMSs by using a dialect of SQL that contains “escape syntax” (enclosed within curly braces) that is translated into vendor-specific SQL through intermediary software called an ODBC driver (several drivers are provided freely by Microsoft as part of Windows: commercial drivers are provided by companies such as DataDirect).

Thus, for example, date-time (“time-stamp”) values are expressed using syntax like {ts yyyy-mm-dd hh:mm:ss}, and strings are concatenated using the function {fn CONCAT (string1, string2)}; the token “fn” indicates a function. ODBC also lets you take advantage of vendor-specific RDBMS capabilities (such as stored procedures). Anything that is not escaped is passed directly to the RDBMS without any attempt at interpretation. At one time, Oracle and SQL Server used different syntaxes for outer joins, but mercifully both now use SQL-92 syntax, so one does not have to struggle with ODBC outer-join syntax any more.

There are a couple of practical problems with ODBC.

- If you use the free ODBC drivers, you may often be disappointed. ODBC Drivers can support the full set of ODBC functionality (Level 3), a basic subset (Level 1 – the free drivers), or an intermediate level 2. For example, certain functions such as EXTRACT, which extracts a *date-time-part* (year, month, hours, etc.) from a date-time-variable, are level 3 functions.
- Microsoft, while still supporting ODBC nominally, (its drivers haven't been updated in ages) has switched its emphasis to a different technology, OLE DB, which is modestly faster (because parsing of the SQL is bypassed) at the cost, of course, of your SQL being entirely non-portable. The free ODBC drivers also have subtle bugs (or “features”, because Microsoft is decent enough to document them). For example, Microsoft's Oracle ODBC driver doesn't handle the large data types (e.g., long text) correctly, causing runtime errors: Microsoft's native Oracle client doesn't have this problem.

15.4.1.1

Using VQBE for Query Generation

Until such time as you have a framework fully built, it is natural to use a GUI-based query tool to ease the task of composing queries that you can use for the SQL in your reports.

MS-Access, in particular, does a remarkable job of shielding the developer from vendor-SQL variations, by generating ODBC-compliant SQL behind the scenes, and VQBE is a significant productivity aid.

However, be warned: there are subtle differences between Access's flavor of SQL and that of your RDBMS (or ODBC SQL). Since the SQL that you see in Access is not what is actually being sent to the server, it cannot simply be copied and pasted into your reporting code: you may have to modify it somewhat. To do this requires knowledge of ODBC syntax, or at least a "cheat sheet" of differences and changes to be made. Also, you can take advantage of certain server RDBMS features only by typing SQL manually, and testing the SQL in "pass-through mode" (where Access omits translation and passes the SQL directly on to the ODBC driver). For example, Access chokes on certain modern SQL-92 constructs such as the CASE function, where a variable takes one of several values based on sequential testing of a series of conditions.

A better alternative to manual translation is to temporarily turn ODBC tracing on in your ODBC Data Sources Control Panel. A very verbose trace, including SQL statements, are written to a log file – a text file whose location and name you can specify. These statements – which actually contain the escape sequences generated by Access if needed – can then be pasted into your code.

15.4.2

Composing SQL: Dealing with a Heterogeneous Data Model

Your framework must generate SQL that can be stored for future use – or copied and modified. Using such generated SQL in combination with the query-composition tools that you already know is more productive than either one alone. The generated SQL can be used in sub (nested) queries that can be used as building blocks, and appear to the top-level query as though they were tables: this way, lower-level details are hidden.

15.4.3

Presentation/Formatting Issues

Creating sophisticated data layouts is a task at which commercial report writers excel: If your needs include free-form reports, charts, geographical maps and barcode output, this is a wheel that is too complex to reinvent. One well-known feature of high-end report writers is *banded reports*. A banded report provides details of a particular quantity such as generated revenue, by multiple levels ("bands") – e.g., by organization, by each department within the organization, and finally by clinician within each department. Each level typically has a header and footer. The header contains some identifying information about each level item – at the very least, the item's name. The footers typically contain subtotals, which are displayed whenever the current item at the previous level changes, – e.g., when a new department is encountered, the total for the previous department is output. The top-level band is that of the Report itself – the header is a label identifying the report's purpose and date/period of run, the report footer contains information such as a grand total.

A banded report essentially combines the forest and the trees. The fancier report writers implement mechanisms such as drill-down to avoid visual overload: here the details of sub-levels within the report (e.g., all clinicians within a department) can be collapsed or expanded with mouse clicks. To generate a banded report, the detail data is sorted at multiple levels through a SQL query against the database, and then aggregate functions (e.g., sum or arithmetic mean) are applied: report writers typically act as middle-tier software that performs the aggregations and formatting.

15.4.3.1

Alternatives to Banded Reports: Multiple HTML Tables

While banded reports can be powerful, this does not mean that you need them every time. Many presentation needs can be met through the use of simple HTML tables. I've found that banded reports tend to overwhelm users when there is a large amount of output to be displayed. A simpler alternative to creating a banded report is to create separate HTML tables, one for each level, based on separate queries. In the above example, we would create one HTML table for the clinicians (preceded by their organization and department), a second for the departments (preceded by their organization), a third for the departments, and finally a single-row HTML table for a grand total. Each preceding query creates a temporary database table that is used as the basis for the next query, except the first, which accesses the actual data tables: this way, the database's workload is minimized because it does not need to repeatedly access the detail data in order to summarize it.

Another tip: The process of *creating* temporary database tables starts with the most granular level first. However, when *presenting* the data, I've found it useful to present the temporary tables in reverse order – grand total first, organizations next, etc. This approach is in a sense the inverse of the banded-report approach. Busy users want summary information first: they can zero in on specific details if they care to.

Consider creating a subroutine that takes a SQL SELECT query as input, and renders an HTML table as output. This routine should also do the following:

- To make the column names more user-friendly, it should convert each column name by converting underscores to spaces and converting the resulting string to title case. If you use the standard SQL technique of using aliases for columns in your query, you do not have to perform extra manual work.
- To minimize visual clutter when presenting the lower-level HTML tables, your routine should allow the option of blanking out repeated values in a given column – so that, for example, when showing the details of all clinicians within an organization and department, the latter two appear only once. For this purpose, it should compare the value of a column's cell with the value in the previous row. If this is the same, the value is blanked out when rendered. This “*duplicate-suppression*” algorithm works in a single pass of the data table: it is easy to optimize by operating on only a certain number of leftmost columns in the table. While specified as a parameter, it can be readily determined for a multi-level report using the total number of columns in the table, and the current level.

- If you want some customize the esthetics of the output, allow the background colors for the table header, and the colors for even and odd rows to be specified as parameters – but also specify some reasonable pastel-based defaults for these. Obviously your table-header text must be visually distinguished from that of the table body.

HTML tables are convenient in that users can readily copy what they want into other programs, such as spreadsheets or word processors, without your having to program this capability explicitly.

15.4.4

A Metadata-Based Framework for Managing Reports

In a large system, the number of reports that your team creates will eventually run into the hundreds. Some of these may generate simple HTML tables, others may utilize the capabilities of a high-end report writer. In any case, you will find it necessary to create a metadata-based framework to manage them. For simple reports whose formatting needs are minimal, this framework can actually allow your team to create and run reports. In the case of reports using commercial report writers, the framework can be used to generate a front-end that explains what a particular report does, and prompts the user for parameters that are passed on when that report is invoked. It is also straightforward to record the permissions of individual users to run individual reports.

Note: SAP Business Objects Enterprise, a relatively recent product, provides much of the same functionality, by integrating with the company's Crystal Reports product. However, the cost of this package runs into several tens of thousands of dollars, and it is not too difficult to create something of your own that has three-quarters of the same functionality, if not the same scalability, and is lightweight enough to be embedded within your own software.

I'll first summarize the tables in the reporting sub-schema, and later provide the details on each of them. Figure 15.4 shows the schema diagram; column names in bold indicate the primary key for the table.

- **Meta_Reports:** This contains top-level definitions of individual reports – an ID, name, description, and so forth.
- **Meta_Report_Steps:** In general, a report consists of one or more steps: each step is defined by a SQL statement, and what it does. Some statements create temporary tables, other steps create output. If there is more than one output step, the report generates multiple HTML tables in sequence. If your framework is merely invoking an external reporting engine, the report will have no steps.

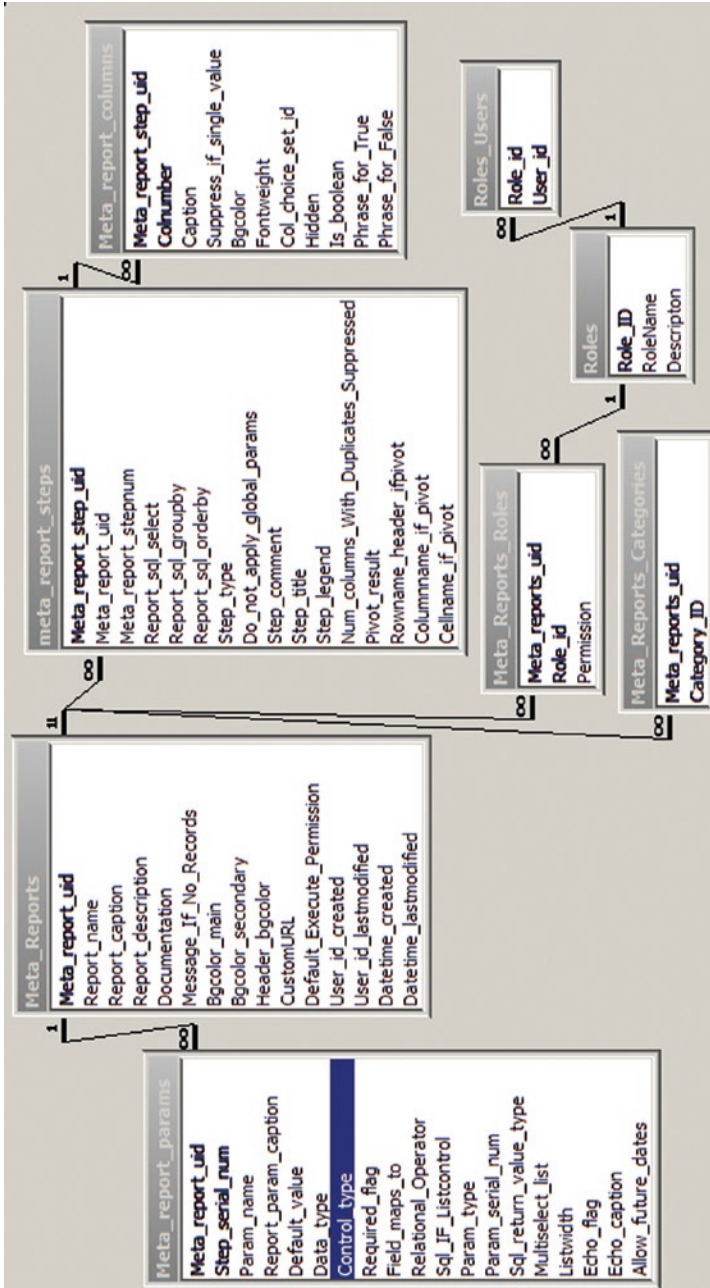


Fig. 15.4 A subschema to support the definition of tabular reports. A report outputs one or more HTML tables, and is defined by a sequence of SQL statements: some of these may create temporary tables that are used in subsequent steps

- **Meta_Report_Parameters:** To make a report specific with respect to its output, you must pass it parameters. Parameters are typically inserted into the WHERE clause of the report's SQL statement. Commercial reporting engines additionally utilize parameters in formulas and for conditional formatting.
- **Meta_Report_Columns:** This table records formatting applying to individual columns in an output step. While a given step may yield multiple columns as output, only those columns that have special formatting needs need to be specified here: many reports will in fact have zero entries in this table.
- **Meta_Report_Roles:** This table lists the permissions that individual roles have for individual reports. We use roles rather than users because roles are far fewer than users, and therefore assigning permission to a role is much faster than assigning to numerous users who have that role. Also, individual users may have more than one role, and assigning to a role is less error-prone than having to first determine what roles a given user has.

Note: Practically all high-end RDBMSs allow an administrator to create user-defined-roles – symbolic names indicating a specific purpose – and assign users to a role. Roles greatly simplify management of privileges: in a metadata-driven system, role information will be used, not just for reports, but throughout your application to tailor the interface to a given user dynamically. In theory, one could read DBMS privileges for a user and utilize them in an application; however, in most applications, the vast majority of users do not have direct access to the DBMS, which is considered a security risk; they merely run an application that has such access, and for applications like e-commerce (e.g., Amazon.com) do not know or care that a DBMS exists. This application, of course, must authenticate the user and determine what that user can do. You must therefore perform role management yourself, which is not particularly complicated if you create your own **Roles** and **Roles_Users** tables. The Roles table has the self-explanatory columns Role_ID, Role_Name and Description, and the Roles_Users table the columns Role_ID and User_ID: the last is a foreign key into a Users table. Neither of these two tables will be described further.

- **Meta_Reports_Categories:** This table is used to categorize an individual report into one or more categories that facilitate searching for a report of interest; the categories are terms that are meaningful to the users. We have not illustrated a Categories table, which simply contains a Category_ID and Category_Name columns.

The description of the columns of individual tables now follows.

15.4.5

Meta_Reports

Meta_report_uid: The surrogate, auto-generated primary key.

Report_name, Report_caption, Report_description, Documentation: These columns record respectively an internal name, a brief caption for the user, a 1–2-sentence explanatory description of the report's purpose, and a more extensive description. The caption and brief description are presented when a list of available reports is presented to the user.

Message_If_No_Records: The message to be displayed if no records are returned by the report. If not specified, you should utilize some standard default message, such as “The report did not return any records. Please check the criteria that you specified and try running it again with different criteria.”

Bgcolor_main: Bgcolor_secondary, Header_bgcolor: The background colors for odd and even rows of tables and the header row. While we use default pastels for these – DEDFC8 (beige), E1E9EA (a light powder blue) and F4F0CA (buff), we have found that users occasionally want to modify these. Originally, we also allowed selection of fonts and font sizes, but we found that the default (8 point Verdana) was hardly ever changed.

CustomURL: If this is not blank, the values of parameters are gathered from the user, and used to invoke the URL. The URL may be a custom program that you have written, or a generic dispatcher of your own creation that invokes a commercial reporting engine. The URL should allow parameters to be passed through the standard GET mechanism as name/value pairs that follow the name of the Web page. I discuss parameters shortly.

A useful tutorial on passing parameters to a report designed with the widely used Crystal Reports is Chatterjee.¹

Default_Execute_Permission: Boolean If True, permission to run the report is granted for all roles except those specified in Meta_Reports_Roles. If False, permission to run is denied except for the specified roles. Whether you set the default execute permission to true or false depends on the nature of the report: reports needed only by a few users should have this set to False, while for those needed by most users, True is more appropriate. The reason for this choice is to minimize administrative overhead in terms of having to set individual role permissions manually.

User_id_created, User_id_lastmodified, Datetime_created, Datetime_lastmodified: These should be self-explanatory.

15.4.6

Meta_report_steps

Meta_report_step_uid: A surrogate primary key.

Meta_report_uid: Foreign key into Meta_Reports

Meta_report_stepnum: An integer used to order the steps and determine the order of execution. Note that successive steps need not be consecutively numbered. Gaps in the sequence may exist (and are in fact convenient if you wish to reorder steps).

Report_sql_select: The SQL statement that defines a step. If you are passing parameters to the report, this statement should not contain GROUP BY or ORDER BY clauses; these are stored in the columns. **Report_sql_group By** and **Report_sql_orderby** respectively. The reason for this is that parameters are used to extend the statement’s WHERE clause, which must come before the grouping and ordering.

Step_Type: The step type can be one of the following:

- **Output:** A SELECT statement that is destined to produce an HTML table.
- **Temporary Table:** A SELECT statement that creates a temporary table, which will be used in a subsequent step. We don’t hardcode the temporary table name for two reasons.

First, the name of the temporary table is system generated, based on the current user and the step number – so that if multiple users run the same report at the same time with different parameters, there are no conflicts. Second, individual DBMSs use different syntax for this purpose: SQL Server uses `Select ....INTO tablename`, while Oracle uses `CREATE tablename as SELECT .....`, so to be database-neutral we generate vendor-specific syntax at the last moment, utilizing an environment variable that specifies the DBMS. This way, we avoid hardcoding vendor-specific syntax into the framework.

- **Execution:** A SQL statement other than the two types above. This is used rarely, but sometimes one needs to perform certain operations – e.g., a bulk update of the contents of a temporary table – before output is generated.

Step_title, Step_legend, Step_Documentation: The title and legend (an explanation of the step’s output) are used for Output steps. They annotate the table that is produced. Documentation is used internally for added explanation.

Num_columns_With_Duplicates_Suppressed: Applies to output steps only. See the duplicate-suppression algorithm in section 15.4.3.1 the number of leftmost columns where duplicate suppression is to be applied are specified here. In theory, duplicate-suppression could be applied to every column, but this may involve unnecessary processing, as well as cause confusion for detail columns.

15.4.6.1

Columns Used for Pivoting Operations

To explain the purpose of these columns, I’ll introduce the concept of *crosstab (pivot) queries*. A crosstab query returns *three* fields. The results are converted into a layout with fewer rows, but more columns, than the original output. One of the fields in the original output is used for the row labels (the first column in the crosstab), the second for the column labels (the 2nd–Nth columns), and the third serves as the basis of the cell contents. In general, the choice of row-label and column-label fields is arbitrary, but typically chosen so that the columns are expected to be fewer than the rows. Both the number of rows and columns depend on the data, and are not predictable.

The first implementation of crosstab queries in SQL – adding the PIVOT keyword to SQL – was, surprisingly, in MS-Access: this implementation is still by far the cleanest. MS SQL Server and Oracle followed suit much later, but these implementations are essentially crippled by requiring the number, and names, of columns to be specified in advance – i.e. hard-coded: if the data changes, the query needs to be rewritten. See the SQL Server 2008 documentation at <http://msdn.microsoft.com/en-us/library/ms177410.aspx> for an example where the IDs of individual salespersons are hard-coded in a pivot query against Microsoft’s sample “AdventureWorks” database that computes the number of purchase orders by vendor for each of five employees.

All the DBMSs have the limitation that they require the PIVOT statement to apply an aggregate function on the “cell” field –e.g., the COUNT function in the AdventureWorks example above. However, there are many circumstances where only the raw values of the cell field must be presented. An example is in the area of longitudinal clinical studies,

where one wants to observe the time-course of the clinical parameter that reflects the condition being treated (e.g., for patients with depression, one might monitor an aggregate depression rating). The raw data fetched by a database query has the structure <patient_ID, time_point, value>, but to facilitate inspection, one would like to lay it out so that the time-points (arranged chronologically) form the column labels, the Patient IDs form the row labels, and the rating values the cell contents.

The algorithm for creating a crosstab is fairly straightforward, if memory-intensive. One first determines the unique row-label values and column-label values through database queries, which also typically sort these values in some desired order. One thus has two ordered arrays, and from this one can create two hash tables, where the key is a label-value and the value is the position in the ordered array. Now one allocates a two-dimensional array, where the X dimension corresponds to the columns (e.g., time-points) and the Y dimension to the rows (e.g., patients). Now one scans the detail data, which has the form <row-label, column-label, value>: detail data need not be sorted. As each row of detail data is encountered, one transforms the row and column labels into X-Y coordinates by consulting the two hash tables, and places the value at the computed position in the 2-D array. After the detail data is completely scanned, the array is printed out, along with the corresponding row and column labels.

The same algorithm can also be used to pivot EAV-modeled data into a flattened structure, where the column headers are the names of attributes of a given class, and the rows correspond to the individual objects that are class members.

The columns that are used to specify pivoting operations are described below:

Pivot_result: Boolean, true if the data fetched by this step is to be pivoted.

Rowname_header_ifpivot, Columnname_if_pivot, Cellname_if_pivot: The columns in the query that are used for the row labels, column labels and cells (values).

15.4.7

Meta_report_columns

Meta_report_step_uid: Foreign key into Meta_Report_Steps

ColumnName: The name of the column to which the formatting applies. We could have used column number, but this is more volatile because users often decide that they want the output to be modified; column names that correspond to columns in the output SQL do not change so often.

Caption: The column label. Generally, this can be left empty, because SQL's aliasing mechanism, and the use of a transformation function, meets practically all needs.

Suppress_if_single_value: Boolean. This applies when you run a report with optional parameters. For example, you may want details for a single state or for all states if the state is not specified. The output table in any case will contain a State column. However, for a single state this column is non-informative, because all cells will have the same value, and merely occupies space. In such a case, you can hide it if you wish, by setting this field's value to True.

Bgcolor, Fontweight: – If not null, one or both of these can be used to visually emphasize a particular column.

Col_choice_set_id: (Please refer back to Chapter 2 if necessary.) Sometimes, an output column is based on an enumerated data type, and the internal codes that would be output by default would not be comprehensible to the user. To output the user-friendly descriptive phrase corresponding to the codes instead, you must incorporate the Choice_Values table in the query, filtering on the ID of the Choice Set for that data type. When multiple columns are based on enumerations, this requires using the Choice_Values multiple times with separate aliases. An alternative that we've found useful is to store the Choice Set ID for a column here: the reporting framework code loads the descriptive phrases into a small hash table, so that conversion of a code to its corresponding phrase is quite efficient.

Hidden: If true, the column is unconditionally “hidden” – i.e., not used to construct the HTML table. You may need this feature when you create a temporary table that you use further, and certain columns that were used in computations do not need to be displayed in the final output.

Is_boolean, Phrase_for_True, Phrase_For_False: Most high-end DBMSs do not implement the Boolean data type directly: instead, you have to create a short integer column (where 0=False) and document its Boolean nature. If you set this value to True, then zero and non-zero values are displayed respectively as Phrase_for_False (default No) and Phrase_for_True (default Yes).

15.4.8

Meta_report_params

Meta_report_uid: Foreign key into Meta_Reports.

Param_serial_num: An integer used to order the parameters and determine the order of presentation. Note that successive parameters need not be consecutively numbered.

Param_type: One of either Global (applies to all steps) or Step (applies to one or more steps, but not all).

Step_serial_num: The steps that a parameter applies to, if type Step. This field is a string that contains either a single number, or a comma-separated list of numbers. The numbers should correspond to the numbers of the steps in Meta_Report_Steps for the given report.

Param_name, Report_param_caption: Internal name and caption presented to the user.

Data_type: A single-letter code that represents the parameter's data type.

Default_value: Default value (used to initialize the parameter when presented).

Control_type: One of Text box, Check Box (for Booleans) or List box (for enumerated data types). Default is text box.

Required_flag: If true, the parameter must be specified. If false (default) it is optional.

Field_maps_to: Applies only for Step parameters. What column name does it map to in the step/s where it is used?

Relational_Operator: The operator that should be applied to the parameter value. Default is equals (=); however, sometimes a set membership operator (IN), or an operator such as >= or <= can be used. The last two are useful for date ranges.

Sql_If_Listcontrol: The SQL that populates the list box, if the parameter is presented as a Listbox control.

Multiselect_list: If the parameter is presented as a list box, can multiple choices be selected? (If this is chosen, the operator automatically becomes IN.)

Sql_return_value_type: One of either Single Value or List (a comma-separated list of values).

Allow_future_dates: For parameters of the date type, whether dates in the future are allowed (Default is True).

Note: If the parameter is used purely for passing to an external URL (e.g., to a report created by a reporting engine), the parameter should be set to be of type Global: in the designer interface, all non-applicable fields should then be disabled.

15.4.9

Meta_Reports_Roles

Meta_reports_uid, Role_id: Foreign keys into the Meta_Reports and Roles tables.

Permission: One of either Granted (True) or Denied (False). In the user interface, based on the setting of **Default_Execute_Permission** for a given report. Only one of these choices will be permitted when you add rows to this table for that report.

I should note here that Business Objects Enterprise (BOE) uses several permissions, each of which can be set to Granted or Denied: Run, Schedule, Design, and so on. Unless you have a large team of power users capable of designing and testing reports, you might find provision of extended permissions overkill: the Schedule permission, however, may be useful. We consider the issue of providing report-scheduling capability shortly.

15.4.10

Requirements of the Reporting Framework Runtime

Reports must be capable of being run both interactively as well as in batch (scheduled) mode.

- Interactive mode typically involves the application presenting a Reports Screen, where a list of reports, along with the nutshell description, is presented in a list. The user may also be presented a list of categories, where selecting a category shows all reports in that category: this latter approach is useful when the reports are numerous. When the user selects a report, any parameters that the user requires are prompted for, and then the report is run.
- The reports that are available to the current user in either mode depend on the user's role/s.

Running a report in batch mode involves several issues:

- The most basic capability is a Web service that can run a report (specified by report ID) that takes the individual parameter/value pairs that it needs. When the report has finished running, it must send a URL to the user by E-mail or a similar mechanism: this URL allows the report output to be downloaded.

Most operating systems provide a programmable scheduler that can be controlled by your application. For example, the current version (2.0) of the Windows Task Scheduler is full-featured, and several open-source code libraries make its rich programming model much more accessible, so that you can control its behavior through procedural code that can, in turn, sense conditions in your database.

It is desirable to provide at least the following batch capabilities: the ability to run non-interactively on a specific date/time or at a repeating time interval (e.g., daily, weekly, monthly) or repeating date (e.g., fixed day of month). Windows Task Scheduler will allow you to create E-mail tasks: it will even let you run tasks in response to “events” on your machine – such as logging on, insertion of a USB device, creation or modification of a file, and hardware error conditions. Reports based on events may be overkill, but reporting engines such as Crystal do allow the possibility of generating selective reports based on conditions such as failure of individual salespersons to meet their quotas.

- If you want to enable the Web service to be available outside the organization’s Intranet, you must build in an authentication mechanism. There are several articles describing how implement Web service security. A fairly simple mechanism is described on the Amazon Web Services site, “Authenticating REST Requests”, <http://docs.amazonweb-services.com/AmazonCloudFront/2010-08-01/DeveloperGuide/index.html?RESTAuthentication.html>. This involves an application that utilizes a secret key that is stored securely by the client: a copy of it is stored on the server against the user’s ID. The client application first makes a call to the server to get the current date/time as recorded on the Server (accurate to the nearest second). This is then hashed with the SHA1 algorithm using the secret key, and sent along with the (public) user ID, the date/time value obtained previously, and the request proper. The server, using this ID, fetches its own copy of the secret key and tries to compute the same hash using the date-time stamp. If the computed hash is identical to the hash in the message, the user is authenticated. For added security, if the difference between the current time and the time-stamp in the message is greater than a certain interval (Amazon uses 15 min, but you can be less generous), the message is rejected.
- When a report runs, it must report success or failure. The report may be one step in a batch process: failure must be conveyed to the top-level process so that appropriate action may be taken.

15.5

Ad Hoc Query

I’ve emphasized earlier that a framework that assists in query composition is necessary for productivity. I’ll first discuss the requirements of such a framework, and the constraints inherent to the problem, with a brief introduction to temporal queries. I’ll then discuss the

process of query from the user perspective, and the corresponding behavior of the system (and the information it needs to store). Finally, I'll discuss how queries can be stored for reuse and modification, with a description of the schema that we use.

15.5.1

The Sparse-Data Problem

For traditionally modeled data, a single query of multiple tables will yield a single output table – even though values in certain output columns may be repeated. If, however, you are dealing with sparse-data, as in EMRs and CSDMSs, the situation is different: *a single query can potentially produce multiple output data streams.*

The reason for this is as follows: in a query that seeks to fetch multiple attributes/parameters, the number of data points returned will vary for each parameter. This is not due to missing data, but due to the way the data-capture process was designed. For example, in clinical situations, the data points are likely to be much more numerous for routine examinations and investigations than for expensive investigations or invasive ones that involve a significant degree of patient risk.

Moreover, for clinical data, one cannot intuitively join “join” the serial results of a blood test for a given patient with those of repeated MRI scans when both were performed on different dates. The best one can do is to create a *time axis* for the patient based on the union of all time-points associated with each parameter, and create a two-dimensional matrix – time-points as rows, parameters as columns – where available values of a given parameter are entered into the cells. Many cells in the matrix will be empty.

If you want to do this entirely using SQL and combine *N* attributes, you have to perform *N-1* pair-wise *full outer joins*: with each joining, missing values of attributes from the two tables on *either* side of the join are recorded as empty (null) values. Full outer joins are significantly less efficient on one-sided (left/right) outer joins, which are less efficient than inner joins.²

The full outer join is necessary even in EAV/CR designs. Here, your attributes are not sparse: however, when using an EAV design to simulate a regular table, certain attributes may be permitted to be empty (null); so assembling the physical-table equivalent requires full outer joins for such attributes to ensure that all objects' data is gathered.

Under some circumstances – bulk data extraction, which I'll consider shortly – you can use the pivoting algorithm of section 15.4.6.1 to perform the equivalent. Here, the row-labels hash table is the set of unique time points, and the attribute IDs form the column-labels hash table. Obviously, when presenting the data, you will use attribute descriptions/captions instead of IDs. The matrix used for the cells is filled up one attribute at a time.

You can also use inner joins instead of full outer joins when certain attributes are the focus of your query, and you either know that every output row should contain an instance of those attributes, or you don't care for output rows where these attributes' values are empty. You could allow the user to specify such attributes by specifying them as *required/mandatory*, as opposed to *optional*.

In EAV/CR designs, if certain attributes have been specified as being mandatory (i.e., not null) in the metadata, then there is a guarantee, as long as there are no bugs in your

user-interface code that for every object there will be an instance of such attributes. In this case, you should take advantage of the “mandatory” constraint and utilize inner joins to fetch such attributes’ data.

The potential creation of multiple output streams has an important bearing on the design of your data warehouse. As I’ve stated earlier, the vast majority of users do not perform ad hoc query, but the ones that do will end up creating numerous temporary tables – which they will preferably want to give meaningful names, since they may want to get back to these later – for example, to export their contents to analytical packages. To keep the data warehouse design clean, *you will have to give each user their own schema* to store these tables. “Schemas” are a means of creating separate namespaces: two tables or views with the same name can co-exist in the same physical database as long as they are placed in separate schemas. You would probably have to do this anyway, but with ad-hoc query of sparse data, it becomes particularly imperative.

15.5.1.1

Taking Advantage of Data Regularity

When you are dealing with multiple patients rather than just one, the output problem gets even more challenging. Because different patients have parameters that occurred or were recorded on different date-times, absolute date-time values are minimally useful: *relative time intervals* (starting from a time-zero, defined in terms of the problem you are addressing) are far more important, but even these can vary greatly in the typical clinical-care situation, complicating the analytical task.

In clinical studies, where parameters are recorded on a fixed schedule, the analytical and display problem can be simplified greatly by utilizing the study calendar information to align the data points. More important, if you know *a priori* that certain parameters always go together (for example, because they are both part of a standard laboratory test panel), then your software should take advantage of this information to simplify the process of creating output.

You can also take advantage of an additional level of clustering: you may find that, in the single-study-query scenario, certain case report forms (CRFs) are always collected together, and therefore the attributes within those CRFs co-occur. A common reason for this is that a single *conceptual* CRF that is very lengthy may be divided into smaller *physical* units, so as to minimize the need to scroll extensively when entering or browsing data, and to minimize risks of data loss in case of situations such as time-outs.

Note for Developers on Time-Outs: Most Web applications do not continually communicate with a remote Web server each time a single field on a page is edited: on a wide-area network, this is the equivalent of a 747 ferrying passengers across the Pacific one at a time, and would reduce responsiveness. Instead, updates are batched. The user saves the data by clicking a button, or is prompted to save just before actions such as closing a CRF or moving to another CRF. For security rea-

sons, if a machine is inactive for a certain period of time, it could mean that the machine is unattended – a common situation for busy caregivers who are often interrupted with emergencies – and so the session should be terminated to prevent unauthorized access. Unfortunately, if the user of the machine had not saved any edits made to the form, these will be lost. The default session time-out for ASP.NET is set by Microsoft at 20 min, but experience with frustrated clinical users has caused us to increase this to between 45 min to an hour, based on what the users at an installation tell us about their work environment.

Related to the scrolling issue, it makes sense to pamper your caregivers – for that matter, anyone earning a high hourly wage – with decent-sized monitors: dual or even triple-monitor cards are now readily available, and three such monitors cost much less than one giant monitor with the equivalent surface area.

When you take advantage of data regularity, you may not entirely eliminate the multiple-output-table problem: however, even if you can greatly reduce the number of tables, you have simplified future analysis greatly. One important source of regularity in your data are the mandatory attributes, as well as certain attributes (such as most demographics variables) that have only a single instance per patient and lack associated time-stamps. Aggregates, by definition, will have only one value per patient. Such attributes can and should be combined.

15.5.2

Temporal Queries

If you are dealing with clinical data, your tool will have to facilitate the ability to answer queries of a temporal nature. In clinical studies, you want to determine the time-course of one or more clinical parameters with treatment; in epidemiological investigations, you want to determine the temporal interval between a phenomenon and its suspected cause. Even here, as we shall see, it pays to take advantage of your problem's specific constraints. I've already mentioned in Chapter 12 that use of the relative-time-interval information in the Study Calendar can greatly simplify aggregate analysis, even at the cost of slightly fudging the interpretation of data for individual patients by ignoring the modest inter-patient time-interval variations.

Temporal databases and queries were at one time a hot topic: A group of researchers even came up with a proposal called TSQL2,³ to add temporal capabilities to existing RDBMS engines. The proposal seems to have gotten nowhere, mainly because it involves a complete overhaul of existing RDBMS table structure, and significant changes to SQL itself: most developers who need temporal capabilities in their applications therefore implement these from first principles.

I've stated earlier that clinical data are associated with either one or two time-stamps: the latter kinds of data, which represent the duration, are less common in raw data. However, when you perform certain aggregate operation (specifically count, sum, average and standard deviation) on clinical data with a single time-stamp, *the result gets two time-stamps*, representing the period over which aggregation was performed. On the other hand, the operations max and min should be associated with a single time-stamp representing the time-stamp corresponding to this value. For data where more than one value happens to be the maximum or minimum, a single timestamp value is problematic, but choosing the first timestamp is normally adequate.

When comparing two *instants* (events with a single time-point representing when the event happened), the temporal operations that can be defined are less than (**before**), greater than (**after**) and equals (the same). In reality, these operations are much more useful if they are qualified by a time-unit and a number. E.g., we would like to know if two time-points are more than 39 days apart. Another consideration is that a computer's clock typically records time at 1/100th of a second, but for certain phenomena, we would consider two time-points identical for practical purposes if they fell within a certain interval (e.g., 1 day, or even 1 week).

When comparing two *periods* (events with a "begin" and "end" time-point) – let's call them A and B – many more operations can be specified. These operations were originally defined by Allen, but their names are not intuitive enough to remember without a simple diagram or by using the begin() and end() operators on a single period. The operators and their definitions are provided below, the former in **bold**:

A **before** B = (end (A) < begin(B))

A **meets** B = (end (A) = begin(B))

A **equals** B = (end(A) = end(B) and begin(A) = begin(B))

A **during** B = (end (A) < end(B) and begin(A) > begin(B))

A **overlaps** B = (end (A) > begin(B) and begin(A) < end(B)) or vice versa

A **starts** B = (begin (A) = begin (B) and end(A) < end(B))

A **finishes** B = (end (A) = end (B) and begin (A) > begin (B))

I've stated these only for the sake of completion. In all honesty, only the **overlaps** operator provides a useful short-hand: the other operators are possibly easier for the user to specify using their definitions. Further, given my earlier point that in practice the user almost always needs to specify the "float" (the time interval within which two time-points may be considered identical), as a software developer you are probably better off if your GUI ignores the Allen operators entirely and mandates operations based on time-point extremities only. Remember that for parameters of the instant type, begin() and end() have no effect, simply returning the time-point itself.

Finally, you can't do everything you'd like to do with time-oriented data using SQL even if you implemented the entire TSQL2 specification. If you wanted to look at cyclical patterns, for example, or determine gradients (rates of change with time), you're better off exporting your temporary output tables to a time-series-analysis package.

15.5.3

Selecting Attributes of Interest

The first step in query is selecting attributes of interest. There are two approaches that you should enable. In a *drill-down approach*, your attributes have been grouped in some logical fashion, and you allow the user to take advantage of those groupings.

- In the EAV-CR scenario, attributes are grouped by the class they belong to.
- In a clinical-study data management system (CSDMS), attributes are segregated into higher-level groups and CRFs. For the common circumstance where users want to query data from a single study, it makes sense to enable drill-down starting with the relatively modest number of CRFs in a single study.
- Another natural grouping in EMRs occurs for non-sparse attributes, such as those related to patient demographics, that are modeled as physical tables.

In any case, when supporting drill-down, you must blur the distinction between artificial, metadata-based groupings (e.g., CRFs) and natural groupings based on physical tables. From the user's perspective, the groupings should not be treated differently.

The other means of searching for an attribute is by key-phrase or partial string search. This is useful when the CRFs and/or attributes are numerous, especially in multi-part CRFs – very often, it is hard to remember in what CRF a given attribute lies. This mode of search is also important when groupings are not useful – for example, when querying data from a *set of studies* that use a common data dictionary but where the design of individual CRFs varies widely. In the latter situation, it is quite possible that an attribute lies in different CRFs in different studies (unless entire CRFs are being reused) and so drill-down is less useful.

From the query interface, *it is important to provide access to the detailed metadata*, such as long descriptions, data types, etc., that are associated with attributes and their groupings. We've managed clinical studies that have more than 2,000 attributes based on questionnaires and physical examination, and sometimes the brief captions that are presented in the forms are simply inadequate in terms of helping an analyst-user who does not work with the forms on a daily basis to identify what is really needed. The process of exploring the metadata to compose queries, in fact, brings to light any inadequacies in the documentation associated with your metadata elements: poor or non-existent documentation has your analysts scratching their heads in puzzlement.

In queries of clinical data, the Patient ID is almost always needed, since all clinical data is associated with patients. In such cases, the query algorithm generally works in principle by filtering patient IDs according to attribute-based criteria, and then fetching certain attributes for those patients. In such cases, it may be worth either providing a shortcut to select Patient ID, or automatically incorporating Patient ID in the query: the latter is our preferred approach.

15.5.3.1

Getting Metadata Associated with an Individual Attribute

Once the user selects an attribute, the software must pull its associated metadata from the database and cache it. The metadata serves three purposes: some data are used for presentation, others to assist validation and ergonomics, and the remainder for generation of SQL.

- The *ID*, *name*, *description*, *data type* and *Choice Set ID* of the attribute. The last is zero for non-enumerated data types. If non-zero, this is used to populate pull-downs in the GUI to allow specification of values using descriptive phrases, as described earlier in section 15.3.1.

- The attribute storage type: (one of: *EAV* or *Physical table*), and the *physical name of the table or view* where the attribute's data lies. Since query involves only selection, views and tables are indistinguishable for query purposes.
- For time-stamped data (which is ubiquitous in clinical settings) we get the *event type*, so that we know whether a data point has zero, one or two associated time stamps. To recapitulate, the respective event types are None, Instant or Period. The Event type is important because, as I have discussed earlier, the nature of the temporal operations depends on the nature of the event: obviously, temporal operations cannot be allowed on attributes where the event type is "None".

We also fetch the *field names* that record the start of event and end of event. In the earlier subsection on temporal operations, the operations *begin()* and *end()* for an attribute, when implemented in SQL, simply access these two field names respectively. (For instant events, the End of Event field is empty. For date values stored in EAV or physical-table form – examples of such values are the Date of Last Menstrual Period – both the *begin()* and *end()* operations would fetch the value itself.)

- For attributes modeled in EAV fashion, the internal (physical) names of the *attribute* and *value fields* in the table or view for the attribute's data. In a production EMR, it is not reasonable to assume that all EAV data is modeled using the general-purpose EAV tables of Fig. 5.5 in Chapter 5, where the Attribute field is "Parameter_ID" and the Value field is "Value". For example, most production systems model laboratory values in a specially designated EAV table, where these columns may be labeled Lab_test_ID and Result. It is for the same reason – *making our query generation sufficiently general* – that the Start of Event/End of Event fields above are also stored in and fetched from the metadata rather than hard-coded.
- Information specifying the *higher-order grouping* of the attribute within forms used in the study – these depend on your particular system, but would include Class ID for EAV/CR and Form_ID as well as Group_ID for CSDMS schemas. In addition, for CSDMSs, I've already stated earlier that it is possible to compute, from the Study Calendar, which attributes go together within a single study.
- If the user has indicated the intention to query data from a single study, each attribute can be assigned a pre-calculated integer (which we call the *co-location number* in TrialDB) which is shared by all co-occurring attributes. More on this in Sect. 15.5.6.2.

15.5.4

Specifying Selection Criteria and Output Options for Attributes

Once the user has selected an attribute, s/he can specify criteria based on that attribute. These are:

- Whether the attribute appears in output – i.e., whether it should appear following the SELECT keyword in the SQL statement. Some attributes are used only for specifying criteria, but the user may not care to view their actual values.
- Whether the attribute is *Optional*. If non-optional (i.e., required) this column *cannot be null for any retrieved row* in the output data. From the implementation perspective, *all required attributes can be fetched together using inner joins rather than full outer joins*.

- Relational operator and value pattern. See the earlier section on Query By Form: the default relational operator is “=” (equal to). The set of patterns we have described for Query-by-Form must be supported here.
- *Date/time or Event restrictions*: The user may wish to restrict the output for a time-stamped attribute based on a range of absolute start and end dates or, in the CSDMS scenario, a range of Events. I’ve already stated earlier than absolute dates are less useful in most circumstances because patients are seen at different times, but they may be important in certain emergencies such as infectious disease outbreaks. The user interface should provide a pair of fields for each option: the user can specify either Event or Absolute date ranges, but not both at the same time.

An *aggregate function* (optional). The aggregate functions that should be supported are MIN, MAX, COUNT, AVG, STD (standard deviation), EARLIEST, and LATEST. The user should either use the aggregate function in a criterion, as in SQL’s HAVING clause of the SELECT statement, or display the aggregate value, or both.

EARLIEST or LATEST are not part of standard SQL but are important for time-stamped attributes. The implementation is straightforward: getting a list of attribute values – optionally filtered by date/event range – conceptually sorted by time-stamp in ascending or descending order for EARLIEST or LATEST respectively, and retrieving the first value. This operation can be achieved in a single SQL statement if you use a sub-query.

As in the case of relational operators, the list of available aggregate functions is constrained by the data type of the current criterion’s attribute (e.g., AVG and STD are available only for integer or real attributes). If no aggregate operator is supplied, a patient will be selected if even a single value for that patient matches the criterion. Aggregate operators other than COUNT are not permissible for string, Boolean, or enumerated data types that reflect nominal (categorical) values. For ordinal (ranked) attributes, strictly speaking, AVG and STD are not meaningful: the median and range (max – min) are more useful. There are several techniques for computing the median using SQL, e.g., Itzik Ben-Gan’s recipe in SQL Server Magazine⁴: these rely on new analytical functions introduced in SQL-99.

- Inclusion of Child Attributes: I’ve alluded to the problem of hierarchically related attributes in section 11.4 of Chapter 11. Certain clinical schemas may contain multiple, closely related attributes, typically variants of a lab test performed in different laboratories, with slightly different variations of normal. When querying the data, the user may simply want to specify a “generic” form of the test – such as a serum Potassium – without caring which lab they came from, and have the system fetch the values of all attributes that are children (descendants) of this attribute.

A related issue is that the query may be defined in terms of the *normal range* for the lab test. For example, the user may wish to look at all values that are above normal. For many clinical parameters, the normal range is based on factors such as age, sex and physiological states such as pregnancy: as stated earlier, there is also a slight variation based on the laboratory where the test was performed and the test method used.

Most laboratory-based reporting systems implement the identification of normal ranges using a metadata table containing the following columns: Lab Test ID, Gender (one of: Male, Female, Either), Lower and Upper Range of Age (in years), Lower Limit of Normal, Upper Limit of Normal. (Physiological states like pregnancy and exercise

are not handled.) Given a patient whose gender and age are known, it is straightforward to determine the normal range using a simple table-lookup: this can be achieved using SQL embedded in a stored procedure: consequently one can determine whether a given value for a given patient is within normal limits, above normal or below normal.

A query criterion such as “Find all patients whose Blood Hemoglobin is greater than normal” can be expensive and tricky to optimize. If queries based on normal ranges turn out to be an important requirement for your users, the simplest solution is to save query time by utilizing extra space: in this case, two additional columns in a laboratory results table (a special-purpose row-modeled table) that record the upper and lower limit of normal *for that patient*, computed at the time the test was performed.

- Saved Table Name, Column Alias: These columns are used in the sparse-data situation, where an individual attribute is potentially destined for its own output table. The column alias is just a more meaningful moniker for the column than its `Attribute_Name` (which is used by default). Note that, if this particular attribute can be aggregated with other attributes in the output, the Saved Tablename may not be used. Instead, the Default Table Name in the Query Definition, or the Saved Table name of the attribute with the smallest serial number that is grouped with this attribute will be used instead.
- Output Time Stamps: I’ve stated earlier that when values of an attribute are generated, the user often needs to see the associated time-stamp/s. These are additional columns in the output: the simplest way to create them is generate names such as `Column_Alias_Start` and `Column_Alias_End`: obviously, attributes with event type Instant only have a Start time-stamp.

In CSDMS situations, you should let the user choose, through a global setting, whether actual time-stamps or event-based time-stamps are output: the latter are more useful. Based on your application, it may also be desirable to generate relative time intervals (by subtracting each time-stamp from the earliest time-stamp) instead of time-stamps themselves. The user should have the option of expressing the time-interval in a scale appropriate to the study – e.g., days, weeks, months, etc.

15.5.5

Saving Queries for Reuse: The Schema Volatility Problem

Any query tool worth the name needs to provide a means for storing and reusing query definitions. Let’s now consider strategies for achieving this for heterogeneous data.

15.5.5.1

Storing the Generated SQL Code: Necessary but not Sufficient

Your query tool will obviously generate SQL code as its end-point. However, the component of this code that accesses attributes modeled using EAV will contain the IDs of attributes and higher-level attribute groupings such as classes, or CRFs in the clinical scenario. To complicate the matter, recall that in the sparse-data scenario, a single conceptual query that selects multiple attributes can result in multiple output tables. In other words, the generated query consists of a *sequence* of SQL statements.

While the user obviously selects attributes by name in the user interface, there is no reason to store such names in the SQL itself. In fact, incorporating the metadata tables that contain the attribute and grouping names in the SQL just to make the code clearer would needlessly increase the number of tables in the join, slowing query performance.

In brief, the omission of meaningful attribute and class names and descriptions from the SQL make it close to incomprehensible. To facilitate understanding, the code would have to be documented separately in detail. It would be nice if the information that is already in the user interface when the query is first composed is saved along with the SQL; this reduces the documentation task significantly.

15.5.5.2

Loading a Stored Query into the User Interface

Very often, a user wants to take an existing query and modify it slightly for reuse in a different circumstance. Obviously it is desirable to enable change through the original query-composition user interface rather than by forcing the user to tinker with the SQL. Therefore the query-storage representation must allow it to be loaded readily, with a modest amount of coding. In other words, there must be more or less a one-to-one correspondence between the elements visible in the user interface and the stored representation.

The simplest way to model the information that is entered in the interface is through a set of relational tables. Multiple tables become necessary because there is a many-to-one relationship between the criteria applied to individual attributes (which are indefinite in number) and the top-level query definition itself. However, because all the relationships are *strictly* many-to-one, using XML as an alternative is appropriate. In fact, the XML representation has the advantage that it is almost human-readable and makes the query as a whole more readily comprehensible than information scattered across multiple tables.

The more important reason for using an alternative besides the generated SQL code alone is that often the representation of classes may change over time, where classes initially modeled using EAV are redesigned as physical tables (and vice versa). This is especially true of the EAV/CR scenario but may occasionally happen for clinical data as well. If this happens, queries that access those classes will fail to work. A high-level, yet computable, query representation allows the SQL code to be regenerated readily.

The high-level specification is analogous to a program written in a high-level language, while the emitted SQL is analogous to machine language generated by a compiler. Compared to alternative approaches (e.g., Johnson and Chatziantoniou⁵) XML is lightweight and readily parsed.

I should mention that my friend and colleague Ani Deshpande, who developed the first Web version of the TrialDB AHQ interface, which we described in,⁶ was a strong advocate for an XML representation at a time when the technology was still new, and I was still skeptical. Now that XML support in RDBMSs has matured, I appreciate his foresight.

In describing the storage model, I'll still use the relational model, because it is somewhat simpler to present as an entity-relationship (E-R) diagram.; we still use relational tables to temporarily store query information as the user is composing it, and in any case the top level information as well as XML is stored in a relational table. I won't describe the XML itself because, other than nesting the many-to-one tables within the top-level definition, it is more or less a direct translation of the relational model: the column names correspond to XML attributes.

The major difference is that the relational model (which is more “low-level”) includes the *detailed metadata* associated with each attribute, which is necessary for the correct generation of SQL, whereas the XML has only the *bare minimum* necessary for identification of the attributes. *However, the XML is still managed within the database schema itself, rather than as a text file stored by the user on disk.* Specifically, it is stored as an XML column in the Query_Definitions table described below.

When the XML is compiled into SQL, the detailed metadata associated with each attribute must first be fetched, and we store that information into in-memory data structures. This way, if the underlying data corresponding to the attribute changes its representation – i.e., some of the metadata's values have changed – then the XML can be recompiled before execution. It is straightforward to implement a batch routine that checks whether the date/times that any queries were last modified are older than the date/time of last modification of any attributes that they depend upon; if so, the dependent queries are recompiled.

15.5.6

Schema for Query Representation

Figure 15.5 shows the E-R diagram for the query-definition subschema. There are three tables: a top level Query Definitions table, and two many-to-one tables. **Query_Columns** holds individual non-temporal criteria (and attributes that should be output); the other, **Query_TimeCriteria**, holds individual temporal criteria.

In stored queries, the contents of the latter two tables are actually represented as part of an XML stream that is stored in the Query Definitions table.

I have not shown a **Query_Attributes** table (with two columns, Query ID, Attribute ID) whose only purpose is to record dependency information on which attributes are used in a query. It is this table that allows changes in specific properties of an attribute – storage representation, data type, choice Set – to allow the queries that depend on this attribute to be identified and flagged for recompilation.

The flagging of queries as out-of-date is performed by an update trigger on the Attributes/Parameters table: I prefer to flag rather than immediately try to recompile the affected queries because a developer may change several attributes one after the other: the same queries may be involved for more than one of these attributes. Immediate recompilation without waiting for the developer to finish would introduce needless delays (possibly recompiling the same query after each attribute change). Therefore recompilation is best

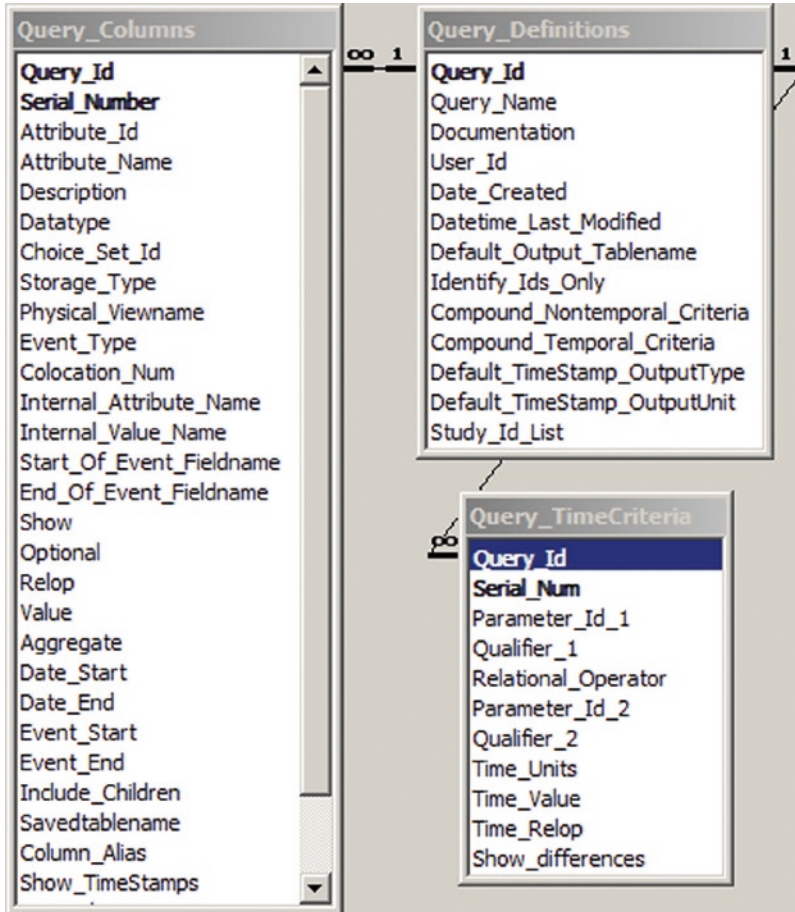


Fig. 15.5 Sub-schema for query definitions

left to a batch process that is invoked on a regularly scheduled basis (i.e., nightly) as well as on demand through a button in the developer interface.

15.5.6.1

Query_Definitions

This table contains the following fields:

Query ID (surrogate primary key), **Query Name**, **documentation**, **ID of user** who created it, **Date-time created**, **Date-time last modified**. These are self-explanatory.

Default output Tablename: The name of the output table that will contain all required and single-instance -per-patient/object attributes.

Compound Non-Temporal Criteria, Compound Temporal Criteria: The rows in the many-to-one tables specify individual criteria. The power of relational query, of course, is the ability to combine such criteria into compound criteria using the logical Boolean operators AND, OR and NOT. Such operators have a relative precedence (NOT>AND>OR), and so parentheses must be used to override the default precedence.

The above two fields contain text expressions that use parentheses and Boolean operators, and reference individual criteria by a serial (row) number: an example expression is (1 and 2) or (3 and 4). Obviously, the individual criteria referenced by the row numbers must be checked for existence before the query is run.

If the expression is blank, it is assumed that all individual criteria are combined with an AND operation.

If both compound criteria are non-blank, it is assumed that the compound temporal criteria are to be AND-ed with the compound non-temporal criteria.

Identify IDs Only: Sometimes, the user may wish to specify several criteria, but does not want to look at any output other than the matching Patient IDs: in a non-clinical situation, the user may want to gather only matching *Object IDs*. These IDs are obviously non-interpretable by themselves, but can be saved to the default output table, and used later in another operation.

Default Time-Stamp Output Type: One of: Absolute, Relative (time-interval based).

Time-Stamp Output Unit: One of: minutes, hours, days, weeks, months, years. Only applies if the Output type above is Relative.

Query_Specification: This is an XML column that contains the XML definition of the query, discussed above.

Additional columns to utilize data regularity: In CSDMS situations, we allow the user to pick a *list of clinical studies* against which the data must be run. As stated, most of the time, users want to access a single study only, or less commonly a family of related studies. (Because of permission issues, it is rare that even an analyst user will have access to all studies.) Based on the studies selected, co-occurrence information can be computed on the attributes selected. In the case of a single study, co-occurrence information is typically pre-computed at study-design time. As stated earlier, this information may allow us to reduce the number of output tables.

15.5.6.2

Query_Columns: Non-temporal Criteria and Output

These have been mostly discussed in sections “selecting attributes of interest”, so I’ll be brief here. The Query ID is the foreign key into Query Definitions. Serial Number provides each criterion with an automatically generated sequential numbering. The columns **Attribute_ID**, **Attribute Name**, **Description**, **Datatype**, **Choice_Set_ID**, **Storage_Type**, **Physical_ViewName**, **Event_Type**, **Internal_Attribute_Name**, **Internal_Value Name**, **Start_of_Event_Fieldname** and **End_of_Event_Fieldname**, **SavedTable**, **Column_Alias** all contain metadata and are initialized when the attribute is picked. In the user interface, only Attribute Name and Description are visible to the user.

Note that if the representation of the data underlying a given attribute changes, or if the data type of that attribute is changed, many of these columns would change their values, which is why a higher-level (XML) representation of the query, which is more immune to such changes, is necessary. As stated earlier, the changeable columns' values are set during a compilation phase, though we use in-memory data structures rather than tables.

The columns **Show**, **Optional**, **Relop**, **Value**, **Aggregate**, **Event_Start**, **Event_End**, **Date_Start**, **Date_End** and **Include_Children** contain the user's criteria and output options. **Show_TimeStamps** (Boolean) determines whether the time-stamps associated with the attribute are also output – the nature of the output depends on the settings in **Query Definitions**.

Co-location number is an integer that is computed either in advance or during query composition. Pre-computation is possible in CSDMSs when the query focuses on a single study: it is computed from the CRF-Event grid (which specifies which forms, and therefore which attributes, are recorded in the same Events. In addition, all attributes that occur in the same repeating group share the same co-location number, but this number is unique across the study. The idea of co-location number is that attributes with the same number can be placed together into the same output table.

In addition, the **Query_Columns** table also contains higher-level groupings for the attribute, such as the **Class_ID** (for EAV/CR designs) and **CRF_ID** (for clinical situations).

15.5.6.3

Study_TimeCriteria: Temporal Criteria

A temporal criterion is of the form:

Qualifier-1 Non-temporal row-reference -1 relational_operator

Qualifier-2 Non-temporal row-reference -2

time-units time-value

Qualifier-1 and **Qualifier-2** are one of: Begin or End. For Instant attributes with a single time-stamp, Begin and End are indistinguishable.

Non-temporal row-references 1 and 2 refer to serial numbers of rows in **Query_Columns**.

Relational operator is one of the standard operators: =, <, <=, >, >=.

Time-Units is one of the standard time units (minutes to years).

Time-Value (optional) is an integer that is combined with time-units, e.g., 3 days.

Time-Relop: One of the standard relational operators, as above. Default is equals (=).

Show differences: Boolean. If true, the differences between the two time-stamps are shown.

Time-Relop, Time-Value and Time-Units are specified together to specify bounds for the difference between the time-stamps associated with the two parameters (or derived values of parameters). In the expression below, they are specified in **Bold Underline**.

BEGIN (parameter-1) > BEGIN (parameter-2) <= 4 days

The above means that we are looking for data where the onset of parameter 1 follows the onset of parameter 2, but the difference between the two is less than or equal to 4 days.

A Detailed Example: In studies of cancer chemotherapy, it is important to determine the time-lag between onset of therapy and the time when the total white blood cell (WBC) count reaches a minimum (the nadir): many chemotherapeutic agents affect WBC production. The process of determining time-to-nadir (in days) is as follows:

Assume that the **Query_Columns** table contains the Chemotherapy Onset (represented as an EAV field of type date, with no separate time stamp) in row 1, and an aggregate operation (MIN) on the WBC count (a number) in row 2. You want only the minimum WBC count *following* the Chemotherapy Onset Date, which must therefore be *less than* (<) the start date (BEGIN) of the minimum WBC count. Also, you don't care about the actual dates: you want only the difference in days.

The fields of a record in Study_TimeCriteria would contain the following information:

Qualifier-1: BEGIN
 Non-temporal row-reference-1: 1
 relational_operator: <
 Qualifier-2: BEGIN
 Non-temporal row-reference-2: 2
 time-units: Days
 time-value: null
 time-relop: null
 Show-differences: True.

If the user specifies that the two attributes specified in Query_Columns should be shown, there will be five columns in the output: Patient ID, Chemotherapy Onset date, the numerical value of the minimum WBC count, the date when the minimum occurred, and the difference (in days) between the minimum-WBC-date and the chemotherapy onset date (i.e., time to WBC nadir).

15.5.7

Generating SQL for Ad Hoc Query

The paper of Desphande et al.⁶ describes an older version of the algorithm for code generation. This paper was written before two technological advances that have resulted in considerable improvement in performance as well as some simplification of the problem. These are:

- The introduction of common table expressions (CTEs) in MS SQL Server and Oracle. CTEs are expressions that behave like temporary tables or subqueries. Like subqueries, they exist only within the scope of a single SQL statement, but have the advantage that, once defined, they can be referenced multiple times in the same query. Their advantage over temporary tables is that existing indexes on the tables used in the CTE can be utilized by the optimizer: by contrast, when a temporary table is created, it is un-indexed. (Our earlier approach created a large number of temporary tables, one for each criterion, and then

combined them in a stepwise fashion. When the first step was complete, any indexes on the original data could not be utilized, which was an issue if the output was voluminous.)

- A still older paper, Nadkarni and Brandt⁷ first described the ad hoc query problem for clinical data, and described the use of macros (“templates”) to get around limitations of the proprietary (and still crippled) SQL dialects that were then the only means of creating stored procedures. Today, the possibility of writing arbitrarily complex stored procedures using a full-featured procedural programming language simplifies that task greatly. While in general I’ve tried to emphasize SQL portability in the software that I’ve created (for example, I try to use ODBC where possible), the advantages of stored procedures here are overwhelming. (You can still call them, once defined and tested, from ODBC, so you have some portability.)

There are basically two approaches to use depending on whether you are dealing with the time-oriented, sparse-data problem – as with clinical data situation – or with non-sparse data (as in EAV/CR designs). If the data is non-sparse, the algorithm is simpler, as there will be only one output table. We will deal with the simple case first, and discuss the complex case next.

For clinical data, all parameters apply to patients. Therefore the basic process of query for clinical data can be stated as follows: first identify patients matching a set of criteria. Then, fetch the parameters on these patients that the user has asked to see. As stated earlier, in general these parameters will not fit into a single table because they have different frequencies. If you *must* create a single table, you get the associated time-stamps for all parameters, compute the union of all these time-stamps (rounding the date-times to a reasonable unit, typically the date part of the time-stamp), and then perform a left outer join between this union and the individual time-stamped parameters. Such a single table has numerous empty cells and is suitable for display rather than analysis.

In either case, the steps are:

1. Forming a conceptual compound query. The conceptual query is a Boolean combination of non-temporal and temporal criteria, specified at a skeletal level, where the individual criteria are denoted only by numbers that serve as placeholders.
2. Converting the units of the conceptual query into actual SQL. This involves expansion of the individual placeholders, using common table expressions that are built up from individual criteria the columns specified as output.
3. Modifying the final query, if necessary, by restrictions that are either row-based, column-based, or both.

I’ll now discuss each step in greater detail.

15.5.7.1

Forming a Conceptual Compound Query

The first step is to convert the compound query into a representation that allows execution.

- Convert the infix expression in the compound non-temporal and temporal criteria to postfix, which eliminates parentheses. There are several examples of infix-to-postfix converters, e.g., all of these must rely on operator precedence.⁸ Thus, the infix expression:

(1 AND 2) OR (3 AND 4 AND 5)

is converted into the postfix

1 2 AND 3 4 AND 5 AND OR

Infix-to-postfix conversion allows evaluation of the expression using a stack: see the Wikipedia entry on Reverse Polish Notation for the algorithm.⁹ The way this algorithm works is by scanning the expression: as long as a value (operand) is encountered, it is pushed on to the stack. The moment an operator is encountered, we check that the number of operands on the stack matches the arguments it takes. (“NOT” takes 1 argument: “AND” and “OR” take two.) If this is not the case, the expression has an error. all operators take two operands.

If we encounter either of the patterns:

operand-1 NOT operand-2 AND (e.g., not x and y)

or

operand-1 operand-2 NOT AND (e.g., x and not y)

we replace these with the pattern

operand-1 operand-2 AND_NOT (introducing the limited set-difference operator).

If the final operator on the stack is a NOT, we don’t allow the expression as it might be computationally prohibitive.

- If a sequence of AND or OR operators is encountered, convert these into N-ary operations. Thus, the sequence

W AND X AND Y AND Z (where w, x, y and z are numbers) is converted to

W X Y Z AND-4.

In an N-ary AND or OR operation, one operates on N arguments in a single step rather than in N-1 steps involving argument pairs. An N-ary capability is useful because SQL itself allows such single-step shortcuts. In terms of implementation, all that N-ary means is that we use the original operator but store a number along with it to indicate how many arguments it happens to be using.

- Rearrange the operands of an N-ary sequence so that they are sorted by their higher-level grouping, such as the physical table or the class. Subdivide the sequence into as many units as there are groupings. Each unit can now be used for optimizations. Thus, if the pair w and x refer to attributes modeled conventionally and in the same table, conventional columnar SQL can be generated. In the EAV/CR scenario, if the unit refers to EAV-modeled data in the same class, the entity columns can be intersected with each other and additionally filtered based on the entity’s class.
- Group the criteria and output attributes: Segregate all detail rows (whether these specify criteria, output or both), based on the higher-level group. In EAV/CR, the segregation is by Class; in clinical-data scenarios, the segregation is by co-occurrence. For each group created, the conceptual query is of the form: Select *attribute-list* from *group* where *criteria*.

15.5.7.2

Translating Conceptual Queries into SQL

I’ll illustrate the task of generating SQL with several scenarios, progressing from the most simple to increasingly complex. Each subsequent scenario is a superset of the previous, in that approaches developed in solving the previous scenario are utilized in the next.

Table 15.1 Transforming the contents of the Query_Columns table into SQL for the simple case of attributes mapping to a single physical table or view

Row number	Physical view name	Internal attribute name	Data type	Show	Relop	Storage type	Value
1	Patient_View	Gender	Enumerated	N	=	Physical	1 (Female)
2	Patient_View	City	String	N	Like	Physical	Milfor*
3	Patient_View	Date Of Birth	Date	Y	<=	Physical	1/1/1950
4	Patient_View	Full Name	String	Y		Physical	

Case 1: The Group Is Based on a Single Physical Table or a View That Does Not Use EAV Modeling

Here the SQL corresponds more or less directly to the conceptual query. For example, consider the following snapshot of selected rows in the table Query_Columns in Table 15.1 above:

The content of these three rows would translate into the SQL:

```
Select FullName, DateOfBirth from Patient_View where
Gender=1 and City like 'Milfor%' and DateOfBirth <= {ts '1950-01-01'}
```

The framework uses the “data type” information to place quotes around string literals and escape date literals using ODBC syntax: the date literal also needs to be transformed so that year of date is placed first. The wildcard symbol (*) is replaced with the ANSI SQL equivalent (%). The framework uses the “Show” column to determine which columns to retrieve and which to use for the WHERE clause. (The DateOfBirth column is used for both.)

Case 2: The Group Is Based on One or More Physical Tables or Views That Use EAV, with Output Destined for a Single Table

This situation occurs in the EAV/CR scenario, where EAV modeling is used to simulate a third-normal-form design and non-sparse data with a highly volatile schema. There will only be one final output table because all attributes that are eventually displayed in the query can eventually be directly or indirectly related to an entry in the Objects table, and the number of attribute instances per object is determined by the simulated schema.

The problem gets more challenging because full outer joins may have to be used (except for attributes guaranteed to be present, where inner joins can be used). The process of composing the individual WHERE sub-clauses based on data type is the same as demonstrated above: the difference lies in how everything is combined.

I'll illustrate this for the EAV/CR scenario using the example of the Marengo et al. 2003 paper,¹⁰ whose full text is available in PubMed Central. This deals with a query related to odor molecules (substances with a distinctive smell), which are characterized by the following attributes: structural formula (displayed as a diagram), functional chemical group, a unique alphanumeric identifier (called the Chemical Abstracts Service (CAS) Number)

and an empirical formula. The last is defined by a list of chemical element symbols each followed by the number of atoms in the molecule – thus the empirical formula for ethyl alcohol would be C-H₄-O. (Hyphens between elements are not part of the standard empirical formula, but we add them to assist search.)

The query seeks to retrieve the details of all odor molecules in the database that have an acid, aldehyde, or alcohol as the functional group, and either have an Object Name that begins with the letters “He” or contain six carbon atoms. The Storage Type for all attributes is EAV, except for the Object Name (which corresponds to the Objects Physical table). The Internal Attribute Name (the name of the physical column) for all EAV attributes is “Value”. All attributes except Structural Formula are mandatory. The Class ID for the Class “Odor Molecule” is 457, and we apply this as a filter to limit the search in the Objects table (Table 15.2).

I will first show the query that is composed and then describe the strategy for composing it. The query itself is shown below (Table 15.3).

Table 15.2 Details of a query asking for odor molecules with specified characteristics

Row number	Attribute name	Data type/physical view name	Attribute ID	Show	Relop	Value
0	Object_Name	String		Y	Like	He*
1	Functional group	Enumerated/Enum_View	1245	Y	In	acid, alcohol, aldehyde
2	Empirical formula	String/String_View	567	N	Like	C6-*
3	Structural formula	Binary/Binary_View	846	Y		
4	CAS number	String/String_View	901	Y		

Table 15.3 SQL for the query correspond to the specification of Table 15.2

```
With
S0 as (select Object_ID, Object_Name from Objects where Class_ID=457 and
Object_Name like "HE%"),
S1 as ( select Object_ID, Value as Functional_Group from Enum_View where
Attribute_ID= 1245 and Short_Name in ("acid", "alcohol", "aldehyde")),
S2 as (select Object_ID, Value as Empirical_Formula from String_View where
Attribute_ID= 567 and Value like "C6-%"),
S3 as (select Object_ID, Value as Structural_Formula from Binary_View where
Attribute_ID= 846),
S4 as (select Object_ID, Value as CAS_Number from String_View where
Attribute_ID=901),
Select Object_Name, Functional_Group, Empirical_Formula,
Structural_Formula, CAS_Number from
(((s0 inner join s1 on S0.Object_ID=S1.Object_ID)
inner join S2 on S0.Object_Id=S2.Object_ID)
inner join S4 on S0.Object_ID=S4.Object_Id)
left join S3 on S0.Object_ID = S3.Object_ID
```

The strategy for composing the query is as follows:

- Each individual condition uses a common table expression (CTE) which is a `SELECT` statement assigned an alias flagged using the word `WITH` that is used within the scope of an outer `SELECT` statement.
- Multiple CTEs can be used within the same scope, separated by commas: here, we simply name each alias using the convention `S<row-number>`.
- Notice that in the EAV/CR design, the `Objects` table is central to the query, in that it must be joined to every other CTE. Here, the CTE corresponding to the `Objects` table happens to be row zero. (Note: even if the query did not specify the `Objects` table explicitly, the framework would still need to introduce it into the query.)
- All CTEs except `S3` (which specifies `Structural_Formula`) are associated with mandatory attributes. That is, every instance of an `Object` must have an instance of these attributes. Therefore the CTEs `S1`, `S2` and `S4` can be inner-joined to `S0`.
- The CTE corresponding to `Structural_Formula`, `S3`, is optional (i.e., may be missing). `S0` must therefore be left-outer-joined to `S3`. (That is, all instances of `S0` and any `S3` rows that happen to match.)

Note that in EAV/CR situations, the decision to implement certain classes as conventional physical tables is up to the system designer. If this is the case, the strategy illustrated in Case 1 (shared physical table and columnar attribute representation) can be combined with that illustrated above.

Case 2a: Combining Output from Multiple Classes in EAV/CR

The above example does not address one of the common situations in EAV/CR modeling where foreign-key relationships between tables are simulated through the use of the `EAV_Objects` table. An example of a query that accesses this table repeatedly in the neuroscience area is: “identify chemicals that oppose the action of inhibitory neurotransmitters in the motor neurons of the spinal cord”.

Here’s some background on the science related to this query, for the non-neuroscientist reader. The spinal motor neurons are responsible for the movements of voluntary muscles. The well-known poison strychnine opposes the action of the amino acid glycine, and so “strychnine” would be one of the rows in the answer. Glycine exerts a “calming” or inhibitory effect on spinal neurons, by binding strongly to molecules called glycine receptors on the surface of these neurons. Strychnine binds strongly to the glycine receptor, preventing glycine from being effective: the result is muscular excitability in small doses (e.g., twitching), and severe generalized convulsions and death with suitably high doses of strychnine. The tetanus bacillus also causes death by a similar mechanism: tetanus toxin prevents *release* of glycine from other neurons (that regulate the action of the motor neurons).

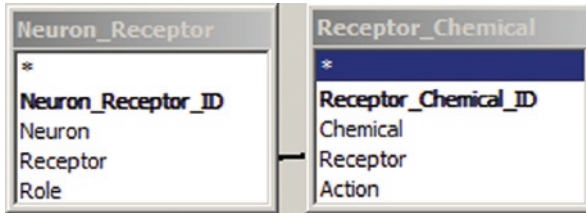


Fig. 15.6 Classes used to capture information relevant to the query of case 2a. The class Neuron-Receptor records many-to-many information about receptors associated with neurons and their roles (excitatory, inhibitory). The many-to-many class Receptor_Chemical records information about the actions of chemicals on individual receptors (antagonist, agonist). The classes Neuron, Receptor, and Chemical have not been shown: these would contain detailed information (e.g., molecular formula for chemical and location for neuron); however, basic information – specifically, Object Name – is in the objects table; and the fields neuron, receptor and chemical really contain object IDs

This query must be formulated in terms of the simulated data model, which is illustrated in Fig. 15.6. I have stripped the classes down only to the two required to answer the above query; the descriptive information that will be displayed to the reader is retrieved from the Objects table.

To understand the SQL that is generated, remember that certain information shown in the diagram is captured using the EAV_Objects table. Thus, the data for an individual record of the class Neuron_Receptor is represented in the following way:

- Every record has a unique identifier (Neuron_Receptor_ID), which is really an Object ID.
- The “Neuron” information is stored in the EAV_Objects table as a triple: Entity=Neuron_Receptor_ID, Attribute=“Associated Neuron” (Attribute ID 441), Value=Neuron ID (points back to Objects table).
- The “Receptor” information is stored in the EAV_Objects table as a triple: Entity=Neuron_Receptor_ID, Attribute=“Associated Receptor” (Attribute ID 442), Value=Receptor ID (points back to Objects table, as for Neuron ID above).
- The “Role” information is an enumeration, whose data is represented in Enum_View (Attribute ID 443).

The data for an individual record of the class Receptor_Chemical is represented similarly, using Attribute IDs 445, 446 and 447 for “Chemical”, “Receptor” and “Action”; the last is also an enumeration.

To understand how the query works, imagine that in Fig. 15.6, we had Object names instead of IDs, and two physical tables instead of simulated tables. Then we would simply have a two-table join. Since neither of these conditions holds, we must essentially create the equivalent of the two physical tables through individual common table expressions. The framework can perform the operation in a single statement, which is reproduced in

Table 15.4), though I'll use alias prefixes (NR=Neuron_Receptor, RC=Receptor_Chemical) to indicate individual CTEs.

Notice that the queries utilize the views Enum_View and Objects_View defined in Figs. 15.1 and 15.2 respectively: For Objects_View, this allows us to access the Object Name corresponding to the Value column of EAV_Objects directly, while for Enum_View, the Short Names (brief descriptions) correspond to enumeration codes are also accessed directly.

The CTEs with the prefix NR are joined on the shared Object_ID (for which we use the alias Neuron_Receptor_ID), and the CTEs with the prefix NC are joined similarly on the shared Receptor_Chemical_ID. Since there are three “Value” columns for both Neuron_Receptor and Receptor_Chemical, we need three CTEs for each.

Finally, the SELECT statement utilizes all six CTEs to display the columns of interest (the Neuron, the Receptor and its corresponding antagonist Chemical) and specifies a WHERE clause – selecting Neurons with the name “Spinal Motor Neuron”, Receptor role as “Inhibitor” and Chemical action as “Antagonist” (Table 15.4).

Table 15.4 SQL for the query of case 2a: this query runs against an EAV/CR model that uses objects as values

```
With
NR1 as (select Object_ID as Neuron_Receptor_ID, Child_Object_Name as
Associated_Neuron from Objects_View where Attribute_ID = 441),
NR2 as (select Object_ID as Neuron_Receptor_ID, Child_Object_ID as
Receptor_ID, Child_Object_Name as Receptor_Name from Objects_View
where Attribute_ID = 442),
NR3 as (select Object_ID as Neuron_Receptor_ID, Short_Name as Role from
Enum_View where Attribute_ID = 443),
RC1 as (select Object_ID as Receptor_Chemical_ID, Child_Object_Name as
Chemical from Objects_View where Attribute_ID = 445),
RC2 as (select Object_ID as Receptor_Chemical_ID, Value as Receptor_ID
from Objects_View where Attribute_ID = 446),
RC3 as (select Object_ID as Receptor_Chemical_ID, Short_Name as Action
from Enum_View where Attribute_ID = 447)
Select NR1.Associated_Neuron, NR2.Receptor_Name, RC1.Chemical from
((((NR1 inner join NR2 on NR1.Neuron_Receptor_ID =
NR2.Neuron_Receptor_ID) inner join NR3 on NR1.Neuron_Receptor_Id =
NR3.Neuron_Receptor_ID)
inner join RC2 on NR2.Receptor_ID = RC2.Receptor_ID) inner join RC1 on
RC2.Receptor_Chemical_ID = RC1.Receptor_Chemical_ID) inner join RC3
on RC2.Receptor_Chemical_ID = RC3.Receptor_Chemical_ID
where Associated_Neuron = "Spinal Motor Neuron" and NR3.Role =
"Inhibitor" and RC3.Action = "Antagonist"
```

Case 3: The Attributes May Need to be Segregated into Multiple Output Tables

This occurs, as I have stated earlier, with clinical data where the number of instances per patient varies greatly both across patients and across individual attributes. By contrast with the previous example, where linking of attribute instances is based on Object ID, here it is based on Patient ID and a time-stamp – either an actual date/time (typically rounded to a time interval such as day), or a relative time-stamp, such as an event that is defined in a clinical study.

Special Considerations for Clinical Study Data: While every event in a study has a machine-generated Event ID, one typically uses an Event Serial Number instead – this is a unique integer (not necessarily sequential) where a larger value indicates that it occurs later in the study. The serial number can be converted to actual time-interval units at the end of the query, by using the Study Calendar.

I've stated earlier that *co-location number* should be utilized to reduce the proliferation of output tables. This can be computed dynamically (by determining which attributes occur in the same Form, if non-repeating, or in the same Group, if repeating) or statically, based on the Event-CRF grid. Attributes that occur just once, such as Demographic parameters like Patient Name and Date of Birth, can also be grouped together.

Once the attributes are grouped based on co-location, the final number of output tables can be determined and presented to the user prior to running the query. This confirmation step shows the attributes for each table. While each output table may be assigned a machine-generated name by the framework, the user will almost always want to substitute meaningful names.

At this point, individual SQL statements are constructed that use common table expressions (CTEs), as shown in Case 2/Table 15.3, with the difference that the links between CTEs are based on Patient ID plus time-stamp – so that there will generally be multiple rows of data for each patient. As in Case 2, either inner joins or left outer joins are used based on whether an individual attribute is expected to be non-null (mandatory) or optional.

Example:

Show Patient ID, Gender, Date of Birth, serum Bilirubin, Alanine Transaminase and International Normalized Ratio (INR, a test of blood clotting function) in Patients in a clinical Liver Disease study who have Cirrhosis and a negative History of Alcoholism and at least one Bilirubin value > 10 mg/dl.

Background Information: Patient ID is used everywhere; Gender and Date of Birth are present in a Demographics table; Bilirubin and Alanine Transaminase are tests of liver function that are commonly performed together: if one is recorded, so is the other. INR is typically performed to monitor patients who take certain blood-thinning drugs (e.g., warfarin), but is also evaluated in patients with liver disease prior to surgery – it is not, however, part of the routine liver function test panel. Cirrhosis and History of alcoholism are recorded together once, at the start of the study. (While most cases of cirrhosis involve alcoholism, it may sometimes occur following exposure to certain toxic chemicals, such as carbon tetrachloride.)

We assume that all the views that define type-specific EAV data also include the Event_ID and the Event Date. Since all patients do not join a study at the same time, the Event Date may vary across patients for the same Event_ID.

Table 15.5 Query definition for Case 3: query of clinical study data.

Row number	Attribute name	Data type/ physical view name	Attribute ID	Show	Rel Op	Colocation number	Value
0	Gender	Enumerated/ demographics		Y		1	
1	Date of birth	Date/ demographics		Y		1	
2	Bilirubin	Real/Real_View	1024	Y	>=	2	10
3	Alanine Transaminase (ALT)	Real/Real_View	1028	Y		2	
4	INR	Real/Real_View	2032	Y		3	
5	Cirrhosis	Boolean/ Int_View	3019	N		4	True (1)
6	Alcoholism	Boolean/ Int_View	3021	N		4	False (0)

Steps: (see query definition in table 15.5 above)

1. Find all patient IDs matching the selection criteria: Cirrhosis=True, Alcoholism=False and Bilirubin >= 10; save the results to a temporary table, named “tmpPatients”.

With S5 as (select Patient_ID from Int_View where Attribute_ID=3019 and Value=1),

S6 as (select Patient_ID from Int_View where Attribute_ID=3021 and Value=0),

S2 as (select Patient_ID from Real_View where Attribute_ID=1024 and Value >=10)

select distinct S5.Patient_ID into tmpPatients from S5, S6, S2 where S2.patient_ID=S5.Patient_ID and S6.Patient_ID=S2.Patient_ID

2. Get the required attributes for these patients. Utilize co-location number, and this time use Event_ID so that co-located attributes are combined into the same output row. There will be three output tables: the attributes for cirrhosis and alcoholism are not output, since we know what the exact values will be. For all queries, utilize tmpPatients, obtained in the previous step, to filter the list of patients.

(a) Select tmpPatients.Patient_ID, Gender, DateOfBirth into Demog_Output from Demographics, tmpPatients where tmpPatients.Patient_ID=Demographics.Patient_ID.

Comment: The Demographics table does not use Event ID: we presume that Date of Birth (and Gender) don't change for the study duration.

(b) Get the liver function test results.

with S2 as (Select Patient_ID, Event_ID, Event_Date, Value as Bilirubin from Real_View where Attribute_ID=1024),

S3 as (Select Patient_ID, Event_ID, Event_Date, Value as ALT from Real_View where Attribute_ID=1028)

select tmpPatients.Patient_ID, S2.Event_ID, S2.Event_Date, Bilirubin, ALT into LiverFunc_Output from (tmpPatients inner join S2 on tmpPatients.Patient_ID=S2.Patient_ID) inner join S3 on S2.Patient_ID=S3.Patient_ID and S2.Event_ID=S3.Event_ID order by tmpPatients.Patient_ID, S2.Event_ID,

Comment: Here, we have to do a double join on Patient_ID and Event_ID since we are interested in all values of Bilirubin, not just those that are above 10. For a given Patient

and Event, the Event_Date will be fixed, so we can pick it from either S2 or S3. The results are sorted by Patient_ID and Event_ID.

(c) Get the INR: we can use a CTE, but it is not necessary.

Select tmpPatient.Patient_ID, Event_ID, event_Date, Value as INR from tmpPatients, Real_View where Attribute_ID=2032 and tmpPatient.Patient_ID=Real_View.Patient_Id order by tmpPatient.Patient_ID, Event_ID

As for Case 2, if any of the attributes were optional, then tmpPatients would need to be joined to the CTE for those attributes using left outer joins.

15.6

Special Topics in Query and Reporting

15.6.1

Pre-compilation of Queries

If certain queries or reports (which utilize queries) are run repeatedly, one can, in principle, achieve some improvement in performance by converting the SQL within them to stored procedures. The rationale is that if a query is run directly, the database engine spends some time parsing it and then working out a data-access plan in a query-optimization phase, finally executing the plan. For a stored procedure (which is compiled), the first two steps are skipped since they are performed at compilation-time, and so some time is saved. (However, if the database structure related to the tables involved in the query has changed – e.g., additional columns or new indexes have been added – then the procedure must be recompiled to recompute the optimal data-access plan.)

When pre-compiling queries into stored procedures, one should look for opportunities to make them reusable by parameterizing them. This is particularly true of reports: the elements defined in the table Meta_Report_Parameters are obvious candidates for such optimization.

15.6.1.1

Performance Issues: A Warning

Don't rush to convert everything into stored procedures for efficiency reasons alone, unless you first benchmark performance with and without compilation, and find the difference in performance significant enough to matter in actual deployment. With certain database engines, notably MS SQL Server, if you use the vendor dialect of SQL alone for stored-procedure development, you cannot modularize code and hide information. All procedures are visible at a top level, even those that are only called by others, so you can end up with hundreds of stored procedures that you must document separately. (Oracle does a much better job, letting you create code modules called *packages*.)

If you need such modularity, you must develop stored procedures using a traditional programming language (i.e., C# or VB.NET). However, if you are using this approach simply so as to create wrapper code that executes straight SQL, Microsoft warns that you might actually

see a *drop* in performance. The SQL that is embedded as strings within your VB.NET/C# stored procedure is *not* optimized – it is executed as though it had just been typed in the first time, and there is additional overhead because of inter-process communication.

15.6.2

Accessing Non-relational Data: XML and Narrative Text

15.6.2.1

Full-Text Indexing and Query

All the big three RDBMS support storage, indexing and query of narrative (full) text. This is an area where the capabilities (and the query syntax) of each vendor are completely proprietary: the DBMS vendors saw the potential of indexing free text long before the ISO SQL committee got their act together.

The current RDBMS versions fully integrate text search with search of structured data in queries. This allows the query optimizer to achieve significant performance gains for mixed queries, which access both text columns and traditional structured data. For example, if progress notes are recorded as text, and the diagnosis is in structured (coded) form, and we want to search an EMR for all patients with Parkinsonism who are observed to be agitated, it would seem reasonable for the optimizer to use the diagnosis as a filter (since this diagnosis is present only in a small fraction of the total patients) and then search the progress notes associated with the patient subset for all variants of the word “agitate”. (MS SQL Server achieved this integration only in 2008.)

RDBMSs free-text capabilities do not attain the sophistication of Natural Language Processing (NLP): being oriented to large volumes of data, they utilize Information Retrieval (IR) techniques, briefly introduced in Chapter 12. The steps used by RDBMSs are described briefly below.

- Input text is broken up into individual tokens using tokenizers or “word breakers”. In English, tokenization simply uses punctuation as a guide, but for languages like German, where compound words are common, the process is more complicated. RDBMSs are provided, out of the box, with tokenizers for several languages, allowing multi-lingual capability.
- Words belonging to a *stop list* are removed from the tokens. Language-specific stop lists – containing extremely common words in the language such as articles, prepositions and conjunctions, which have no value for indexing – are also provided by the RDBMS.
- Language-specific *stemmers* convert words to a standard inflectional (“stem”) form: for example, “go”, “went”, “gone” and “goes” become “go”. The original text is preserved, but the index uses the stemmed version.
- A language-specific thesaurus is also provided, usually in the form of an XML file, but an installation can add its own files. A thesaurus can be configured to be sensitive to accents (important for French) or not. This contains two types of entries: expansion sets (a group of equivalent synonyms) and a replacement set (indicating that one phrase should replace the other phrases in the set): Microsoft’s example is that “NT5” and

“W2K” should be replaced with “Windows 2000”. This is equivalent to designating a *preferred term*. While the original text is stored, the thesaurus is used to perform *query expansion*.

15.6.2.2

Querying Full-Text

In MS SQL Server, full-text query capabilities are invoked with the function `Contains (column-name, query-expression)` embedded within your SQL `SELECT` statement. The *column-name* is the name of the full-text column (which should have been indexed) and the *query expression* is where you specify your search pattern. Among the patterns are:

- Whole Words or Phrases: You can specify individual words, e.g., “vomiting”, or a phrase (words with spaces, such as “morning sickness”). For phrases, punctuation between words in the original text is ignored.
- Wildcards: Specifying the pattern “head*” will search for all words beginning with the prefix “head”.
- Proximity Searches: You can search for words “near” other words: a certain number of words apart, in the same sentence, in the same paragraph. Capability varies widely with RDBMS.
- Inflections/Synonyms: You can specify that all inflections (e.g., past tense, plural form, etc.) or synonyms of a particular word be used to broaden the query.
- Language: You can specify that query be limited to text in a specified language. (When entire documents are stored in full-text columns, each document is associated with a Locale Identifier that specifies its language.)
- Compound Boolean Queries, using AND, AND NOT, OR: Any of the above can be combined using Boolean operators e.g., “head*AND vomiting” will return only those records containing the whole word “vomiting” as well as the prefix “head”.

If multiple text columns in a table are indexed, SQL Server also allows `ContainsTable (Table-name, query-expression)` which searches all columns for the specified pattern. There is also a function called `FreeText( column-name, phrase)`, which takes an arbitrary phrase and does a Google-style “similarity search” after the engine tokenizes, stems and expands it using a thesaurus. After the search, matching documents are ranked based on how closely they match the search phrase.

15.6.2.3

Querying XML Data

I’ve touched on RDBMS XML capabilities in an earlier chapter. While capabilities vary widely, it is possible to index columns containing XML data for better performance (consult your RDBMS manual for details). The query optimizer, as for full-text, can process mixed queries (that access XML columns and traditional columns), using the latter to reduce the number of rows whose XML content needs inspection.

Certain XML documents, notably Microsoft Word documents stored in the Office XML format, contain both XML as well as narrative text. It is possible to additionally create *full-text indexes on XML columns* containing such documents: the XML tokens are treated as token boundaries. This allows the best of both worlds: searching the content irrespective of how it is formatted, and searching for formatted information (such as headings or hyperlinks).

I've stated earlier that, in general, I don't advocate using XML as an alternative for data-type-specific EAV tables (even though it is possible to do so). The real challenge in dealing with XML is stated below.

15.6.2.4

Challenges in Querying Text and XML

Text and XML illustrate in a particularly vivid manner the challenges of creating a universal interface that is both powerful and intuitive enough to be operated by novices. XQuery, in particular, while a very powerful tool, has somewhat of a learning curve (it can use regular expressions, for example) and is highly programmer-oriented. Additionally, to use it productively in many circumstances requires knowledge of the hierarchical structure of the XML schema associated with a given column.

There are commercial tools, such as Stylus Studio (www.stylusstudio.com) that come with graphical user interfaces that facilitate composition of XQuery graphically – using features such as IntelliSense, syntax coloring and code completion – but to use them productively, you have to know XQuery: these tools increase the productivity of a professional significantly (in the same way that Visual Studio and Eclipse do), but they will not teach a total novice what XQuery is all about.

For full-text, the problem of building a user interface is somewhat more tractable, as the variety of operations is relatively more limited. (For a general ad-hoc query interface, this must be implemented as a subsystem, with an interface that is brought up only on demand for full-text columns.)

My friend and former colleague Dr. John Fisk implemented such an interface, described in.¹¹ This work, published in 2003, is somewhat dated: at the time most RDBMSs lacked the full-text muscle they exhibit today, and integration of full-text and relational data in the optimizer did not always exist. So we used a programmable search engine (dtSearch, still an excellent and highly affordable package for indexing of Web sites) to implement such integration manually by partitioning the query into a relational part and a full-text part, executing the former first, and passing the result set to the latter to restrict the full-text search. Today, however, I would use the direct approach.

15.7

Bulk Data Extraction

Bulk data extraction is intended to transform data into an analyzable format. Columnar-modeled data is already analyzable. Special-purpose row-modeled data may occasionally need transformation into columnar form – though several data mining

association-detection algorithms, notably Agarwal and Srikanth's a priori algorithm,¹² will operate on such data directly. EAV-modeled data almost always needs to be transformed: as stated earlier, this transformation is referred to as *pivoting*.

In general, for clinical data, where there is a variable number of attribute instances per patient, you cannot meaningfully pivot any arbitrary set of EAV-modeled attributes into one giant table: only attributes that are associated with each other through a higher-level grouping can be pivoted together. This is the same reason why, for ad hoc query of clinical data, output goes into multiple tables rather than a single table: the presence of a higher-level grouping ensures that the number of instances of these attributes is roughly the same (allowing for some attributes to be optional).

15.7.1

Extraction of Clinical Data

In the case of EMRs or CSDMSs, attributes that you pivot together must lie in the same form: however, in the case of “repeating groups” that are used to model many-to-one relationships, each repeating group must be pivoted separately. The pivoting algorithm has been described in Dinu et al.¹³ This algorithm is memory-intensive, but fast: it is almost identical to that described in Sect. 15.4.6.1 earlier. It operates on one grouping at a time; thus, to extract all data for a study, one iterates over all case report forms. Within each form, one separately extracts the non-repeating single-instance information into one extract, and then operates on each repeating group (if any exists), each into its own extract.

The algorithm relies on creating a 2-D array supported by a pair of hash tables: the rows – the “entities” – are form instances for single-instance data, and group instances for repeating-group data. The columns are the individual parameters/attributes, along with additional columns such as patient-identifying information and time-stamps. By using hash tables that determine the row and column into which a value should go, the full-outer-join problem is bypassed. Data can be read one type-specific EAV data view at a time: the rows need not be sorted.

We output the data as tabular delimited text files with the first row containing the column names: this format can be imported into almost any package. It is a straightforward matter to generate a database schema from the metadata to hold this data.

15.7.2

Extracting of Data with Sub-structure

Pivoting of data in EAV/CR-based designs can get more complex because attributes are not necessarily atomic: many attributes have sub-structure – they are themselves entities with attributes of their own, and some of these attributes may also be entities, and so on. Here, one would want to extract data a class at a time and get details of all instances – objects belonging to that class.

As I've stated in an earlier chapter, the table EAV_Objects table is used to capture sub-structure. In theory, one would have to perform an indefinite recursive search, each time

fetching all the values of a child object's attributes' values: recursion would stop when all atomic values were obtained. However, the full list of attributes can get enormous, and in reality, the user may not care for the details of every single object. For example, for certain classes of objects, only the name of the object, and possibly a few selected attributes, might suffice.

Therefore it is best to create an interface that allows the user to specify, through a point and click mechanism, precisely which attributes are desired, and save this specification for future use. In such an interface, which would use a "tree-view" widget, the user can start from a class, expand attributes that happen to be classes, select child attributes, and so on. In the interface, if the user selects a class-type attribute but does not select any of its child attributes, only the Object Name and Brief Description/caption will be presented. The order of the output columns can be specified by repositioning individual items within a "Chosen Attributes" list box.

In most situations, the level of recursion will be pre-determined, based on your simulated schema – i.e., your metadata design. Therefore, all branches of the tree can be determined in advance, at extract-definition time. In uncommon situations, however, you may have a recursive design, where *the child attribute belongs to the same class as the Parent*. This can occur, for example, in a bill-of-materials hierarchy or a manager-employee hierarchy. Extraction such as we are now describing is not intended to deal with such data, as the number of levels of recursion is unpredictable and depends on the data – in fact, pivoting is not even a proper approach for displaying such data, which is best presented using a graph or tree-drawing program. There are several open-source and commercial programs – beginning with the venerable GraphViz¹⁴ developed at AT&T Research – that, given a set of node pairs and link information, will draw a picture for you. As far as bulk extraction is concerned, the best you can do is display the basic information (name and caption) of the child object.

15.7.2.1

Extracting Array Data

Extracting attributes of the array type into a neat rectangular structure that is part of the table containing non-array attributes is problematic. This is because, with a sequence of child values for a given object, the number of values is almost certainly likely to vary with the individual object. This means that, if you intend to export the data into columnar form for use by an analytical package, it is not possible to export the data for a class that contains one or more array-type attributes into a *single* rectangular structure, because the number of columns is not predictable in general.

However, the data can be exported into a rectangular format if you consider how array data is modeled in traditional RDBMS design. You would create a new table, which is related many-to-one to the parent table. This table, which holds array values, borrows the primary key from the parent table, and has two additional columns, one representing a sequence number (an array index), and the other the value at that index.

You would *extract each array attribute into its own table*. In case the values are non-atomic (as in the case of the EAV_Objects table), you would use the child Object's summary description as the value.

Unless you wish to allow the user to arbitrarily recurse through the details of child classes modeled as arrays, it is simplest to make your extract-specification user interface sense whether individual child attributes are modeled as arrays, and prevent expansion of such attributes in the tree-view if they happen to be classes. In other words, the user can select such attributes for output, but only summary information will be generated, and this will be output separately.

15.7.2.2
The Extract Specification

The sub-schema that supports extract specification and the extraction process is shown in Fig. 15.7. The left half of this diagram shows the specification proper. This is stored in a pair of tables, **Extract_Specification** and **Extract_Details**. The latter table is related many-to-one to the former.

Extract_Specification has a machine-generated surrogate primary key (*Extract_ID*), a reference to the class to be extracted (*Main_Class_ID*), and also has documentation fields (*Extract Name*, *Description*, *Date Created*, *Date last Modified*, *Created By User*). An additional column, *Sort_Objects_By* specifies the column in the Objects table by which the Objects are sorted when first fetched. (This is typically Object Name, but an alternative criterion, such as the date/time an object was created, may be used.)

Extract_Details serves as the database analog of a tree structure, whose nodes are the attributes of interest. The way to represent a tree (or any graph data structure) in a database is to record three sets of columns, the first describing the *source* node, the second the *destination* node, and the third the *connection* between them. (In a tree, the source is the *parent*, and the destination is the *child*.)

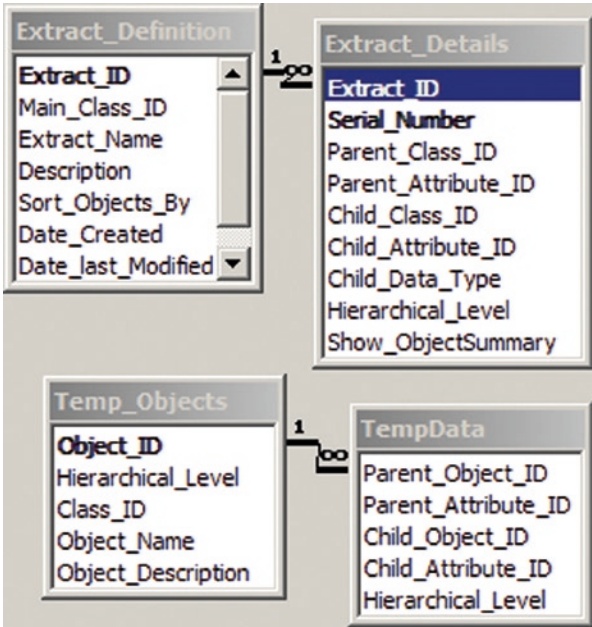


Fig. 15.7 Sub-schema for data extraction: EAV/CR design

The columns in this table are described below.

- **Extract ID:** Link to the top-level Extract Specifications table.
- **Parent Class ID, Parent Attribute ID:** These define the parent node. Note that if you have implemented custom data types and allow the same attribute to be used in multiple classes, the combination of fields identifies a node uniquely. However, your schema may have a strict many-to-one relationship between Attributes and Classes – this is the design implemented by Dr. Luis Marengo at Yale – so that given an Attribute ID you can determine its Class by looking it up in the Attributes table. Even here, however, at the cost of four extra bytes, you can save a lookup by redundantly storing the Parent Class ID.
For the top-level attributes belonging to the target class, both of these will be zero.
- **Child Attribute ID, Child Data Type, Child Class ID:** These define the child node. The data type is redundant, but also save a lookup. For atomic (non-class) attributes, of course, the Child Class ID will be null.
- **Hierarchical Level:** A sequentially generated number. Level 0 indicates the top level of the tree – i.e., the immediate descendants of the class that we want to extract. This number is used to sort the rows within a given extract ID: extraction starts from level 0. The level of a given attribute can be computed from the tree structure representing the class and its child classes.
- **Serial Number:** Indicates the position of the attribute in the final output. For array attributes, this column should be null, since they will be output to separate tables.
- **Show Object Summary:** Boolean; this applies only when the Child Attribute represents a class. After navigating from a parent object to a child object in order to get its attributes, the user may wish to show a summary description of the child Object (i.e., name and caption/brief description), or may not – the child, while part of the tree – so that child objects must still be retrieved and stored – is used only to access its attributes.

The primary key of this table is (Extract_ID, Serial_Number).

15.7.2.3

Temporary Data Tables for Hierarchical Object Data

Given that the extraction process is arbitrarily recursive, intermediate objects pulled from the database must be stored temporarily in special tables for further use if needed. The schema for temporary storage is shown on the bottom of Fig. 15.7. The two tables, **Temp_Objects** and **TempData**, are indexed for improved performance. Their contents are purged at the start of the extraction process and filled later.

Temp_Objects stores summary descriptions of objects along with the level of the tree at which they are accessed: Its columns are a subset of the Objects table (Object ID, Class ID, Object name, Object Description) along with a Level column that is copied from the Extract_Details row during each step. Its contents are populated, one level at a time, in Step 3 of the Extraction Algorithm described in the next subsection.

TempData is used to store child objects that are fetched by query of a three-table join – EAV_Objects, Extract_Details and Temp_Objects, a level at a time, in Step 2 of the Extraction Algorithm. Population of **TempData** alternates with population of **Temp_Objects** until there is no more data left to extract.

Values of atomic attributes from the other EAV tables, while accessed, are not stored in tables: they are copied immediately to memory (see step 1 of the algorithm).

15.7.2.4

Extraction Algorithm

The extraction algorithm relies on a 2-D string array supported by two hash tables that help to place a value in an appropriate cell in the array. The number of rows in the array is equal to the number of objects in the main class. The number of columns is defined by the number of output columns defined in `Extract_Details` for the given `Extract ID`. A row defines an output column if the child data type is not “Class”, or if the child data type is “Class” and *Show Object Summary* is True. (While the summary fields – typically the name and description – are in two columns, we will place them concatenated and delimited into a single column to simplify the algorithm.) The number of columns is determined in advance, at the time the extract is defined. The number of rows, of course, depends on the data.

The hash table that helps identify the row has the key-value pair (`Object_ID`, row-number). The `Object IDs`, when initially fetched from the database, are sorted by the `Sort_Objects_By` column.

The hash table identifying the column of the array has the key value pair (parent-attribute-id| child-attribute-id, column-number): the numeric parent and child attribute IDs are concatenated with a delimiter-character. Attributes may not be unique in the extract, and so for disambiguation, we need both the parent link and the child link. An example of duplicate Attribute IDs is when a class is linked to another class more than once – e.g., ship-from address ID and ship-to address ID both point to class “Addresses”, and we want to display the source and destination cities, both referring to the attribute `City` within `Addresses`.

The algorithm initially fetches all Level 0 objects – i.e., all objects that belong to the target class ID – into `TempObjects`. Then it sets an integer variable, `current_level`, to zero, and repeats the following four steps:

1. It gets the values of all atomic attributes at the current level. Since we know the data type of each attribute, we know which table to access: we can improve performance, however, by accessing all attributes of a given data type at a time. The algorithm populates the array at the positions of these attributes (treating them as children the parent attribute ID is zero).

```
SELECT XD.Parent_Attribute_ID, E.Attribute_ID as Child_attribute_ID,
       E.Object_ID, E.Value
FROM Extract_Details XD, EAV_Int E, Temp_Objects O
where XD.Hierarchical_Level = O.Hierarchical_Level AND XD.Parent_Class_ID =
O.Class_ID
AND E.Object_ID = O.Object_ID and XD.Child_Attribute_ID = E.Attribute_ID
and XD.Hierarchical_Level= current_level and Extract_ID = current_extract_ID
and XD.Data_type = data_type
```

The first two fields (`Parent_Attribute_ID`, `Child_Attribute_ID`) in the statement determine the column, the third column (`Object_ID`) determines the row in which the fourth field (`Value`) is placed.

2. A virtually identical query appends rows to TempData by querying the EAV_Objects table.

```

INSERT INTO TempData
( Parent_Object_ID, Parent_Attribute_ID, Child_Attribute_ID, Child_Object_ID,
  Hierarchical_Level )
  SELECT E.Object_ID, XD.Parent_Attribute_ID, XD.Attribute_ID, E.Value,
    current_Hierarchical_Level FROM Extract_Details XD, EAV_Objects E,
    Temp_Objects O
  where
    XD.Hierarchical_Level = O.Hierarchical_Level AND XD.Parent_Class_ID =
    O.Class_ID
    AND E.Object_ID = O.Object_ID and XD.Child_Attribute_ID = E.Attribute_ID
    and XD.Hierarchical_Level= current_Level and Extract_ID = current_extract_ID
    and XD.Data_type = class

```

The only difference is that the current level is also inserted, and the extract rows are filtered on the data type “Class”. (Data type, as stated in an earlier chapter, is a single letter: Class would be indicated by “C”.)

3. Summary information is fetched for all Objects matching the Child_Object_ID values at the current level, but the Level associated with them is incremented by 1.

```

INSERT INTO Temp_Objects
( Object_ID, Class_ID, Object_Name, Object_Description, Hierarchical_Level)
  SELECT O.Object_ID, O.Class_ID, O.Object_Name,
    O.Object_Description, current_level+ 1
  FROM TempData TD, Objects O
  where TD.Child_Object_ID = O.Object_ID
    and TD.Hierarchical_Level = current_level

```

4. Increment *current_level* by 1.
5. Repeat steps 1 through 4 until Extract_Details has no records where the Hierarchical_Level matches *current_level*.
6. Finally, display all summary Object descriptions by joining TempData, Extract_Details and Temp_Objects filtering by the current extract ID where Show_ObjectSummary is True.

```

SELECT XD.Parent_Attribute_ID, TD.Child_Attribute_ID, TD.Child_Object_ID,
  TObj.Object_Name, TObj.Object_Description
  FROM Extract_Details XD ,TempData TD, Temp_Objects TObj
  where
    XD.Child_Attribute_ID = TD.Child_Attribute_ID AND
    XD.Hierarchical_Level = TD.Hierarchical_Level And
    TD.Child_Object_ID = TObj.Object_ID And
    XD.Extract_ID =current_extract_Id AND
    XD.Show_ObjectSummary=True

```

15.8

Exporting Metadata to Make Extracts or Ad-Hoc Query Output Comprehensible

Extracts are normally destined for statisticians, many of whom, while good at their job, are not necessarily comfortable with SQL, despite SAS's having introduced the PROC SQL command to the SAS programming language to allow combination of data from multiple tabular datasets, and who may not have spent significant time working with your system. In any case, when you provide someone with tabular output that is intended for further analysis, they shouldn't be forced to explore your system to find out what the parameters in this output mean. You must therefore provide at least a bare minimum of metadata to allow the data to be interpretable, after it has been imported into an external analytical package.

The bare minimum comprises a variable name, data type, a caption and description of enumerations (for any variables based on enumerations). Ideally, this information should be output in a format that specific analytical packages (e.g., SAS, SPSS) can import and use directly. This usually means generating basic scripts in the programming language of the target analytical package. SAS, for example, has a command called PROC FORMAT that is used to describe enumerations as sets of code/description pairs.

You may have to wrestle with issues such as variable-name length: SAS has a limit of 32 characters, and if you permit longer variable names (we do ourselves), you may have to use an approach of programmatically truncating the name to 30 characters and adding numeric suffixes for disambiguation.

15.8.1

Metadata for Meta-analysis

Meta-analysis is a statistical technique that is used to complement systematic review: it seeks to combine the results of multiple studies concerning a common theme, to draw broad conclusions about a particular area. The original paper by Mary Lee Smith and Gene Glass,¹⁵ (which defined the term "meta-analysis") pooled the results of a large number of studies involving psychotherapy, claiming to prove that any form of psychotherapy resulted in better patient outcome than no psychotherapy. This work was subsequently attacked by critics who claimed that combining patient outcomes across diverse forms of therapy into a single analysis was, in effect, comparing apples, oranges and sushi. The contribution of this paper was to show that, under appropriate circumstances – if the variables of interest to the analysis objective are common to all studies, or can be transformed into such a common form – such comparisons were legitimate. However, such circumstances may be somewhat exceptional: based on the meta-analytical research hypothesis, it may not be legitimate even to compare oranges and tangerines.

While meta-analyses began by pooling summarized results (p values) across studies, it may be possible, especially in these days where federal funding of studies often mandates sharing of the raw data after an embargo period – to access the raw data itself. (The journal that Gene Glass edits – *Education Policy Analysis Archives*, <http://epaa.asu.edu/> – was one of the first open-access journals, and among the first to insist that any one publishing a quantitative study in the journal must consent to deposit all the raw data at the journal website.)

15.8.1.1

Describing Parameters in Depth

In any case, the task of the meta-analyst is simplified considerably by detailed descriptions of the variables in a study – which includes the validation checks defined on them, the units in which they are measured, and so on. To minimize experimental variation, clinical researchers have been specifying parameter measurement with precision in great detail.

For example, to accurately compare the effects of two medications (which are known to be therapeutically effective) for chronic hypertension, one must control all the conditions that can influence the measurement of arterial blood pressure. For example, there is a diurnal variation in blood pressure, and if you measure a person's blood pressure immediately after arrival at a clinic, and 10 min later (after that person has had a chance to settle down) there will be a significant difference in readings. Therefore, the detailed description of blood pressure measurement in a given study must include measurement conditions (e.g., activity level, position) and body site where measured, time of day when measured. The OpenEHR foundation has advocated the idea of *archetypes*^{16,17} –rigorous and maximally agreed-upon *computable* definitions of clinical concepts – that include definition of all variables that influence the concept.

Speaking from personal experience, the acid test of how well your individual parameters are documented in your system is provided by an attempted meta-analysis by someone else. This is when any slipshod practices and short-cuts taken by members of your team become apparent. As to what baseline level of documentation is satisfactory, my personal belief is that no variable should be documented within your system in less detail than would be required if such a description was included in a peer-reviewed publication.

15.8.1.2

Facilitating Data Transformation

Automated or semi-automated facilitation of meta-analysis of multiple data sets by electronic inspection of element definitions is an unsolved problem. Amalgamation of data from separate studies is only possible under either of two circumstances:

- One or more parameter definitions across two or more studies are determined to be semantically identical – that is, they map to the same controlled-terminology elements, and additionally use the same data type, units, and enumeration. In the case of laboratory values, they should be measuring the same analyte, as reflected by more or less the same range of normal values.
- It is possible to define a mathematical transformation that converts two parameters from different studies into a common grain, sometimes with minor or major information loss.

To illustrate a worst-case information-loss scenario, if one study measures smoking by number of years smoked currently and in the past, while another measures the same by cigarettes per week, all that the merged data can tell us is whether a given individual is a

non-smoker, ex-smoker or current smoker. The resultant information may be useful or it may be worthless for the hypothesis that you intend to explore.

Automatic or facilitated transformation of variables is a hard problem in all but the most trivial cases. The problem is that many concepts in research – especially in areas that are under-represented in the standard biomedical vocabularies, such as psycho-social areas – do not have an exact mapping to concepts in such vocabularies. It is possible to use post-coordination to create mappings, but it takes a lot of effort, and the results may not always be usable. (Try mapping items such as “I am bothered by things that usually don’t bother me”, one of the panel of questions in a self-evaluation scale for depression.)

Appendix: Composing SQL that Uses Relationship Metadata

Understanding this section requires some computer-science exposure – specifically, graph theory. If you need to bone up, a data-structures/algorithms book that is reasonable and not overwhelming is Sedgewick.¹⁸ This section is relevant to the ad hoc query of non-sparse, heterogeneous data as modeled using an EAV/CR framework, though it also applies to a conventionally modeled schema.

The Universal Database Concept

A traditionally implemented (column-modeled) database schema can be regarded as an *undirected graph* data structure, where a table can be regarded as a vertex or node, while a primary-key/foreign key relationship between two tables can be treated as an edge. *Undirected* means that one is permitted to navigate in either direction along any edge, even though the ends of the edge have a specific meaning in terms of primary key vs. foreign key.

In theory, if one wants to retrieve columns from several tables, through criteria on columns that are dispersed across several tables, a program could help in composing part of the SQL – specifically the join expression:

- For a pair of physical tables that are directly related, the join expressions is: *primary-key-table.primary-key-field = foreign-key-table.foreign-key-field*. If there are multiple columns in the join, the above is repeated, and the individual column-pairs are combined with an AND operator.
- If the paths between the tables whose columns the user has specified are known, *the software could automatically include intermediate tables in the path, even without the user having specified these tables*.

An idea originating in the late 1980s with David Maier and Jeffrey Ullman – the “Universal database” (UDB) – takes this insight to an extreme level. The UDB concept attempts to create the illusion, for the end-user, that a database consisting possibly of hundreds of tables is one gigantic table with a large number of columns. In other words, the user concerns oneself only with individual columns, and need not be bothered with the existence of

individual tables: foreign keys in particular, can be hidden from the user. The software SAP Business Objects (BO) is based on this premise: in BO terminology, the virtual giant table is the “Universe” and the “Objects” are the individual columns.

Weaknesses in the UDB Premise

In terms of graph theory, the basic UDB approach is guaranteed to work only for schemas that happen to be *trees*, where there is *only one path that connects any two tables*. (If there is more than one path, the schema is said to have *cycles*.) In real schemas, such an assumption never holds. There are two common conditions where the assumption is violated: see Fig. 15.8 below.

- One instance of cycles is where one table is directly joined to another more once. (In Fig. 15.8, there are two links between A and D.) An example is an employee-hierarchy table, consisting of two columns, Manager ID and Employee ID, both these columns are linked to a Persons table that contains details of the individual. Another example is Ship-From and Ship-to address IDs in a sales order, both of which point to an Addresses table.
- Another cycle is where a table is directly linked to other tables, which also happen to be linked to each other by different paths. (In Fig. 15.8, C is linked directly to A and B, but A is also linked to B, so to get from A to C, one can use the direct route A-C, or the indirect one A-B-C.) This happens in reference databases of biomedical information where multiple tables store different kinds of facts, but each fact needs to be backed up with a bibliographic citation. If all citations are stored centrally in a single table, then multiple tables may contain a foreign-key column, Citation ID, which points to the citations table.
- Another concern here is that it is not enough to know that two tables are connected, but *how* they should be connected in a specific circumstance. Most of the time, one needs a standard (inner) join, but at other time, one needs a left or right outer join, and very rarely a full outer join (the union of left and right joins).

In other words, the basic UDB paradigm will take you only so far: at some point, the user must leave Never Land and become aware of the real world, where separate tables exist.

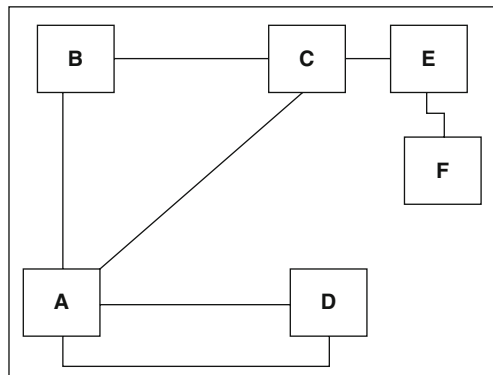


Fig. 15.8 Cycles in an undirected graph

Cycle Detection and Elimination

In Business Objects, setting up a “Universe” involves detecting and eliminating cycles. The standard algorithm for cycle-detection in undirected graphs is based on depth-first search: see Bard¹⁹ for an accessible explanation. Essentially, one maintains lists of edges and vertices: all vertices are set to “unvisited”. Starting with any arbitrary vertex, you move along each of the edges connected to it that have not already been used. Any unvisited vertex encountered is set to “visited”. If a previously visited vertex is encountered, the edge that got you to this vertex is a cycle, or part of one. For example, in Fig. 15.8, starting from A, you would reach D, B and C. Moving on to B, you would reach C, which has already been visited: therefore A–C is a cycle. Moving from D to A would similarly detect a cycle.

To eliminate cycles, you need to duplicate the multiply-connected vertexes: in terms of database schemas, you introduce alternative aliases for a table. You keep repeating this process until no cycles are detected. In Fig. 15.9, which is a transformation of Fig. 15.8, adding the two aliases A2 and D2 for A and D would *seem* to have eliminated cycles.

However, the problem gets more complicated. Remember that in the double-link between A and D of Fig. 15.8, *two separate columns in one table (the foreign keys) are linked to the primary key of the second table*. Let’s assume that D is the foreign-key table with two columns Col1 and Col2. (These could be Ship-From and Ship-to address IDs.) Let’s assume that Col1 is linked to A, and Col2 is linked to A2. Then, in Fig. 15.9, the only way to go from Col2 to B is via A2 and C (which may not be what the user wants). Similarly, any route from Col1 must involve B, which again may not be what is needed. So one must duplicate each D node once again, creating two new nodes D3 and D4, with D3 linked to A, and D4 linked to A2.

You must deal with the outer-join problem in a similar way. Let’s say that you mostly link A and B with an inner join, but occasionally need to use A right-outer-join B. You would duplicate A to create A3 (you could alternatively duplicate B – the choice is immaterial). You could then designate the link A–B as an inner join and A3–B as a right outer join. In general, if there are N cycles, you have to create N*2 extra aliases in the worst case.

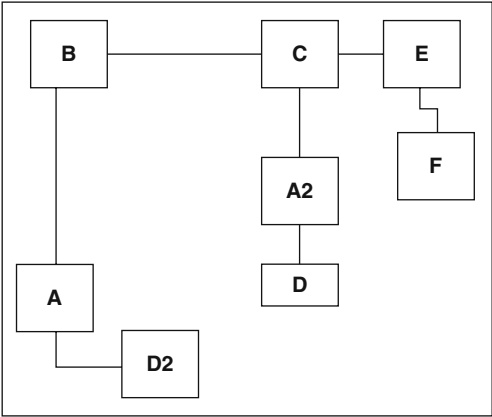


Fig. 15.9 Cycle elimination by vertex duplication

Can Database Schemas be “Hidden” Successfully Using a UDB Approach?

As the number of multiple links between tables grows, the setup of a “Universe” becomes increasingly laborious. For the dubious “benefit” of hiding the physical schema and inter-table relationships from a supposedly naïve and E-R-diagram-phobic user, the virtual UDB schema can become much more complex than the actual physical schema. There are several practical issues here.

- The UDB concept originated at a time when GUIs were non-existent, and E-R diagrams existed only on paper as a modeling tool. SQL, including join expressions, had to be entered mostly by typing. In most modern environments, E-R diagrams are nothing to be scared of. Modern query GUIs such as MS-Access’s VQBE generate the join expression between two tables for you if you click and drag between the two columns that you want to connect. (You can specify the join type by changing the connection’s properties.) VQBE will actually utilize existing relationships in your schema, if you select two tables that happen to be directly related, pre-connecting them. (If you select two tables that are not directly connected, VQBE is currently not smart enough to include intermediate tables. You need to refer to the E-R diagram and add intermediate tables manually – which is not too big a deal.)
- If you still insist on going the BO route, a fairly challenging issue involves giving the duplicated column aliases meaningful names that would somehow indicate the path desired by the user when a particular alias (or aliases, such as D4 and A2 in our running example) were picked implicitly. This is a problem that I believe contains an intractable contradiction: the user who is supposedly too naïve to know about separate tables must now be forced to not only deal with the tables but also the paths between them.
- In my experience, serious analysts, who know SQL and are quite comfortable with E-R diagrams, find the BO approach particularly condescending and confining: they are likely to spend more time fighting against the software than getting work done. Such individuals would be more productive with direct read-only SQL access to the schema. (An amusing episode involved watching a friend and colleague of mine, the Universe designer/maintainer for an institutional Business Objects installation, bypass BO completely and type SQL to answer queries posed to him by a researcher. This contradicts the well-known dictum about “eating your own dog food”: don’t force your users to use something that you wouldn’t use yourself.)
- Finally, in the case of EAV designs, where the same table can be joined to itself multiple times for set intersection operations, and many join operations are not readily predictable, setup of a “Universe” becomes so complicated that your typical developers – unless they are billing by the hour and primarily focused on job security – are likely to throw up their hands in frustration and turn in their resignations.

This is not to say that automatic join composition cannot be a source of developer productivity. The danger is of creating software that so insists on helping a user that it becomes a straitjacket or gatekeeper that persistently underestimates the user’s intelligence: it then becomes a database analog of the infamous Microsoft Office Assistant. Good software should know when to get out of the way, and should allow the power user ways of bypassing it if desired: VQBE, for example, lets developers type SQL.

Join Metadata in EAV/CR Designs

If you are using the EAV/CR approach of maintaining information on your inter-class/table joins in your metadata, this information can be used to compose join expressions. Remember that in an EAV/CR Design, the `Meta_Relationships` table's contents represent the edges in the schema's graph, while `Meta_Classes` and `Meta_Attributes` hold vertex information. The difference, compared to traditional schema is that, based on which classes are modeled using EAV, some of the relationships may be simulated rather than physical. (I'll explain shortly how join expressions can be composed for simulated relationships.)

At runtime, the graph is best loaded into memory to speed up the operations of cycle detection and join composition. The linked-list representation of the graph (see Sedgewick¹⁸) is most appropriate, though it can be optimized for performance. The edges are loaded into an array, and contain the primary-key class, the foreign-key class, and the foreign-key attribute: the primary-key attributes are known from the definition of the primary-key class. IDs are used rather than column names: the latter can be readily determined by lookup of the Class or Attribute IDs.

Given a set of classes that the user has specified, the problem is to find a unique path that connects these. This can be done by picking any class (vertex) in the list, and traversing its connecting edges until all the classes specified by the user have been reached. By the time this has been done, one or more intermediate classes may have been encountered. One then works with this subset of nodes to detect cycles. Conceptually, cycle detection is a second step, but in practice, by keeping track of which vertexes have already been visited, we can detect cycles simultaneously with the traversal process. There are also certain clues that would immediately identify cycles, such as a double relationship between a pair of tables.

1. If you do not detect cycles, exit; a join expression can be composed.
2. If you detect a cycle, a join expression cannot be composed without user assistance: the user must be shown the subset of the graph that contains the classes of interest, where the edges are labeled with the foreign-key attributes, and asked to disambiguate by selecting the desired edges (or deselecting the undesired edges). Note that, unlike the BO approach, we are assuming here that the user is aware of the database structure and knows about relationships. (The graph can be drawn using one of several graph-drawing subroutine libraries.)

This will work in most circumstances: sometimes, however, when there is a double relationship between a pair of classes, the user may specify *both* edges. (In the manager-employee situation, for example, the user may want details of both manager and employee.) In such a case, the software must create an extra alias for the primary-key table – preferably after having the user confirm that s/he really intends to use that table more than once. One may have to create aliases for other tables that are also intended to be used more than once, if they are part of the chain.

3. After the user has selected edges, go to step 1, working with the reduced edge-list this time. If any tables that are used more than once, the diagram must be modified to include the extra aliases.

Note that the level of software engineering to implement the above steps is not trivial. You should certainly not attempt graph-drawing from first principles: commercial programs also include smart layout algorithms that place the vertexes in 2-D space so as to prevent edges from crossing each other as far as possible: such packages are well worth the modest investment. However, for relationships that are simulated using classes modeled as EAV, the effort of composing SQL by working directly with the schema and using a commercial graphical query tool that is not EAV-aware can be considerable. If you have your developer team try to compose queries this way a couple of times, they are likely to find the experience so disheartening that you should be persuaded that it is better to bite the bullet and solve the problem once and for all with a framework.

Dealing with Simulated Relationships

The join-expression composition approach for a pair of tables is fairly straightforward. However, for simulated relationships in an EAV/CR design, the join is only modestly more complicated. Recall that every object has a unique `Object_ID`, which points back to the `Objects` table, and that foreign-key relationships are implemented through the `EAV_Objects` table. See Fig. 15.10 below. (This diagram has been composed using MS-Access VQBE.)

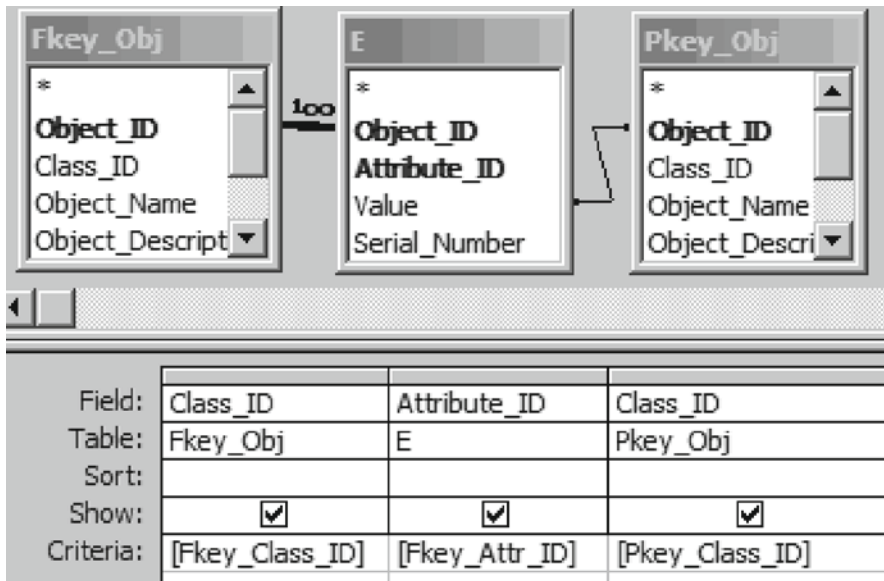


Fig. 15.10 An illustration of simulated relationships between classes that are modeled as EAV, and the SQL that is necessary to implement the equivalent join expression. The SQL is:

```
SELECT Fkey_Obj.Class_ID, E.Attribute_ID, Pkey_Obj.Class_ID FROM Objects AS
Fkey_Obj , EAV_Objects AS E, Objects as Pkey_Obj Where Fkey_Obj.Object_ID = E.
Object_ID and ON E.Value = key_Obj.Object_ID and Fkey_Obj.Class_ID=Fkey_Class_ID
AND E.Attribute_ID=Fkey_Attr_ID AND Pkey_Obj.Class_
```

Focus on the central table E (an alias for EAV_Objects) in Fig. 15.10. The Object ID column and the Value column both point to the Objects table; here, we have used two aliases for Objects, Fkey_Obj and Pkey_Obj (to indicate the foreign-key and primary-key classes respectively). The relationship information connecting the two classes has three columns: the Primary-Key class ID, the Foreign-key class ID, and the Foreign-Key class's Foreign-key Attribute ID. In the figure, we have denoted these in the Criteria row by abbreviated forms within square brackets: these would be substituted with actual IDs.

Fig. 15.10 also shows the SQL (automatically generated by VQBE) that implements the relationship: the criteria values have been italicized, to indicate that they are placeholders that would be substituted.

References

1. Chatterjee J. Working with parameters with crystal reports and ASP.NET 2.0. Available from: <http://www.aspfree.com/c/a/ASP.NET/Working-with-Parameters-with-Crystal-Reports-and-ASP-NET-2-0/1/>. Cited 11/1/10.
2. IBM Corporation. Full outer joins in DB2. 2010.
3. Snodgrass RT, Ahn I, Ariav G, et al. TSQL2 language specification. *ACM SIGMOD Record*. 1994;23(1):65-86.
4. Ben-gan I. Calculating the median gets easier in SQL Server 2005. *SQL Server Magazine*. 2010.
5. Johnson S, Chatziantoniou D. Extended SQL for manipulating clinical warehouse data. *Proceedings of the AMLA Annual Fall Symposium*; 1999:819-823.
6. Deshpande A, Brandt C, Nadkarni P. Metadata-driven ad hoc query of patient data: meeting the needs of clinical studies. *J Am Med Inform Assoc*. 2002;9(4):369-382.
7. Nadkarni P, Brandt C. Data extraction and ad hoc query of an Entity-Attribute-Value database. *J Am Med Inform Assoc*. 1998;5(6):511-527.
8. Pillai P. Infix to postfix conversion. Available from: http://scriptasyllum.com/tutorials/infix_postfix/algorithms/infix-postfix/index.htm. Cited 10/1/10.
9. Wikipedia. Reverse polish notation. Available from: http://en.wikipedia.org/wiki/Reverse_Polish_notation. Cited 9/1/10.
10. Marengo L, Tosches N, Crasto C, Shepherd G, Miller P, Nadkarni P. Achieving evolvable Web-database bioscience applications using the EAV/CR framework: recent advances. *J Am Med Inform Assoc*. 2003;10(5):444-453.
11. Fisk J, Mutalik P, Levin F, Erdos J, Taylor C, Nadkarni P. Integrating query of relational and textual data in relational databases: a case study. *J Am Med Inform Assoc*. 2003;10(1):21-38.
12. Agrawal R, Srikant R. Fast algorithms for mining association rules. *Proceedings of the 20th VLDB Conference*. Santiago, Chile; 1994.
13. Dinu V, Nadkarni P, Brandt C. Pivoting approaches for bulk extraction of Entity-Attribute-Value data. *Comput Methods Programs Biomed*. 2006;82(1):38-43.
14. AT&T Research. Graphviz – a graph visualization tool. Available from: <http://www.graphviz.org/>. Cited 8/2/10.
15. Smith ML, Glass GV. Meta-analysis of psychotherapy outcome studies. *Am Psychol*. 1977;32:752-760.
16. Leslie H. International developments in openEHR archetypes and templates. *HIM J*. 2008;37(1):38-39.
17. Kalra D, Beale T, Heard S. The openEHR Foundation. *Stud Health Technol Inform*. 2005;115:153-173.
18. Sedgewick R. *Algorithms in C++*. 3rd ed. Reading: Addison-Wesley; 2002.
19. Bard JF. Cycles in an undirected graph. Available from: <http://www.me.utexas.edu/~bard/IP/Handouts/cycles.pdf>. Cited 10/1/2010.

16.1

Introduction: Data Warehousing 101

A *data warehouse* is kind of database whose architecture (and underlying supporting technology) has been optimized for highly efficient query, at the cost of sacrificing features that support robust interactive inserts, updates and delete actions. The difference between a data warehouse and a *data mart* (which is also optimized for the same purpose) is partly one of scope. While warehouses are supposed to encompass data across an entire organization, data marts are typically smaller scale (e.g., departmental in scope) though in an ideal situation they would receive data from a warehouse, effectively serving as front-ends to the latter.

The differing scope in turn determines the choice of technology. Warehouses typically utilize a specialized RDBMS architecture, while data-mart implementations also have the option of utilizing multi-dimensional database (MDDB) technology (most of which has been acquired by RDBMS vendors), and/or utilize expensive (and relatively low-capacity) but extremely fast solid-state memory as a magnetic-hard-drive substitute.

Henceforth, for conciseness, when I use the term “warehouse”, I will (unless I specifically qualify the term) use it simply to mean a query-optimized database – i.e., either a data warehouse or a data mart. The architectural differences between a database that is specialized for query and one that is not (a *transactional system*), along with the rationale for the differences, are summarized below:

- Transactional systems must support data creation, update and deletion (CUD) by large number of multiple concurrent users. They must obviously support some query (you can’t edit data before finding it first), but interactive query capabilities are stripped down only to those necessary for the majority of users to do their jobs. Architecturally, this means that the number of indexes associated with a table (to facilitate fast search) is similarly restricted to only the columns that are frequently searched by end-users.

For every additional index associated with a table, whenever a record/row is created or deleted, the contents of each index must be re-adjusted. Similarly, if a row is altered, any indexes on the fields/columns that are changed must again be re-adjusted. Readjustment involves disk-based accesses and writes, which are slow compared to operations in memory: for a large table with numerous indexes, the delays introduced may be significant in the multi-user situation

Disk-based B+ trees, which I've introduced in [Chap. 4](#), are typically used as the underlying technology – for indexes in transactional systems: these offer a reasonable balance between query performance and interactive CUD performance (though even here, with numerous indexes, performance may be a problem as stated above.)

- By contrast, *the contents of a warehouse are typically populated by bulk data transfer from external production systems on a periodic basis*. Once a warehouse schema is designed, data must be extracted from production systems, checked for errors, cleansed, restructured, and then imported. These steps are collectively referred to as the ETL (Extraction-Transformation-Load) process. Kimball and Caserta's superlative "Data Warehouse ETL Toolkit"¹ describes the details (and the minefields) involved. This book is valuable because the knowledge within it has been acquired by the authors through actual large-scale implementations.

Because there are no interactive changes to content, the designer is free to utilize indexing technologies that allow extremely fast query even if these offer dismal CUD performance. Bitmap indexes and hash indexes, summarized in [Chap. 4](#), are examples of these.

The absence of interactive data changes means that *in warehouses, referential integrity constraints are considered unnecessary overhead*: all such constraints are verified and enforced during data preparation prior to loading the warehouse contents.

- The design of a warehouse schema may differ from that of a transactional schema. The most well-known design approach, originally devised by Ralph Kimball for data marts,² is called the *star schema*. This consists of a central *fact* table linked through multiple foreign keys to several *dimension* tables: in the E-R diagram, these surround the fact table like the spokes of a star.

The fact table contains detail (or sometimes aggregate) information on a particular subject area: patient observations in clinical medicine, sales or purchases in business applications. The dimension tables contain lookup information that pertains to individual aspects of the fact. For example, in the I2B2 clinical data mart schema, which we will discuss later, the dimension tables contain details of various aspects of clinical observations: the patient, the encounter, the provider, the underlying clinical concept. In a business scenario, a sales fact table would be related to dimension tables that contain information on products, salespersons, dates, customer, sales location (e.g., store) etc.

A fact table's foreign keys typically correspond to information from a *combination* of tables in the transactional system where the data originated. For example, while an "observations" fact table has columns for Patient ID and Provider ID, in a transactional schema, Patient ID would be stored in a higher-level table, and only the "entity" (the encounter ID) would be stored: an Encounter table would reference the Patient and Provider.

Information in the dimension tables is typically denormalized, so that all relevant information related to a particular dimension (e.g., Patient) is placed in a single table. Sometimes, however, their content is normalized into multiple tables. For example, a products table may be related to a higher-level product category table. The additional "branches" from the dimension tables yield what is called a "snowflake" schema, again from its appearance. Compared to star schemas, snowflake schemas take up less disk space at the cost of slower query.

The star schema allows the use of *join indexes* – indexes on a fact table that are actually based on columns in dimension tables. (These are usually bitmap indexes, because the columns have low cardinality.) For example, in epidemiology-related queries, it is important to find the frequency of particular clinical observations by the patient’s geographical area: the latter information is associated with the Patient dimension table.

16.1.1

Star-Schemas Are Not Always Mandated

In the Kimball design, an organization-wide warehouse consists of a series of fact tables dealing with different themes, all sharing common dimensions. For example, a Products table would serve as a dimension table for fact tables dealing with both sales and purchases. However, a star-schema-based design is not mandated.

It is important to remember that the star schema represents an optimization and design trade-off. It achieves excellent performance for a family of queries, chiefly those that group aggregates by attributes contained in the dimensions. In fact, MDDB software can automatically recognize a star or snowflake schema and interactively import data from the contents of such a schema. (MDDB technology achieves its high query speeds by pre-computing aggregates based on dimensional attributes of interest: when a user asks for such aggregates, the software simply presents the pre-computed data.)

However, other types of queries may be inefficient with a star schema. Bill Inmon, who originally coined the term “data warehouse”, differentiates between two categories of users, whom he calls “farmers” and “explorers”.³ The former need regularly scheduled reports whose content is stable, and may be well-served by a star schema. However, “explorers” – analysts who have a general idea of where they want to go, but haven’t worked out the exact details – may be better served by a relatively non-committal third-normal-form schema.

16.1.2

Approaches for Creating a Warehouse

Warehouse efforts that would encompass the entire scope of an organization take significant resources and time. As with all large projects, there are two approaches: top-down (designing and implementing the entire system), or bottom-up (starting with one area/department at a time). The sensible medium combines top-down design with bottom-up implementation (what Kimball and Caserta call “Planning Large, Building Small”). Top-down design does not have to be all-encompassing, but it avoids the risk of painting oneself into a corner. In the Kimball approach, top-down design using the star-schema approach means ensuring that the dimension tables’ structure and content is standardized so that they will meet the needs of multiple stakeholders, not just those where implementation will first happen.

The steps involved in implementation, irrespective of scope, are as follows:

- Data Identification/discovery: Determining what data will go into the warehouse: in what production systems it currently resides, what format it uses, and what subset of the conceptual tables/columns in a given source will be used.

- Identification of Data Constraints (“Business Rules”): Because the warehousing process involves getting familiar with a large number of data sources, constraint information must be pieced together in multiple ways: by inspecting the schema definition of RDBMS sources, inspection of source code (including stored procedures) if available, and interviewing individuals who work with the systems and understand their inputs and outputs. The older a source system, the more likely it is that constraints are enforced in application code rather than at the database level. Because of the nature of the discovery process, putative constraints must be verified and validated by determining how many records/fields violate a putative constraint, and deciding whether such violations are legitimate, in which case a putative constraint would have to be modified, or even removed.
- Data Profiling: This involves determining the quality of the data in terms of conformance to business rules as well as accuracy (in terms of measures such as spelling errors – usually determined through random sampling). Policies need to be developed regarding handling of errors of a particular kind in a given source: these may either be ignored, attempted to be fixed algorithmically, flagged for manual correction later while rejecting the record, or treated as catastrophic and requiring process termination.

It may turn out, from sampling of the data, that the quality of certain sources may be so poor that a major cleansing effort will be required, and sometimes the cleansing effort may dwarf the warehouse-construction effort proper. If widespread enough, quality issues may even be a show-stopper. A notorious example involved the Rite-Aid warehouse effort more than a decade ago, where it was discovered during the feasibility stage that there was no enforcement of drug terminology (which applied to inventory, purchases and sales): the generic acetaminophen was spelled by end-users in more than 20 different ways.

Once your organization has committed to building a warehouse or mart, the following processes are involved.

- Extraction of Source Data: This occurs in two phases: the initial extract, and incremental extracts. The initial extract is stored on dedicated hardware (the data staging area) for further transformation. Incremental extracts run on a regularly scheduled basis are either pushed from the source system, or pulled by extraction programs. Note that incremental extracts are only possible if the source system has a means of tagging records changed, added or deleted since the last extract (through the use of audit-trail tables and columns recording date/time of creation/last change). For legacy systems, such data may not exist, and so incremental extracts for such sources are not possible. Extraction involves the following sub-tasks:
 - Sub-setting: Selecting subsets of the columns (and rows) of individual classes.
 - Data Quality Checks and Cleansing: Simple scripts are written that identify the records/fields that violate individual constraints, run and the output saved for further inspection. (If the constraints are stored as computable metadata – a point I discuss in the next section – such scripts may even be generated.)

Data cleansing – fixing of the problems identified – is ideally performed once, in the source systems: otherwise, it would need to be performed each time new data is pulled. If, however, the source system is an inflexible black-box legacy system whose contents are not being updated and the errors can be fixed algorithmically, it may be logistically simpler to perform the changes after extraction.

- Transformation: Several types of transformation are performed. These do not necessarily have to be performed using RDBMSs: flat files manipulated by UNIX-type filters (grep, awk, sort, uniq) are often dramatically faster than RDBMSs for bulk transformation. There are numerous types of operations possible: the more common ones are:
 - Generation of Artificial Primary Key Values for Dimension Tables: This is an important function in warehouses, mainly because the source data comes from several different places. For warehouses of clinical data intended for research purposes, actual Patient Ids must be replaced with surrogate IDs this way. Key generation can be done by RDBMSs, but is often performed most easily by non-DBMS programs by appending sequentially generated numbers to sets of unique records.
 - Lookups: Once artificial primary keys are generated, these values are used to replace the original key values in Fact tables.
 - Sorting and Duplicate Removal. These are often used when working with flat-file text data.
 - Data Merge/Join: This involves denormalization of both fact and dimension data: most easily effected through SQL. The inverse operation, *conditional splitting*, involves separating data from a single table into two or more output destinations based on data values.
 - Generation of calculated values: The commonest computations involve calculation of aggregates using an aggregate function: count, sum, average, max, min, stdev – based on individual dimension columns or sets of columns. A common computation for clinical data involves calculation of a patient's age at the time an observation was recorded.

The transformation process is onerous, and several commercial tools (such as Microsoft SQL Server Integration Services) facilitate transformation through a GUI that allows a developer to create a chain of transformations (a *data flow*) and generates code that can be tested and executed later as a batch process. Be warned, however, that this code, even if output in a form such as XML, has a proprietary structure, and is not necessarily intended to be human-comprehensible: it is intended to be executed by an engine, or interpreted by designer software (e.g., presented as an editable diagram to the developer).

During transformation, it is important to record mapping (*lineage*) information that traces every row (and column) in the final output to its origins. At a minimum, this means that every row needs to be tagged with an ID representing its source system of origin, along with time stamps that represent when that data was created/last updated in its source. In the case of columns whose contents are computed, process metadata must record when the calculations were performed, and the source columns from which they were derived (the latter are part of the calculated expression). Mapping between source and final output (“target”) columns is thus many-to-many in the general case.

Lineage is critical for several reasons:

- When users wish to ask questions of the warehouse, they will frame questions in terms of variables/attributes in the production systems that they already know and use. So there must be a way of tracking the ultimate destination of the desired data elements. If end-result data are presented without the users being able to ask the software where they originated and how they were processed, they are unlikely to be believed.

- Lineage information (not just for source system variables but also their intermediate equivalents in the data staging area) allows *dependency tracking*, which in turn allows *impact analysis*: determining what processes or outputs are influenced by particular input (and changes to it).
- **Loading** the transformed data into the destination (warehouse or mart). Loading is typically performed in a particular order: dimension tables before fact tables. Certain operations that are independent of each other can be parallelized.

16.2

The Role of Metadata in Data Warehousing

Metadata plays a central role in almost all aspects of warehousing, and Kimball and Caserta's book, cited above, remains the best source for in-depth information. They describe metadata as the "DNA" of the warehouse – "everything about it except the data itself".

Warehouse vendors have converged behind a standard called the Common Warehouse MetaModel (CWM): a brief overview is provided by Vetterli et al.⁴ From the developer/user perspective, however, CWM is the equivalent of assembly language: it implies the use of the Unified Modeling Language and its XML counterpart, XML Metadata Interchange (XMI). CWM is intended for vendor tools to be able to interchange metadata with each other. However, due to the fact that the standard (available at⁵) has a large number of "optional features", the fact that a particular software package calls itself "CWM-compliant" provides no guarantee that it will interoperate well with other packages.

Kimball and Caserta also emphasize that even though the vast number of warehouse tool vendors all support recording of some kind of metadata or another, each tool only deals with one aspect of the problem, and there is still no end-to-end solution. The Kimball group⁶ also warns against a common pitfall: excessive reliance on vendor data-access tools to record metadata rather than capturing the same in your schema. Remember that many individuals – DBAs, statisticians, analysts – will not use the same data-access tool. They are more likely to use SQL directly. If nothing else, the use of an RDBMS-based shared metadata repository that they can browse (with added basic reporting capability) spares your organization the cost of buying multiple (expensive) tool licenses simply to use their metadata-viewing capabilities.

Further, the representation used by vendors is almost always proprietary. Even for a single vendor offering an end-to-end solution, each component of the vendor's toolset may maintain metadata in internal non-human-readable formats that may not even inter-operate with other components of the toolset. In the book "The Microsoft Data Warehouse toolkit", Mundy et al.⁷ point out that this "tower-of-Babel" situation applies even to the well-regarded and highly affordable SQL Server toolset, which is more than adequate for medium-sized operations.

While some vendors (e.g., SAS Business Objects) provide an API to access most of the metadata, if you wish to query and report on it in ways that the vendor did not anticipate, you have to write programs that pull this information out and import it into a relational structure anyway.

In brief, creating your own metadata infrastructure (starting with a relational schema) is unavoidable. In this chapter, I'll provide general guidelines for the most part, rather than a canned schema, simply because individual needs will vary widely: a solution that may be overkill in one scenario may be inadequate in another.

16.3

Toolset for Warehouse Metadata Management

To recapitulate Kimball and Caserta: metadata is everything about the warehouse except the data. In other words, even the program code used in the ETL process and documentation of various aspects of the system – front end, ETL, data sources, user training materials and standard operating procedures – falls under the purview of metadata. While I've been recommending an RDBMS-based design as the mainstay of all metadata architectures till now, it will not suffice for metadata as diverse as that encountered in a warehouse project – though it will still be a linchpin.

For management of code and documentation, industrial-strength version control systems that also incorporate bug tracking and feature requests, such as Microsoft's Team Foundation Server (TFS) or the open-source Subversion, become necessary, with front-ends for code development (Eclipse, Visual Studio) and collaborative documentation development (such as Adobe RoboHelp™). (Note that while TFS uses MS SQL Server as its back-end data store, it incorporates functionality that would take small teams years to implement on their own.)

16.4

Categories of Warehouse Metadata

The categories of metadata follow from the individual steps involved in warehouse creation.

16.4.1

Data-Source Metadata

Data-Source Metadata is built right from the identification and profiling stages. The metadata approach that we have detailed for EAV/CR – information on classes, attributes, relationships and enumerations – can serve as the starting point for recording this metadata, which is both technical as well as descriptive in scope. Please refer back to Chap. 11 if necessary.

These tables can actually serve to document the structure, *not only of your source data, but also of the tables in the data staging area as well as the final (presentation) area*. The advantage of recording all information in the same set of tables, as I'll demonstrate, is that it simplifies the important task of recording *lineage information*: the link between individual tables/columns in the presentation area to their ultimate sources, by way of intermediate staging tables.

However, for warehousing purposes, the EAV/CR tables need to be augmented greatly, with additional columns as well as with additional tables. This is because every class of data must originate in a particular source, and within a particular source, not all the classes/attributes are used in the warehouse. Finally, sources may not necessarily be RDBMS-based. The additional information that you need to record is categorized below.

16.4.1.1

Glossary: Concepts and Terms (Synonyms)

Many members of the warehouse team may not be familiar with the domain that the warehouse will address. Even for a healthcare organization, many concepts will relate, not to clinical medicine, but to business and administrative concepts. It is therefore important to develop a shared glossary, with detailed definitions of all concepts, and synonyms (terms) for individual concepts. While a basic relational structure (i.e., the concepts/terms table of controlled terminologies) is possible, it is also necessary to be able to generate electronic documentation that allows individual concepts to cross-reference (i.e., *hyperlink* to) each other via the definition text. Wiki technology⁸ is the most useful delivery mechanism.

16.4.1.2

Extra Columns for Meta_Classes and Meta_Attributes

Both these tables require an additional Boolean column – **Unused** (default False), which apply to Data Sources only, indicating if it is not used in the warehouse. For tables/classes that are unused, details of their attributes may be skipped, as may details of unused attributes in partially used classes.

For legacy data present in fixed-width files, non-relational COBOL files, or culled from “report scraping”, Meta_Attributes would have to record the **offset** of individual columns (for fixed-width files) and **number of occurrences** (for non-first-normal data).

For classes in staging and presentation areas, the **update frequency** (in a time unit, preferably hours) should be recorded, as well as **update schedule** (one of: regularly scheduled, when needed). Fact tables may change daily or even more frequently, while certain dimension tables (notably Clinical Concept tables) generally need to change only when updated by the controlled-terminology provider.

16.4.1.3

Data Stores

This table contains one row for each data source that will be used for the warehouse’s input data. The columns are:

- **Data Store ID** (machine generated surrogate primary key), **Source Name**, **Description** and **Store Type** (one of: Data Source, Staging Area, Presentation Area.) Data Store ID will be used as a foreign key in Meta_Classes, since every class of data must be located within a particular store.

- **Format:** This applies to data sources only. Source data may not be relational: it may be based on spreadsheets, delimited flat files, fixed-width-column flat files, Word documents, output from a black-box mainframe system, and so on. You will need to create an enumeration for the various types of formats that you encounter.
- **Schema Diagram:** For RDBMS stores, several tools (notably Visio) allow you to reverse-engineer a schema diagram, and this can be stored as a binary file, which yields a picture worth several thousand words. For legacy systems, you may have to create your own from a description of the system, its source code, and its input/output.
- **Schema Definition Information:** Kimball and Caserta recommend that, for the staging and presentation areas (both of which will use a DBMS), the complete database details should be recorded for ready reference. This information turns out to be useful if performance needs to be tuned later. One can store this simply as data-definition-language SQL in a large text field, or divide information into a finer grain. Information in the tables/columns and views would be redundant with structured information already in the Meta_Classes and Meta_Attributes tables; additional information would include details of indexes, database partitions and security settings for various roles and users.
- **Ownership Information** (Which department/s own the information), **frequency of update, Legal Limitations** on the store's use (which will determine who sees the data associated with it).
- **Access Information:** For data sources only. This technical metadata includes the means of accessing this source electronically – e.g., with password-based ftp from a particular site, at a particular time of day if appropriate. It should be in a form that is directly executable or interpretable by your data-extraction programs, and may need to be divided into multiple fields.

16.4.1.4

Constraints

This applies primarily to data sources. In a large organization, it is unlikely that any single individual will understand the working of every source system, and so, as these systems are studied by different team members, information on data constraints must be recorded as such constraints are discovered.

Ultimately, most or all this information – default values, non-null checks, range and enumeration checks, referential integrity, cross-field validation – would be stored in structured form, as described in the EAV/CR chapter. However, because of the putative nature of constraints when first identified, it is preferable to record these against individual classes initially as narrative text.

The structure of this table is as follows:

- **Constraint ID** (machine-generated primary key)
- **Constraint Type** (one of: Cross-table, Intra-table-cross-field, Single-field)
- **Class ID:** if Intra-table, the table to which it refers. For referential integrity checks, this is the foreign-key table.

- **Attribute ID:** If Single-field, refers to the Attribute to which the constraint applies.
- **Constraint Description:** Narrative Description of the constraint
- **How is the Constraint Enforced in Source System?** By schema DDL, by shared enumeration table, or application code.
- **Constraint Name:** A short name, used to refer to the constraint in output (e.g., when running checks against a table to see if constraints are violated).
- **Date Created, Date Last Modified, Whether Encoded in Structured Form** (Boolean), **Date Encoded, Person Responsible for Encoding.** Self-explanatory.

16.4.2

Metadata Generated During the ETL Process

16.4.2.1

Quality Indicators and Error Metadata

These are generated both during the initial Data Profiling effort, as well as continually during ETL. Larry English's book⁹ is an extremely readable, if occasionally repetitious, guide to implementing data quality. The metadata related to this consists of both summary statistics as well as detail information that assist quality monitoring.

The initial profiling effort detects the extent of error and determines the nature of remedial measures. If, however, the same kinds of errors keep repeating, this indicates problems with production-system design, process design or even specific individuals responsible for data creation. Therefore, when *screens* (programs designed to catch one or more types of errors in a particular data source) are run, it is important, when analyzing the reported errors in totality, to record associated variables: the source system, the table/class screened, the attribute involved, the date (and time) the source data was created, the screen that was run, the constraint that was violated (*error type*), the user who created the data or last modified it.

The schema that reports errors naturally fits into a star or snowflake schema where the fact table (that contains row-by-row error information – sometimes a single row may have multiple errors) – is surrounded by dimension tables that record the information just described. (The Data Store, Meta_Classes and Meta_Attributes tables serve as dimension tables, since these are three of the variables. Note, however, that particular diagnostics may refer to cross-variable checks, so there will have to be an outer join between the error table and Meta_Attributes.)

Occasionally there may be a problem with the screen rather than the data itself: e.g., if the screen operates on flat text files rather than RDBMS contents, and the input was changed by the addition or deletion of a field and the software team omitted to update the screen, a sudden cascade of errors may be reported when field #4 (which is now field #5) fails type-checking.

Kimball and Caserta recommend computation of an error score based on a subjective weight (from 1 to 100) assigned to each individual type of error, with totals by error type and even an overall score for a table/class that sums the error-type scores, but I'm skeptical of the value and meaningfulness of the last statistic.

16.4.2.2

Dependency (Lineage) Metadata

The nature of dependencies between source and target columns is many-to-many, as I've stated earlier. Thus, a particular column may be computed from several columns, and a given column may be used in several different computations.

However, dependencies do not involve columns (and their containing tables) alone. Data is processed by *jobs* (programs), which in turn generate output that may be processed further. Therefore the representation must be more generalized. This implies that *individual jobs should also be represented in metadata*.

Kimball and Caserta recommend the following basic structure for a Jobs table: Job ID, Name, purpose/description, job category (one of: extraction, transformation, load), program code (text/BLOB), program code language (e.g., SQL, Perl, C++), input parameters (this may be split into a separate many-to-one table) and current version number.

The general way to represent dependency information in a database is through three columns: independent object, dependent object, nature of dependency. This structure allows the standard database operation called transitive closure: determining the chain of data and jobs responsible for producing a given output. Thus, a job uses certain tables/classes as input and creates particular tables as output. In turn, the individual input tables are created by previous jobs, so that those jobs must complete before the current job commences. Of course, a table contains columns, column-job dependency must also be captured.

I've stated that most GUI-based tools that support creation of transformation data flows generate executable objects. However, the ease with which you are able to get lineage information out of them, in a form that you can import into a dependency table, varies widely. (SQL Server 2008 Integration Services, for example, only provides you with an example as to how to implement lineage tracking yourself if you need it.)

16.4.2.3

Process Performance Metadata

This data is primarily statistical in nature. Because of the experimental process of determining the optimal way to transform data, performance must be recorded. Kimball and Caserta recommend that the following information be recorded on every job in the ETL process: Job ID, number of rows processed, and performance parameters related to the job. The performance parameters include start date/time, end date-time, mode of termination (success/failure), elapsed time, error rate, rejection rate, etc. Such data provides information on the steps' accuracy, efficiency and success.

16.4.3

Front-End Metadata

The Meta_Classes and Meta_Attributes tables, of course, store details on front-end fact and dimension tables as well. In addition, one must store standard query and report

definitions. The Reporting schema described in the previous chapter can be utilized here (including metadata supporting access security through Roles and Users). User activity (starting with a history of logons) may need to be tracked – notably in clinical situations for access to identifiable data.

Usage Metadata: These are statistics on how often individual queries have run, or how often individual aggregates are consulted. Certain commercial front-end tools allow collection of such statistics. Resource usage is also important to document in situations where the warehouse acts as a cost center with a charge-back mechanism to individual departments based on usage.

End-user documentation and training aids are an important component of metadata. These may be delivered by mechanisms such as wikis and help-generation utilities, as stated earlier.

16.5

Case Study: The I2B2 Star Schema

The I2B2 (Informatics for Integrating Biology and the Bedside) effort at Partners Healthcare System (Principal Investigator: Isaac Kohane) has multiple objectives, many of which are related to delivery of tools for researchers. Among these is the I2B2 star schema, designed primarily by Shawn Murphy, whose earlier work on clinical data query has been described in an educative series of papers.¹⁰⁻¹³

The I2B2 data-mart schema is described in.¹⁴ Here, a central Observations fact table is surrounded by dimension tables for Patient, Provider, Encounter (Visit) and Clinical Concept. (The primary key of the fact table is a composite of the four IDs related to these dimensions, plus a Start Date/time, indicating the date/time of the Observation. Remember that the same observation can be recorded multiple times for the same patient.)

Like all star schemas, I2B2 is an optimization for a particular purpose – here, to identify anonymized patients who match a particular complex Boolean combination of clinical findings. The idea is that investigators, before planning a research project, need to perform a feasibility analysis in terms of the number of patients in the local healthcare system matching a particular clinical/disease profile. Only if the cohort identified is large enough is it worth making the effort to actually write up the proposal and ask for funding.

The Patient and Encounter dimensions are anonymized. The idea is that researchers should be able to ask research related questions of the data without the local Institutional Review Board worrying about compromising patient confidentiality. If the proposal is approved and funding obtained, a separate linkage between surrogate Patient IDs and the actual (identifiable) Patient IDs and demographic data can be utilized to identify real patients and contact them regarding participation in the study.

Clinical facts have different data types: numbers, short strings, long text (e.g., clinical text) and XML (clinical text marked up by NLP software). The I2B2 fact table uses an EAV design with multiple data-type-specific columns in a single table, with an indicator column indicating the data type of the current column. (Thus, if the data type for a

particular row was Numeric, the numeric value column would be consulted.) In addition to the value column, additional (optional) columns provide extra information: end date/time (if applicable), units of measurement, modifiers (e.g., route of administration, for drugs).

All tables (fact+dimension) contain lineage information (whose importance has been described earlier): the source from where it originated, and the date/times a particular row was updated in the source system, are downloaded from the source, and imported into the schema.

16.5.1

Security Architecture

Among the interesting aspects of the I2B2 architecture is the security layer. As already stated, access to identifiable Patient Demographic fields is restricted, as are many details of Encounter information: for example, even if Patient ID were hidden, but we knew the exact date/time and location of a given encounter, and had access to the encounter data, we could determine the Patient involved.

Even though identifiable structured data (e.g., linked to the patient and encounter) are off-limits to the vast majority of users, the clinical text may yield inadvertent clues to the patient identity, because it may include part or all of the patient's name, city, etc. Programs that claim to cleanse such data of identifiable information don't do a perfect job: many surnames are common words in English – e.g., Sands, Brown – and clinical text does not always supply cues, such as proper-casing of proper nouns or sentence structure, that assist parsing. Consequently, encryption is a safer, non-destructive mechanism for securing such data.

Different users have varying read permissions to the data, from least level ("obfuscated", or ability to see only aggregate numbers with random data added), aggregate data without obfuscation, detail of all columns except encrypted columns, detail rows with encrypted columns also, and all data (including the identifiable data stored separately).

16.6

Conclusions

Warehouse initiatives, while not as risky as Enterprise Resource Planning projects, still have a high risk of failure. Along with organizational dysfunction and inadequate hardware resources (one effort I knew used an extraction process against a *live* MUMPS database that would have produced daily extracts only if a day had 720 hours), the commonest cause of failure in the projects I've witnessed or heard of is the absence of a detailed metadata strategy.

Documentation as a component of metadata cannot be an afterthought because the ETL process has a large component of trial and error: one may need to try several approaches (flat files and UNIX-style filters, RDBMS queries or a mixture) and benchmark them to see which performs best. Further, there are so many transformations involved from source to

final product that the only way to maintain lineage information is to operate in the fashion of the mythological hero Theseus, who traced his path through the Minoan Labyrinth by unraveling a ball of string. In other words, you have to document your steps as you work: you cannot afford to miss a step: if you do, you have to start all over again.

While many commercial tools do a satisfactory job on the lineage issue, the worst thing your team can do here is to buy a metadata tool without having first evaluated it thoroughly and tried it out in a realistic test scenario. I know of one team that purchased a tool without any evaluation, had difficulties installing it, and then proceeded with an ETL effort while the tool was still shelf-ware: the end-result was completely bereft of lineage information.

Given the significant risk involved, data warehousing is one area where both technical and political skills must be employed maximally. The body of knowledge has grown rapidly, the technology is advancing steadily, and it pays to be both suitably humble as well as be prepared to continually acquire new information. Apart from the Kimball books and, several Kimball group articles, originally written in industry publications, are also available freely from the Kimball Group's Web site (www.ralphkimball.com/) at no cost after initial registration. Another invaluable resource is Larry Greenfield's website, the Data Warehouse Information Center (www.dwinfocenter.org), which is less technically oriented, but has numerous articles dealing with warehousing challenges (including political ones).

References

1. Kimball R, Caserta J. *The Data Warehouse ETL Toolkit*. New York: Wiley; 2008.
2. Kimball R. *The Data Warehousing Toolkit*. New York: Wiley; 1997.
3. Inmon W, Rudin K, Buss C, Sousa R. *Data Warehouse Performance*. New York: Wiley; 1998.
4. Vetterli T, Vaduva A, Staudt M. Metadata standards for data warehousing: open information model vs. common warehouse metamodel. *ACM SIGMOD Rec*. 2000;29(3):68-75.
5. Object Management Group. Common warehouse metamodel. 2010 [cited 10/3/10]; Available from: <http://www.omg.org/technology/cwm>.
6. Kimball Group. Design tips. 2010 [cited 11/2/10]; Available from: <http://www.rkimball.com/html/designtips.html>.
7. Mundy J, Thornthwaite W, Kimball R. *The Microsoft® Data Warehouse Toolkit: With SQL Server™ 2005 and the Microsoft® Business Intelligence Toolset*. New York: Wiley; 2006.
8. Leuf B, Cunningham W. *The Wiki Way. Quick collaboration on the Web*. Reading: Addison-Wesley; 2001.
9. English L. *Improving Data Warehouse and Business Information Quality: Methods for reducing costs and increasing profits*. New York: Wiley; 1999.
10. Murphy SN, Morgan MM, Barnett GO, Chueh HC. Optimizing healthcare research data warehouse design through past COSTAR query analysis. *Proc AMIA Symp*. 1999:892-896.
11. Murphy SN, Barnett GO, Chueh HC. Visual query tool for finding patient cohorts from a clinical data warehouse of the partners HealthCare system. *Proc AMIA Symp*. 2000:1174.
12. Murphy SN, Chueh HC. A security architecture for query tools used to access large biomedical databases. *Proc AMIA Symp*. 2002:552-556.
13. Murphy SN, Gainer V, Chueh HC. A visual interface designed for novice users to find research patient cohorts in a large biomedical database. *AMIA Annu Symp Proc*. 2003:489-493.
14. Partners Healthcare. I2B2 Data Mart Design document. 2010 [cited 10/1/10]; Available from: https://www.i2b2.org/software/projects/datarepo/CRC_Design_15.pdf.

The arrival of standardization within a field is a sign of its maturity. An entertaining and educative account of the history of standardization can be found in the widely reproduced article by Nesmith¹: it began with weights and measures, and spread into areas such as the gauge of railroad tracks and standardized dimensions of hardware units. In computing, the first (de facto) standard was possibly in the area of high-level programming languages (FORTRAN) in 1957, though it was adopted on non-IBM hardware only in the early sixties. The ASCII character-encoding scheme (=American Standard Code for Information Interchange) arrived in 1960.

There are several challenges in standardization.

- One must pick the right time to standardize and the right technology to standardize on. While the choice of door size dimensions or voltage on electrical grids is somewhat arbitrary, backing the wrong technology results in the proposed standard becoming widely ignored. This happened with the CODASYL network-based database model after commercial RDBMSs were introduced; some of the proposed standards for clinical decision support may well end up with a similar fate, as I've stated in an earlier chapter.
- One must ensure that the standard is sufficiently comprehensive. A very limited standard that omits to address the hard issues gives individual vendors legitimate reasons to extend it to meet real needs, but in mutually incompatible ways. SQL was babelized this way; the Health Level Seven (HL7) version 2.x standard is the most well-known example of this situation in the biomedical field.
- An issue that has nothing to do with a standard itself, but everything to do with individuals who don't understand its scope, arises when one applies a standard originally devised for one purpose to another for which it is ill-suited. I'll discuss an example of this in more detail.

As far as metadata standards are concerned, I aim to persuade you in this chapter that there can be no single truly "universal" metadata standard that meets the requirements of comprehensiveness and human usability. This is because of the various uses that metadata serves: the concepts that are embodied in a particular application of metadata design – for example, user interface design with validation – must be reflected in the data structures that are created to represent the problem that is being addressed.

In other words, *a human-usable metadata standard must enable one to address a problem with the appropriate degree of abstraction*. It should not be so concrete and over-specified as to act as a hindrance to further evolution, but it should not be so far removed from the actual problem to be solved that it becomes a struggle to use it. This requirement rules out metadata standards such as XML Metadata Interchange (XMI), which is the XML-based, non-visual counterpart of the Unified Modeling Language (UML). While indeed versatile and capable of representing anything that can be represented in UML, it is really intended for programs: it is far too abstract for human use for an application such as modeling a biological sample repository.

A second lesson that I hope to convey is that metadata cannot stand in isolation: it defines and describes data. *Therefore, when one defines a metadata model to solve a specific problem, it must be accompanied by a corresponding data model*. The data model cannot necessarily be predicted or derived from the metadata model – for example, I’ve shown in earlier chapters that the actual representation of a class of data can vary from a physical table to something simulated using EAV – but the two must be linked. “Linking” simply means that there is a way to navigate directly from the metadata sub-schema to the data sub-schema and vice versa. Thus, in the EAV-based designs of EMRs and CSDMSs, the data model references the attributes (parameters) and their higher-level groupings, such as Forms, and in EAV/CR systems where certain classes are represented as physical tables, the Class and Attribute definitions within the data model directly allow you to reference the tables and columns where the classes’ data is stored.

With these caveats I’ll discuss some of the problem-specific standards that are being employed. This chapter is divided into two parts: the first deals with non-HL7-based standards (which are reasonably non-controversial), and the second with HL7 version 3 (HL7v3) related standards, which have triggered religious wars. This is despite the fact that HL7v3 has the potential to be arguably the most important standard in all of biomedicine.

17.1

Self-Describing Attribute-Value Pairs

Many units of data contain accompanying metadata that provides extra information that helps to interpret the data. Such metadata has significantly different structure and purpose from the kind of metadata that has been this book’s focus. In terms of its structure, such metadata is represented as attribute-value pairs where the list of possible attributes is constrained by set membership, and the permissible values for a given attribute may also be constrained. The metadata is not necessarily human-readable: it may be compressed into a binary format.

Thus, most image interchange formats, such as JPEG or the DICOM standard used for medical images, involve metadata as well. For example, JPEG files have self-describing metadata on the extent of compression, original image dimensions, color/grayscale information, and so on. In the case of DICOM, there are literally more than a thousand possible metadata fields, many of which are vendor-defined. This metadata is automatically created by

the software that creates the images, though some of it (such as the degree of compression/image quality) is specified by the user.

Web pages can also contain metadata: I'm mentioning this only for the sake of completion, since much biomedical content is served via the Web. The HTML tag for this purpose is "meta" with the attribute and value defined by the HTML attributes "name" and "content". For example:

```
<meta name="Description" content="Controlled Terminologies in Biomedicine"/>
```

Among the permissible values for the name are: description, keywords, author, copyright, and the set of tags that are derived from the Dublin Core,² indicated with the prefix DC (e.g., DC.Creator, DC.Publisher, DC.Date). The older, non-Dublin tags were originally intended to serve as a guide to Web search engines by providing description and keywords. However, these were thoroughly abused by Web-page creators in an arms race to improve page-ranking: they attempted to fool the naïve prevalent search engines such as Alta Vista by repeating the same word or phrase tens or even hundreds of times in the value field. Google now ignores these tags entirely: it will only use such content in order to make a decision as to whether to blacklist the page if it detects the above cheap trick.

17.2

Standards for Descriptive Metadata: ISO/IEC 11179

ISO/IEC (=International Standards Organization/ International Electro-technical Commission) 11179 refers to a standard for Metadata Registries, which are supposed to contain all of an organization's metadata. Such registries are intended to support exchange of data between systems by recording the semantics of individual data elements. I'll simply refer to it as 11179 henceforth.

11179 can be represented using a very simple subschema. It resembles a thesaurus in that there are Concepts and Data Elements: the latter are analogous to terms for a given concept. The difference is that in a thesaurus, as in all terminologies, the Concept is central, while in 11179 the Data Element is central. This is because limited validation and presentation information is associated with a Data element, in the form of a "Value Domain". Information associated with the Value domain includes the data type, minimum and maximum length (for strings), enumerated values, range, number of decimal places, the presentation format (e.g., long or short dates, percentage). Concepts can be categorized into higher-level groups ("Contexts") and Data Elements can also be similarly categorized by "Classifications".

The idea is that the same concept may be represented physically in different ways in different places there should be one data element for each instance of a concept in a given source, where the meaning of "Source" is left to the implementer: it may be a single database or even a single table (or file, if we consider pre-relational systems). Obviously, a concept may not be represented in every single source. Issues such as referential integrity are not explicitly stated: in 11179, a primary key and its corresponding foreign keys are the same data element.

Thus far, the 11179 meta-model seems fairly straightforward. However, it has several limitations.

- In terminologies, a textual definition is associated with a concept. In 11179, it is (incorrectly) associated with the Data Element.
- In Chapter 5 I've stated that single-element validation does not suffice. Cross-element validation cannot be defined in 11179. Cross-element validation constraints (or formulas that specify how a particular element is computed from others) are some of the important "business rules" in both transactional as well as data warehouse systems, and not just in the biomedical domain. 11179's omission of cross-element constraints is somewhat paradoxical since 11179 is intended to support exchange between systems, which is the first step toward integration.
- In terms of terminologies, the only kind of relationship (between concept and term) that 11179 models is the synonym. Other types of relationships, such as general-specific, are not addressed. This makes it a misfit for modeling terminologies, even though one repository, the National Cancer Institute's Cancer Data Standards Repository, has tried to use 11179 as the basis for modeling both a terminology as well as case report forms. The problems with this approach have been discussed in Nadkarni and Brandt.³ The Extended Metadata Registry (XMDR) consortium,⁴ aims to extend the specification to deal robustly with terminology issues: however, the group does not appear to be very active currently. (Most of the site's Web pages are of 2009 vintage.)

While public-sector organizations (the US Government, the United Nations) apparently use 11179, the question is to what uses the metadata within these registries has been put. Certainly, 11179's penetration in the commercial data-integration market is now zero: Oracle, the solitary vendor that supported it, does not do so any more.

17.3

Technical Metadata Standards: CDISC

The Clinical Data Interchange Standards Consortium (CDISC), despite its broad name, is actually concerned only with interchange of clinical data generated in the conduct of clinical trials (with emphasis on trials regulated by the FDA). It has several sets of interchange standards, which are XML-based: the two most important ones address the interchange of clinical trials metadata – definitions of protocols, events, case report forms, intermediate-level groups and individual parameters, which collectively constitute the Operational Data Model (ODM) – and the data that is represented according to the ODM.

The meta-model is quite comprehensive: it recognizes, for example, that text such as captions and definitions (as well as the descriptive phrases associated with the individual choices in an enumeration) can be represented in different languages, not just English. Many of its components are defined in terms of a higher-level XML element called *TranslatedText*, which consists of a Unicode string accompanied by a language tag (`xml:lang`) that identifies the language of the string.

The ODM also makes a partial attempt to address calculated parameters and cross-element-based validation. It uses string expressions whose programming language is specified: it is assumed either that an implementation of that language exists locally (e.g., Oracle PL/SQL), or that someone with appropriate expertise can manually translate those expressions into the language that is available.

The CDISC specification is quite easy to understand and well-documented. There are a few aspects that it does not currently model – for example, dependencies within data elements within a CRF. However, the specification is being continually enhanced, and the committee is responsive to suggestions: I would expect it to be a matter of time before some of the lacunae are filled in.

17.4

Interchange Standards for Gene Expression and Proteomics

The gene expression community uses a standard called MAGE-OM (MicroArray Gene Expression Object Model), which is based on a UML design that models the gene expression domain; MAGE-OM forms the basis of an XML-based interchange format – MAGE-ML (=Markup Language). MAGE-ML has a complex enough structure that MAGE-ML documents are intended to be created through software rather than directly by humans.

The URL <http://www.mged.org/Workgroups/MAGE/introduction.html>, which is maintained by the European Bioinformatics Institute, provides details on the specification. Most of the pages on this site are dated 2006 or earlier: the specification has stabilized since that time.

Gene expression involves the use of a cell's DNA to synthesize messenger RNA (mRNA), much of which is then used as a template for protein synthesis. In a gene expression experiment, one is interested in measuring the changes in the levels of mRNA in a tissue after an experimental intervention (e.g., after administration of a drug to an animal), by comparison with a control experiment (without the drug). Every single gene in the DNA is not activated by a given environmental stimulus: only some genes are, while the activity of other genes may actually decrease. mRNA itself is chemically unstable, so that rather than measuring it directly, it is used to synthesize "cloned DNA" (cDNA) using the viral enzyme reverse transcriptase. After being tagged with a fluorescent dye, the cDNA is measured semi-quantitatively by measuring the fluorescence signal after binding (hybridization) to a grid (microarray) containing thousands of individual genes or gene fragments ("probes") that are attached in specific positions to a solid surface (e.g., a silicon chip).

There are numerous factors that can affect gene expression. Simply taking the same tissue sample from an animal at different times of day, for example, will show variation in the expression of different genes, and animals put in cages with other animals will show different results from isolated animals), and so animals. Therefore all conditions in the control and test experiment must be identical except the intervention that is to be studied. Even within a single experiment, replicates are used to get an estimate of the variation due to sampling or hybridization; thus, the same gene fragment may be present in more than one location of the array. Gene expression itself follows a time course after the

introduction of the experimental factor – for example, some early-activated gene products may activate other genes, whose expression shows a delayed rise – so that time is an important factor to consider in any experimental design.

The interpretation of experimental results relies on methods of dimension reduction. While the changes in individual genes may be reported, the presence of several thousand genes makes it hard to see the forest for the trees. Therefore the genes are typically clustered into functional families (using a hierarchically structured information source such as Gene Ontology).

One of the problems that bedeviled the gene expression field is that experiments reported by one group were often not reproducible by another group because the experimental conditions had not been documented in the requisite detail. A consortium of researchers eventually determined the minimum information about a Microarray Experiment (MIAME) necessary to allow reproducibility. This information it turns out to be quite voluminous, and MAGE-OM – a representation of the minimal information set – includes several kinds of data.

1. The design of the array (e.g., what gene fragment is used where, what technology is used, manufacturer information).
2. The experiment design, where one or more experimental factors (e.g., time or drug concentration) are varied, the steps involved in processing, and the materials used in each processing step.
3. The details of the software and steps used for signal processing and analysis.
4. The actual experimental results.

The first three categories may be regarded as metadata in that they help to interpret the fourth.

17.5

Proteomics Standards

The motivation behind proteomics is that the quantification of mRNA does not always predict the extent of protein synthesis reliably: apart from the fact that mRNA is not always translated into protein, the production of protein is variable, depending on numerous physiological factors, as well as the parent gene itself. Further, proteins are often chemically modified after their initial synthesis – e.g., by adding carbohydrate, lipid, or phosphate – and it is modified forms that are biologically active. Some proteins will form chemical complexes with others. Therefore it is often necessary not only to measure protein quantity but to characterize the protein structure in detail.

The methods of characterizing proteins in mixtures are based on a separation process (such as liquid chromatography or electrophoresis) followed by detection. Recent high-throughput methods involve the use of mass spectrometry, which detects electrically charged (ionized) proteins, with the ionization being created by a laser beam or application of a high voltage to disperse a liquid into an aerosol (electrospray).

Proteomics data interchange standards are also XML-based. The standards body for Proteomic standards is HUPO (Human Proteomics Organization). Similar to MIAME above is the idea of MIAPE (minimum information about a proteomics experiment), and the analog of MAGE-OM is FuGE-OM (Functional Genomics Experiment).⁵ There is also a standard for exchanging mass spectrometry data, mzML⁶ and one for Molecular Interactions (MI XML). These standards are not intended to dictate the structure of a database that would store proteomics data.

17.6

Metadata for Medical Observations: OpenEHR Archetypes

OpenEHR⁷ is a proposed open-source standard that is aimed at addressing the implementation of a vendor-independent person-centered health record. The idea is that the concepts commonly used in clinical medicine must be represented in a rigorous and fully specified way that is also computable. Their definition through consensus opinion facilitates their reuse. This full specification of a concept is called an *archetype*.

Thus, the archetype for blood pressure recording is defined by all of the following:

- The value and the units of measurement.
- The type of reading: e.g., systolic, diastolic, mean arterial.
- The body site where the reading was taken – e.g., left upper arm.
- The position of the body: standing, sitting, lying, tilted
- The instrument used: mercury or aneroid.
- How the reading was determined – e.g., whether the criterion for recording diastolic blood pressure used abrupt softening of the pulse sounds as heard through the stethoscope, or absence of the pulse sounds.
- The time of day when it was recorded. If measured at a clinic, the time elapsed after the patient arrived at the clinic. It is known that, with a busy commute, blood pressure is slightly higher if measured immediately after the patient arrives: it comes down after the patient has had time to settle.

It turns out that for clinical studies, the concept of archetypes is critical, even if clinical researchers are not familiar with this informatics term, and even though the matter of rigor in definition preceded OpenEHR by at least four decades. When two drugs are being compared, for example, and the differences in effect are of the order of a few mm Hg, all of the above factors must be noted in the study protocol and controlled. The OpenEHR model also proposes that the individual aspects of an archetype be associated with concepts from a standard controlled clinical vocabulary such as SNOMED CT. As of this writing, the OpenEHR repository includes about 150 archetypes. I'm not sure that these are fully annotated with SNOMED CT concepts: however, the effort at defining archetypes will also help in identifying lacunae in SNOMED CT coverage.

OpenEHR is promulgated by the non-profit OpenEHR foundation, and the model is being tried in individual institutions in Britain, Australia and individual vendors in the

USA, Brazil and parts of Europe, though it is unclear whether it is being employed in production settings. One practical problem with the storage model advocated by OpenEHR is that the entire record for the patient be stored in a single large chunk of XML: this approach poses serious limits to scalability if deployed for a large caregiver with a large patient population, especially in the multi-user setting, where a more atomic means of storage of the different parts of the record is preferable. Refer to the previous chapter on XML, and the section on database storage of XML in particular, for why this is so. However, it is perfectly feasible as a means of storage for a *personal* health record that is controlled by the patient.

Despite the above caveats, the concept of archetypes (irrespective of how they are actually stored in a database) is a valuable one. Further, the OpenEHR initiative deserves support, if only as a counterweight that serves to keep the large EMR vendors honest and responsive to their customers.

17.7

The HL7 Version 3 Reference Information Model (RIM)

I've stated earlier that HL7v3 was originally intended to fix all of the problems resulting from the under-specification of the v2.x standard. It has been under development since 1996. HL7 v3 is based on a metadata model called the Reference Information Model (RIM). A good summary of the model's highlights is described in Benson.⁸ To quote this author, "HL7 v3 was intended to handle most if not healthcare communications in an unambiguous way, using a relatively small set of constructs, which can be learnt relatively easily." The claim about ease of learning is highly controversial: the use of a small set of constructs is both strength and weakness, as I'll discuss later.

17.7.1

Overview

I'll now present the HL7 v3 RIM in a nutshell. At the core of the RIM are three top-level classes and three association classes (which inter-relate the former). The top-level classes are:

- **Entity:** living or nonliving individuals, including groups, involved in healthcare, which have an independent existence. Example of entities are Person, Organization, materials such as Medications, micro-organisms, medical devices, and so on.
- **Role:** a function or responsibility of an Entity. Thus, a Person can have the role of Employee, Patient, Caregiver, etc., an organization the role of Employer, Insurer or Health Maintenance Organization, a microorganism can have the role of Pathogen. Obviously, a single individual may have more than one role.
- **Act:** The actions/activities in which entities with roles are involved. Examples include clinical encounters, financial transactions, clinical observations, diagnostic tests, prescribing medications, surgical procedures.

Other classes are derived from the above (e.g., Person and Organization are derived from Entity and Patient from Role). As in standard object-oriented inheritance, the top-level classes contain a set of attributes (for example, Roles and Entities have Names), and the derived classes augment these with attributes of their own.

One of the attributes of Acts is “*mood*”: this word is confusing, but it means that a particular act has happened – or not happened as yet but is requested, proposed, promised, or intended. Acts can also have a *status* – e.g., completed, aborted, canceled, suspended, active and so on. Acts have an “Activity Time” – analogous to *transaction time* in temporal databases, which is the time when the action was recorded in the system – while both Acts and Roles have an Effective Time. For Acts, Effective Time is the time (or period) when the act is supposed to happen (which may be in the future). For Roles, it is the time interval when the Role is valid: e.g., for a Person in the role of Care Provider whose contract is renewed yearly, the valid time ranges between the date when the person was hired and the (future) last day of the contract.

The association classes are:

- **Participation:** The involvement of a Role in an Act, either as performer, or as the target. For example, a surgery involves the involvement of one or more caregivers in the role of surgeon, assistant, etc. and a patient who is the recipient.
- **ActRelationship:** Describes the nature of the association between a source act and a target act. Actions in healthcare are performed consequent or subsequent to other acts. The association is the *relationship type*. For example, clinical note creation *documents* a patient encounter, a diagnostic blood draw *fulfills* a clinician’s order, and so on.
- **RoleLink:** Relationships between roles – e.g., between family members or members of a medical team.

The XML that is the proposed basis for interchange is derived directly from the Information Model. Specifically, the attributes of the classes in the RIM become XML tags.

HL7 v3 supports (indeed, requires) the use of controlled vocabularies such as SNOMED CT and LOINC. The names of clinical observations, laboratory tests and investigations utilize coded values, which consist of a code, an optional descriptive phrase (“display name”) and the source vocabulary (“coding system”) that is used for encoding. The last is specified as an Object Identifier (OID), using a Dewey-decimal-type notation rather than a symbolic name. (For example, HL7’s OID is 2.16.840.1.113883.)

17.7.1.1

Extensibility

The RIM is intended to serve as the basis of domain-specific models for specialized areas of biomedicine by allowing new classes and attributes to be derived from those existing in the RIM. Understandably, the semantics of these models should not conflict with those already specified in the RIM, though catching non-obvious semantic violations is a hard problem. Presumably, such models would have to be implemented through HL7 committees, or by groups working closely with HL7.

17.7.2

Limitations and Controversies

The RIM's UML Class diagram is not particularly complicated: it consists of about 40 classes overall (including the six core classes summarized above.) In terms of modeling elegance for a domain as vast as healthcare, this would be considered a *tour de force*. Despite being in development for a decade and a half, however, HL7v3's market penetration in EMR applications is estimated to be of the order of 1% – chiefly new applications⁹. The HL7 voting body (which has numerous vendors in its membership) has consistently refused to officially endorse version 3 as the replacement for version 2.x. One must therefore consider possible reasons for this reluctance beside vendor cussedness.

17.7.2.1

Compatibility and Stability Issues

I've stated in Chapter 1 that HL7v3 is revolutionary, not evolutionary. In other words, there is no backward compatibility with 2.x: applications using 2.x must have their heart and guts ripped out and rewritten completely, with no straightforward migration path. Given HL7's under-specification and the lack of a data model, this was probably inevitable, but it does not diminish the pain of re-engineering.

Over time, HL7v3 has undergone repeated revisions: the current version number exceeds 100. (For example, there were originally 100 classes in the RIM, but the work of Schadow et al.¹⁰ reduced them drastically.) In fairness, the problem to be addressed is very hard, but the frequency of revisions also means that volunteers who are prepared to risk building production applications with moving-target technology become hard to find. Further, the process of development, the end results and the accompanying documentation have their critics, notably Prof. Barry Smith, whose blog is HL7 Watch, at <http://hl7-watch.blogspot.com/>. Many, though not all, of Smith's criticisms are legitimate: rather than try to steal his thunder, I'll simply direct you to his highly readable posts, which are accessible all the way back to 2005.

17.7.2.2

Complexity and Limitations of the XML-Based Messaging Standard

The HL7-XML message syntax has been criticized as extremely verbose and needlessly complicated: this point has been raised by several parties, including the Gartner Group.¹¹ Aerts,⁹ in a fervent plea that HL7-XML should not be used by CDISC as a format for submitting clinical data (as was proposed at one time), asserts that this is because the XML syntax is derived computationally from the RIM's UML, resulting in "spaghetti XML". However, in other cases where XML has been derived this way – notably the Gene Expression Group's MAGE-ML, mentioned above – the result has not attracted this much adverse publicity.

The difference is that MAGE-OM contains numerous classes that very accurately reflect the science of gene expression experiments. The derived XML, while still having a complex structure that requires software to compose, is still more or less readable to someone working in the gene expression area. HL7v3 messages, by contrast, are much more abstract. This is because they are intended to address the hard problem where the sending and receiving system know almost nothing about each other. Therefore, metadata defining what is being communicated – e.g., the coding system being used – forms a large part of the message. In fact, the metadata may be much more than the actual data being communicated, and this situation repeats for almost every data item. Also, some ancillary information is constant (e.g., if a set of laboratory results is being reported, the performing laboratory need be identified just once). In many cases, the ancillary information is much less important or even unimportant – as for intra-institutional data transfer.

Like many inter-system communication scenarios (such as SOAP used for Web Services, and CORBA in the pre-Web age), the HL7-XML message syntax is designed for the worst-case scenario. This makes for excessive verbosity when such a scenario does not apply, and a simpler, more lightweight approach might be preferable.

17.7.2.3

Possible Solutions

Ann Wrightson¹² identifies a potential solution: to eliminate the repetition, it is necessary to devise a means of specifying metadata that describes the *entire* message stream. The data can now be interpreted using the above metadata, making it more compact: it could also allow the substitution of user-friendly aliases for codes. Such an approach is currently being worked on by HL7's Implementation Technology Specification committee, but it's anyone's guess when the standard will change.

In the field of programming languages, developer productivity has been enhanced through the use of *domain-specific languages* (DSLs),¹³ which are programming languages that are extremely capable for the task they are intended to address, but stripped-down in most other respects. Examples of DSLs are SQL for database queries and R for statistical programming. SQL can achieve in a single statement complex data retrieval would otherwise take several pages of code from first principles, and in R most statistical operations are primitives in the language.

XML has been used as the basis for DSLs, such as MathML for expressing mathematical equations. Currently, HL7-XML is far too general purpose in nature (and consequently, far too vast) to be composed in a straightforward fashion, even by software. Distinct sub-languages for special tasks, such as communication of clinical observations or lab tests, may improve its usability. An example of a sub-language developed partly in collaboration with HL7 is the CDISC Lab-XML standard (<http://www.cdisc.org/lab>), used for transmitting laboratory test data. Interestingly, data that conforms to this standard need not use XML: it can be transmitted as delimited text files. (CDISC seems to have correctly realized that ease of use is more important to promulgating the use of a standard than creating jobs for XML programmers.)

The difference between working directly with the HL7v3 RIM versus working with DSLs is like the difference between quantum mechanics and classical physics. While quantum mechanics is “fundamental” in that its predictions reduce to those of classical physics for large systems, classical physics is dramatically easier to work with when you are simulating the stresses on an airplane wing or automobile chassis.

Similarly, every concept in healthcare might *possibly* be related to the RIM if one spends enough effort using the RIM and existing biomedical controlled vocabularies. However, even within HL7 there is skepticism that this is currently possible. In any case, rather than have to wrestle with RIM concepts and attributes at every stage, it is much simpler and far more productive to define higher-level abstractions in terms of RIM concepts (or not – see Dolin et al. below), and then to work with the abstractions only.

17.7.3

The HL7 Clinical Document Architecture (CDA)

The clinical document architecture¹⁴ is a means of specifying clinical text such as a discharge summary in semi-structured form, with embedded XML markup. It provides a migration path for legacy data. At the most basic level, the markup can be minimal – little more than the equivalent of an HTML page, with a header and body, with the body content being minimally marked up or not marked up at all. At the other extreme, the body is divided into sections: within each section, units of text can be interleaved with individual observations, which can be annotated in great detail using the HL7 RIM attributes and links to controlled-vocabulary items. The authors of the CDA specification take care to emphasize that CDA is only an interchange format, and not an authoring specification or a specification for internal data storage.

Interestingly, it is permitted to employ additional attributes not defined in the RIM at local sites if needed, provided these do not conflict with RIM-derived tags. In principle, therefore, one could combine the standard CDA elements with, for example, markup related to natural-language-processing as described in chapters 3 and 12.

The creators of the HL7 CDA specification foresee the creation of templates that will apply to specialized medical sub-domains – e.g., the reporting of a standard chest X-Ray examination. However, they would understandably prefer to delegate such a task to groups that have the requisite domain expertise.

The idea behind CDA is that, through maximal structuring *a priori*, it should be possible to do a better job at capturing and preserving clinical semantics in narrative text. However, Dolin et al., in the *official* HL7 document describing the CDA¹⁵ doubt that it is currently possible to capture semantics completely. They state, “While the framework provided by the RIM and by CDA and by the shared HL7 Clinical Statement Model are a critical component of semantic interoperability, they are not currently sufficient, particularly given the lack of a global terminology solution, and the fact that each terminology overlaps with the RIM in different ways.”

Eric Browne,¹⁶ in a document on the OpenEHR website, explains the above limitation more clearly with examples. He points out the HL7 RIM uses an inconsistent mixture of approaches at modeling clinical concepts. Some of the simpler concepts are modeled explicitly using classes and attributes, while for other concepts – specifically, those more

difficult to model – it simply references entries in controlled vocabularies. Browne suggests, again with examples, that OpenEHR archetypes, discussed earlier, may be suitable candidates as the foundation for such templates.

Because the use of CDA does not necessarily imply buying into the still-controversial HL7-XML messaging specification in its entirety – as previously stated, an incremental approach towards adoption is possible – its adoption has been somewhat more widespread than that of the rest of HL7v3.

17.8

Final Words on Standards

Throughout this book, I've discussed standards in multiple contexts. The history of software provides examples of standards that have succeeded as well as failed. Henning's article, "The Rise and Fall of CORBA",¹⁷ should be mandatory reading for even the non-technically inclined, for the clarity with which it summarizes the factors that influence standardization success and failure. I'll summarize these points here.

Henning advocates that *no standard should be approved without a reference implementation*. The reference implementation should be used to implement a few projects of *realistic* complexity. This stringent requirement ensures that the standard's designers "eat their own dog food" – a phrase believed to be coined at Microsoft, referring to using your own product for critical projects before selling it to customers, so as to ensure its robustness and fitness for its intended purpose.

Such internal evaluation also acts as a crucible for clarity – ambiguous aspects of the standard will, by confusing the implementers, be unmasked as such – as well as a brake on the addition of features that turn out to be disproportionately hard to implement compared to the value that they provide.

I should also add my own take here: forced internal evaluation can act to minimize excessive volatility in the standard, by forcing the reference implementers to experience first-hand the effort it takes to keep up with their team's own changes. Consequently, intended changes are likely to be thought through much more deeply, and it is much less likely that the standard will change with every new phase of the Moon.

Henning also points out that cooperation and trust are necessary pre-requisites for developing a quality standard: these are often lacking in large industry consortia composed of competing vendors and customers. He notes that open-source efforts, often characterized by a benevolent dictatorship of a few individuals who have a clear architectural vision, have a better track record than the more "democratic" process of an industry consortium, and also that open-source innovation is much more Darwinian, because survival of a particular approach depends on a critical mass of users.

References

1. Nesmith A. A Long, Arduous March Toward Standardization. *Smithsonian Magazine*. 1985:83.
2. Dublin Core Metadata Initiative. Dublin Core metadata element set, version 1.1: reference description. 2002 [cited 2002 Feb 23, 2002]; Available from: <http://dublincore.org/documents/dces/>.
3. Nadkarni PM, Brandt CA. The common data elements for cancer research: remarks on functions and structure. *Methods Inf Med*. 2006;45(6):594-601.
4. Extended Metadata Registry Consortium. XMDR. 2010 [cited 11/1/10]; Available from: <http://xmdr.org>.
5. Human Proteomics Organization. Functional Genomics Experiment (FuGE) Home Page 2010 [cited 11/1/10]; Available from: <http://fuge.sourceforge.net/index.php>.
6. Deutsch E. mzML: a single, unifying data format for mass spectrometer output. *Proteomics*. 2008;8(14):2776-2777.
7. OpenEHR Consortium. OpenEHR. 2010 [cited 11/2/10; Available from: www.openEHR.org.
8. Benson T. *Principles of Health Interoperability HL7 and SNOMED*. London: Springer; 2009.
9. Aerts J. Ten good reasons why an HL7-XML message is not always the best solution as a format for a CDISC standard (and especially not for submission data). 2008 [cited 9/2/10]; Available from: http://www.xml4pharma.com/HL7-XML/HL7-XML_for_CDISC_Standards.pdf.
10. Schadow G, Russler DC, Mead CN, McDonald CJ. Integrating medical information and knowledge in the HL7 RIM. *Proc AMIA Symp*. 2000:764-768.
11. Rishel W. HL7 V3 messages need a critical midcourse correction. Gartner Industry Research Publication ID Number: G0014095. 2006 Gartner Group Inc., Stamford, CT.
12. Wrightson A. Is it possible to be simple without being stupid? Exploring the Semantics of Model-driven XML. *Extreme Markup Languages 2007*; 2007; Montreal.
13. Wikipedia. Domain-specific language. 2010 [cited 10/1/10]; Available from: http://en.wikipedia.org/wiki/Domain-specific_language.
14. Dolin RH, Alschuler L, Boyer S, et al. HL7 Clinical Document Architecture, release 2. *J Am Med Inform Assoc*. 2006;13(1):30-39.
15. Health Level Seven. HL7 Clinical Document Architecture version 2.0. 2004 [cited 8/1/10]; Available from: <http://xml.coverpages.org/CDA-Release2-Unofficial.html>.
16. Browne E. Archetypes for HL7 CDA documents. 2008 [cited 9/1/10]; Available from: http://www.openehr.org/wiki/download/attachments/3440870/Archetypes_in_CDA_4.pdf.
17. Henning M. The rise and fall of CORBA. *ACM Queue*. 2006;4(5):28-34.

A

- Active server pages (ASP), 105
- Ad hoc query (AHQ)
 - advice for academic informaticians, 297
 - comprehensible output
 - data amalgamation, 351–352
 - metadata for meta-analysis, 350–351
 - parameter detailed descriptions, 351
 - definition, 294
 - drill-down approach, 320–321
 - individual attribute metadata, 321–322
 - key-phrase or partial string search, 321
 - problem constraints, 296–297
 - query representation schema
 - non-temporal criteria and output, 328–329
 - query-definition subschema, 326–328
 - temporal criteria, 329–330
 - schema volatility problem
 - loading into user interface, 295–296
 - SQL code storage, 294–295
 - selection criteria and output options, 322–324
 - sparse-data problem, 317–319
 - SQL generation
 - conceptual compound query, 331–332
 - more physical table group, 334–335
 - multiple classes in EAV/CR, 335–337
 - multiple output tables, 338–340
 - problem simplification, 330–331
 - single physical table group, 333
 - translating queries into SQL, 332–333
 - temporal queries, 319–320
- Agency for Healthcare Research and Quality (AHRQ), 131
- Alanine aminotransferase (ALT), 7
- Alert
 - accuracy
 - clinician judgment, 115
 - computational sophistication, 114
 - incomplete information, 113
 - individual variation, 114
 - medication, 114–115
 - medication reconciliation, 113
 - software integration, 113
 - allergy, 111
 - customizability, 120–121
 - drug-drug interaction, 110–111
 - implementation, 111–112
 - laboratory test, 110–111
 - Microsoft Office, 119–120
 - thresholds, 115–116
 - unintended consequences, 117–119
 - work flow and software-integration, 116–117
- Alzheimer's disease, 55
- American Society of Testing Materials (ASTM), 124
- Apgar score, 262
- Arden syntax
 - decision-support framework, 130
 - essential features, 125–126
 - explanation section, 125
 - limitations
 - Arden code, 128–129
 - batch-oriented algorithms, 127–128
 - data structures and library routines, 129
 - EMR database schemas, 127
 - event mechanism, 126
 - laissez-faire approach, 127
 - MLM, 125
 - programming language, 124
- Asynchronous JavaScript and XML (AJAX), 94
- Audit-trail table, 195–197
- Automatic interface generation, 78–79

B

- Binary data type, 24
- Binary-large-object (BLOB), 88

- Biomedical data modeling
 - ER diagram, 47
 - UML, 47–48
 - XML uses, 48–49
- Biomedical metadata standards
 - descriptive metadata standards (*see* ISO/IEC 11179)
 - gene expression and proteomics, 377–378
 - HL7v3 reference information model
 - association classes, 381
 - clinical document architecture (CDA), 384–385
 - compatibility and stability issues, 382
 - extensibility, 381
 - repetition elimination, 383–384
 - top-level classes, 380–381
 - XML-based messaging standard, 382–383
 - human-usable metadata standard, 374
 - internal evaluation, 385
 - OpenEHR Archetypes, 379–380
 - proteomics standards, 378–379
 - self-describing attribute-value pairs, 374–375
 - standardization challenges, 373
 - technical metadata standards (*see* Clinical Data Interchange Standards Consortium)
- Bitmap indexes, 60–61
- Boolean data type, 18
- Boolean query, 66–67
- Bulk data extraction
 - array data extraction, 345–346
 - clinical data extraction, 294, 344
 - extraction algorithm, 348–349
 - extract specification, 346–347
 - temporary data tables, 347–348
- Business process execution language (BPEL), 165
- Business process markup notation (BPMN), 166–168
- C**
- Cascading style sheet (CSS), 92
- Categorical data type, 17–18
- Caveats
 - Boolean query, 66–67
 - EAV data pivoting, 67–68
 - EAV design, 69
 - EAV infrastructure, 68–69
 - genotyping, 69–70
 - NOT operation, 67
- Cimino's criteria
 - requirements of terminology, 236–237
 - vocabulary in domain, 237–238
- Clinical data. *See also* Structured clinical data
 - ERP applications, 55
 - flexibility, 56
 - protocol, 56–57
 - representation, structured
 - clinical event, 58
 - patient's problem, 58
 - problem recovery, 59
 - software developer
 - Caché, 72
 - InterSystems, 73
 - myth and reality, 70–72
 - sparse data storage
 - attribute-value pair and event, 59
 - caveats (*see* Caveats)
 - EAV (*see* Entity-attribute-value (EAV) model)
 - key search, 60–61
 - key-value pair storage, 62–63
 - specific parameters, 55–56
- Clinical Data Interchange Standards Consortium (CDISC), 376–377
- Clinical document architecture (CDA), 384–385
- Clinical-guideline representation language
 - doctor/amateur-as-programmer metaphor, 182–183
 - expression-language standardization efforts, 186–187
 - general-purpose workflow-language standard, 183
 - infrastructure-creation challenge, 181
 - medical-software development, 184–185
 - operational runtime engine, 185–186
 - workflow technology, 181–182
 - XML-based syntax, 181
- Clinical research information systems (CRISs), 77
- Clinical study data management systems (CSDMSs), 6
 - analysis and reporting, 290–291
 - vs.* EMRs
 - critical quality control, 280–281
 - differential access support, 282
 - electronic data capture, 279–280
 - eligibility criteria, 275
 - event-CRF cross-table, 276–277
 - event scheduling, subject calendar, 276
 - experimental design representation, 282–283
 - multi-site scenarios, 283–284
 - patients *vs.* subjects, 274
 - personal health information storage, 281–282

- projects, 278
 - real-time self-reporting, subject data, 278
 - recruitment, 274
 - software localization, 284–285
 - structured data capture, 279
 - protocol management and study conduct
 - patient-monitoring and safety, 290
 - related tasks, 289–290
 - separate-database approach
 - meta-analytic data extraction, 287
 - reusing parameters, 286–287
 - study planning and protocol authoring
 - informatician's role, 288–289
 - recruitment and eligibility
 - determination, 289
 - scheduling enrolled subjects, 289
 - Common object request broker architecture (CORBA), 191
 - Common toxicity criteria (CTC), 83
 - Common Warehouse Metamodel (CWM), 364–365
 - Complex decision support
 - black box, 187
 - business workflow technology, 177
 - clinical-guideline representation language
 - doctor/amateur-as-programmer metaphor, 182–183
 - expression-language standardization efforts, 186–187
 - general-purpose workflow-language standard, 183
 - infrastructure-creation challenge, 181
 - medical-software development, 184–185
 - operational runtime engine, 185–186
 - workflow technology, 181–182
 - XML-based syntax, 181
 - drug interaction, 178–179
 - executable knowledge module, 187
 - metadata-driven software, 177
 - multiple working implementation, 191
 - proteus guideline framework, 189–191
 - QMR, 179–180
 - table-driven approach, 177–178
 - terminology service, 188
 - web service, 188
 - XML-based protocol, 188
 - Computed field implementation
 - dependency, 90–91
 - expression language, 89
 - security consideration, 90
 - Computerized physician order entry (CPOE), 110
 - Concept recognition
 - in clinical and scientific text, 239–241
 - pre-coordinated vs. post-coordinated concepts
 - deprecation reasons, 242–243
 - hypertension and renal disease, 241–242
 - software assistance, 243–244
 - Conceptual compound query, 331–332
 - Conditional list
 - ergonomic operation, 83
 - interface generation, 95
 - Conference on Database Systems Languages (CODASYL), 130
 - Custom data type, 25
- D**
- Data interchange
 - using XML, 50–51
 - without using XML, 49–50
 - Data mart, 359
 - Data modeling, 48–49
 - Data-source metadata
 - concepts and terms (synonyms), 366
 - constraints, 367–368
 - data stores, 366–367
 - extra columns, 366
 - Data type
 - binary, 24
 - categorical and ordinal, 17–18
 - custom (user-defined), 25
 - date, 22
 - enumerated data representation
 - missing values, 29
 - older approach, 26–27
 - recommended approach, 27–29
 - numeric
 - integers, 19–21
 - real numbers, 21–22
 - referential integrity, 29–31
 - structured vs. unstructured data, 25
 - text
 - pseudo-numeric data, 23
 - storage data, 24
 - Data warehousing
 - architectural differences, 359–360
 - creating approaches
 - data constraints discovery, 362
 - data identification/discovery, 361
 - data loading, 364
 - data profiling, 362
 - source data extraction, 362
 - transformation, 363–364
 - vs. data mart, 359
 - data-source metadata

- concepts and terms (synonyms), 366
 - constraints, 367–368
 - data stores, 366–367
 - extra columns, 366
- fact table, 360–361
- front-end metadata, 369–370
- I2B2 star schema
 - fact table, 370–371
 - purpose of anonymization, 370
 - security architecture, 371
- metadata during ETL process
 - dependency (lineage) metadata, 369
 - process performance metadata, 369
 - quality indicators and error metadata, 368
- metadata management toolset, 365
- metadata role, 364–365
- star-schemas, 361
- Date data type, 22
- Decentralized Hospital Computer Program (DHCP), 70
- Decision-making
 - alert
 - accuracy, 113–115
 - allergy, 111
 - customizability, 120–121
 - drug-drug-interaction, 110–111
 - implementation, 111–112
 - laboratory test, 110–111
 - Microsoft Office, 119–120
 - thresholds, 115–116
 - unintended consequences, 117–119
 - work flow and software-integration, 116–117
 - open and closed loop system, 110
 - patient care, 109–110
- Decision support infrastructure
 - Arden model, 157
 - complex-guideline implementation, 157
 - workflow
 - BPEL and BPMN limitation, 168–169
 - business process execution language, 165
 - business process markup notation, 166–168
 - characteristics, 158–159
 - design implementation, 164–165
 - ERP, 173–174
 - flexibility, 171–172
 - HTTP protocol, 160
 - identifying and implementing reusable functionality, 162–163
 - Java-based workflow code frameworks, 170
 - model, 157–158
 - multi-tier client-server approach, 160
 - performance, 163–164
 - pure visual programming metaphor limitation, 169–170
 - rapid reflex reaction, 172–173
 - software framework, 159–160
 - web service basics, 161–162
 - WF essentials, 170–171
- Delta tracking, 196–197
- Descriptive metadata
 - algorithmic knowledge, 234
 - Cimino's criteria
 - requirements of terminology, 236–237
 - vocabulary in domain, 237–238
 - description logics (DL)
 - applications, 245
 - subsumption operation, 245
 - supporting semantics, 244
 - large-scale ontologies (*see* Large-scale biomedical ontology)
 - object-oriented modeling, 233–234
 - pre-coordinated *vs.* post-coordinated concepts
 - deprecation reasons, 242–243
 - hypertension and renal disease, 241–242
 - software assistance, 243–244
- SNOMED-CT, 246
- terminology
 - basic taxonomy, 228, 231
 - concept recognition, 239–241
 - content of, 227
 - controlled vocabulary, 228, 230
 - data exchange basis, 238–239
 - folksonomy, 230–231
 - ontology, 229
 - poly-hierarchy approach, 232
 - pure-hierarchy approach, 231–232
 - thesaurus, 229, 232
 - use, 227
- Unified Medical Language System (UMLS), 234–235, 246–247
- Descriptive metadata standards.
 - See* ISO/IEC 11179
- Dictionary/metadata tables
 - attribute description table, 200
 - attribute hierarchy table, 201
 - Choice_Sets and Choice_Set_Values tables, 200
 - class categorization, 201–202
 - class description table, 200
 - class hierarchy table, 200
 - class presentation information, 201
 - dictionary and catalog table synchronization, 203–204
 - ergonomics metadata table, 201
 - essential sub-schema, 202
 - metadata absence, 205–206

- relationships table, 200
- user-defined data types, 204–205
- Distributed common object model (DCOM), 191
- Double data entry (DDE), 280–281
- Drill-down interface, 210
- Drools
 - rule actions, 153
 - rulesets, 153
 - variable_rule_dependency, 153–154
 - variables, 153
- Duplicate-suppression algorithm, 307
- E**
- Electronic decision support implementation
 - Arden syntax (*see* Arden syntax)
 - clinical guidelines
 - ambiguities and under-specification, 132–133
 - bypass implementation, 134
 - caveat emptor, 131
 - complex guideline, 131–132
 - flowchart nodes, chronological separation, 134
 - GEM, 135
 - guideline-software framework, 135
 - implicit assumption, 133–134
 - patient specificity, 134
 - surgical-risk evaluation protocol, 133
 - decision-support software
 - curly-braces, 136
 - database views, 136–137
 - subroutine library, 137–138
 - vMR (*see* Virtual medical record)
 - EMR design, 123–124
- Electronic mechanism. *See* Alert
- Electronic medical records (EMRs), 5, 55
- Enterprise resource planning (ERP), 55, 173–174
- Entity-attribute-value (EAV) model, 123
 - approach, 63–64
 - conceptual design, 65
 - data pivoting, 67–68
 - data representation
 - Microsoft SQL Server, 225
 - XML for sparse attributes, 224
 - design indications, 69
 - audit-trail table, 195–197
 - data classes and modest data volumes, 194
 - hybrid classes, 194–195
 - Semantic-Web applications, 197
 - dictionary/metadata tables
 - attribute description table, 200
 - attribute hierarchy table, 201
 - Choice_Sets and Choice_Set_Values tables, 200
 - class categorization, 201–202
 - class description table, 200
 - class hierarchy table, 200
 - class presentation information, 201
 - dictionary and catalog table
 - synchronization, 203–204
 - ergonomics metadata table, 201
 - essential sub-schema, 202
 - metadata absence, 205–206
 - relationships table, 200
 - user-defined data types, 204–205
- EAV/CR approach
 - dictionary tables, 200
 - framework components, 199
 - RDBMS catalog tables, 199–200
 - self-mapping, 214–215
- EAV/CR data sub-schema
 - array representation, 209
 - controlled terminologies links, 208–209
 - object dictionary approach, 206–207
 - sub-structure representation, 208
- infrastructure, 68–69
- metadata schema details, 222–224
- mixed schemas requirements
 - data table classes, 197
 - data-type specific table, 198
 - limitations of basic EAV model, 197–198
 - supporting routine operations, 198
- phenotype representation
 - definitions, 220
 - repository representation, 220–222
- search/browsing considerations
 - attributes grouping, 213
 - display mode vs. editing mode, 213
 - drill-down interface, 210
 - object of interest search, 213–214
 - order of presentation, 212
 - summary vs. detail views, 210–212
- selective user access, 219
- SQL generation for CRUD operations
 - create operation implementation, 219
 - delete operation implementation, 217
 - SQL injection defenses, 216–217
 - unauthorized access via a login screen, 216
 - update operation implementation, 217–218
- Entity-relationship (ER), 47
- Ergonomic operation
 - conditional list, 83
 - default values, 81
 - electronic form, 81–82
 - online help, 82–83

- optional field, 82
- skip logic, 81
- Eval mechanism, 146–147
- Event-CRF cross-table, 276–277
- Event scheduling, subject calendar, 276
- eXtensible Markup Language (XML), 12
 - begin-tag, 33
 - biomedical data modeling, 47–49
 - content, 34
 - data export, 51
 - data interchange
 - using XML, 50–51
 - without using XML, 49–50
 - end-tag, 33
 - vs. HTML, 34–35
 - medical natural-language processing, 51–52
 - misconception
 - document content, 42–43
 - human work, 41–42
 - procedural programming, 44–45
 - relational database (*see* Relational database)
 - reference content, 49
 - strength
 - bracketed syntax, 36
 - buzzwords, 35
 - configuration files, 37
 - data format, 38
 - large organization, 38–40
 - message digest, 36–37
 - tool base, 37–38

F

- Folksonomy, 230–231

G

- Gene expression and proteomics, 377–378
- Glycosylated hemoglobin (HbA1C), 112
- Graphical user interface (GUI), 3
- Guideline Element Language, Object-oriented (GELLO), 186–187
- Guideline Element Model (GEM), 135

H

- Hash indexes, 60
- Health level seven (HL7), 7
- Heterogeneous data models, data retrieval
 - ad hoc query (AHQ)
 - advice for academic informaticians, 297
 - comprehensible output, 350–352
 - definition, 294
 - drill-down approach, 320–321
 - individual attribute metadata, 321–322
 - key-phrase or partial string search, 321

- problem constraints, 296–297
- query representation schema, 326–330
- schema volatility problem, 324–326
- selection criteria and output options, 322–324
- sparse-data problem, 317–319
- SQL generation, 330–340
- temporal queries, 319–320
- bulk data extraction
 - array data extraction, 345–346
 - clinical data extraction, 294, 344
 - extraction algorithm, 348–349
 - extract specification, 346–347
 - temporary data tables, 347–348
- caching metadata, 300
- conceptual operation, 293
- metadata in EAV/CR designs, 356–358
- non-relational data access
 - full-text indexing and query, 341–342
 - querying full-text, 342
 - querying XML data, 342–343
- pre-compilation of queries, 340–341
- problem of querying, 294–295
- query by form (QBF)
 - number of returned records, 304
 - patterns for criterion specification, 302–303
 - results presentation, 303–304
- restricting the output, 300–301
- simple searches, 294
- standard reports
 - banded reports, 306–307
 - canned reports, 294
 - composing SQL, 306
 - metadata-based framework, 308–310
 - meta_report_columns, 313–314
 - meta_report_params, 314–315
 - meta_reports, 310–311
 - meta_reports_roles, 315
 - meta_report_steps, 311–313
 - multiple HTML tables, 307–308
 - ODBC-related issues, 305
 - reporting framework runtime, 315–316
 - VQBE for query generation, 305–306
- UDB concept
 - cycle detection and elimination, 354–355
 - hidden schemas, 355–356
 - weaknesses, 353–354
- VQBE, 295–296
- warehouse schema design
 - enumerated data types, 298–299
 - object data types, 299
 - vs. production schema, 298

HL7v3 Reference Information Model
 association classes, 381
 clinical document architecture (CDA),
 384–385
 compatibility and stability issues, 382
 extensibility, 381
 repetition elimination, 383–384
 top-level classes, 380–381
 XML-based messaging standard, 382–383

I

Inference engine. *See* Rule engine
 Informatics for Integrating Biology and the
 Bedside (I2B2) star schema
 fact table, 370–371
 purpose of anonymization, 370
 security architecture, 371
 Institute of Medicine (IOM), 109
 Integers, 19–21
 Interface generation. *See also* Structured
 clinical data
 electronic-form definition, 84–85
 essential table, 86–87
 forms, 87
 groups, 87
 intermediate group of parameters, 85–86
 parameters, 88–89
 InterSystems Caché, 72
 ISO/IEC 11179, 375–376

J

JavaScript Object Notation (JSON), 94

K

Knowledge bases, 252–253

L

Laissez-faire approach, 127
 Language independent query (LINQ), 52
 Large-scale biomedical ontology
 change management
 allowing or disabling changes, 268
 metadata and data tables, 266
 metadata change detection, 267–268
 testing and reporting consequences, 268
 closed-world vs. open-world, 255
 core thesaurus and EAV/CR metadata
 schema, 257
 enforcing selective access, 265–266
 functional requirements, 250–251
 imported inter-concept relationships,
 264–265
 local content invalidation, 269
 lower-level and higher-level metadata

relationship types and semantic
 inverses, 264
 SNOMED, 262–263
 merged concepts, 269
 metadata and data subschema
 schemas, 260
 ontologies vs. databases
 knowledge bases, 252–253
 modern RDBMS, 251–252
 prototypes vs. production systems
 preventing misuse challenge, 255
 Protégé tool, 253–254
 scaling up challenges, 254–255
 semantic web technologies, 256–257
 SNOMED CT concept, 269–270
 thesaurus kernel
 concepts/objects table, 259
 inter-concept relationships, 260
 local IDs vs. content IDs, 258–259
 rapid search of terms, 259–260
 terms/synonyms table, 259
 types and enumerations, 261–262
 UMLS, class and attribute structure, 261
 Lemmatization, 260

M

Machine-readable concept model (MRCM),
 269–270
 Massachusetts utility multi-programming
 system (MUMPS)
 FORTRAN and BASIC, 71
 programming language, 70–71
 VAMC, 72
 Medical Logic Module (MLM), 125
 Medication reconciliation, 113
 Metadata
 biomedical system
 knowledge, evolution, 4
 user interface, 5–6
 book organization, 8–9
 caching, 300
 classification, 8
 database management, 2–3
 data input, 80–81
 definition, 1
 ergonomic operation, facilitation
 (*see* Ergonomic operation)
 implementation principles
 drive software, 10–11
 guidelines, 11–12
 inter-operability
 HL7, 7
 semantic data, 7–8
 vendor's system, 6

pitfalls
 development tool, 14
 documentation, 15
 software requirement, 13
 protocol-subcomponent, 83–84
 safeguarding, 12–13
 singular/plural, 2
 Metadata-driven software system, 2–3
 Microarray gene expression data
 (MGED), 47
 MicroArray Gene Expression Object Model
 (MAGE-OM), 377–378
 Microsoft office, 119–120
 Modeling and implementing skip logic
 combination of parameters, 92
 form-design, 93
 skip_dependent records, 91
 web pages, 92

N
 National Guideline Clearinghouse (NGC), 131
 Natural Language Processing (NLP),
 239–240
 Nominal data type. *See* Categorical data type
 Non-first-normal form (NF2) design, 209
 Non-relational data access
 full-text indexing and query, 341–342
 querying full-text, 342
 querying XML data, 342–343

O
 Object dictionary approach, 206–207
 Online mendelian inheritance in man (OMIM),
 51
 Ontologies vs. databases
 knowledge bases, 252–253
 modern RDBMS, 251–252
 Ontology, 233–234
 Open Database Connectivity (ODBC),
 72, 305
 OpenEHR Archetypes, 379–380
 OpenMRS, 108
 Ordinal data type, 17–18
 OWL, 256–257

P
 Personal health record (PHR), 113
 Protégé tool, 253–254
 Proteomics standards, 378–379
 Proteus guideline framework, 189–191
 Protocol management
 patient-monitoring and safety, 290
 related tasks, 289–290
 Pseudo-numeric data, 23

Q
 Query by form (QBF)
 number of returned records, 304
 patterns for criterion specification, 302–303
 results presentation, 303–304
 Query representation schema
 non-temporal criteria and output, 328–329
 query-definition subschema, 326–328
 temporal criteria, 329–330
 Quick medical reference (QMR), 179–180
 Quinlan's algorithm, 150–151

R
 Ranked data type. *See* Ordinal data type
 Rapid application development (RAD), 14
 RDBMS catalog tables, 199–200
 Real numbers, 21–22
 Reference information model (RIM), 138–139
 Referential integrity, 29–31
 Relational database (RDB), 63
 complex validation, 45
 legitimate uses, 46
 Schematron, 47
 XQuery, 45
 Relational database management systems
 (RDBMSs), 3
 Resource Description Framework (RDF), 197
 Rule engine
 comprehensiveness and reliability, 150–151
 condition evaluation, 146
 conflicting or redundant rule, 148–149
 decision tree, 151
 drools, 152–154
 Eval mechanism, 146–147
 forward and backward chaining, 145
 individual action, 144
 inference engine, 143
 managing schema, 151, 152
 rule-based software system, 143
 rule framework strength, 147–148
 runtime evaluation, 146
 software-engineering skill, 149–150
 stack representation, 147
 syntax, 144
 working memory, 144

S
 Scalable vector graphics (SVG), 44
 Schematron, 47
 Schema volatility problem
 loading into user interface, 295–296
 SQL code storage, 294–295
 Semantic Web Rule Language (SWRL),
 256–257

- Separate-database approach
 - meta-analytic data extraction, 287
 - reusing parameters, 286–287
- Serum glutamic-pyruvic transaminase (SGPT), 7
- Service-oriented architecture (SOA), 161–162
- SNOMED, 263, 269–270
- SNOMED-CT, 246
- Software localization, 284–285
- SQL generation
 - AHQ
 - conceptual compound query, 331–332
 - more physical table group, 334–335
 - multiple classes in EAV/CR, 335–337
 - multiple output tables, 338–340
 - problem simplification, 330–331
 - single physical table group, 333
 - translating queries into SQL, 332–333
 - CRUD operations
 - create operation implementation, 219
 - delete operation implementation, 217
 - SQL injection defenses, 216–217
 - unauthorized access via a login screen, 216
 - update operation implementation, 217–218
- Stemming, 260
- Structured clinical data
 - data edit capture
 - clinical data sub-schema, 100–101
 - form elements and database elements, 100, 102–103
 - metadata table, 98–99
 - SQL operations, 103
 - flowsheet, 106–107
 - form generation, 97–98
 - forms runtime environment
 - browser code development, 104
 - technology, 105–106
 - web server code, 104–105
 - generated form and troubleshooting, 96–97
 - interface generation
 - computations and validations, 94
 - computed field implementation, 89–91
 - conditional list, 95
 - form elements, definition
 - (*see* Interface generation)
 - modeling and implementing skip logic, 91–93
 - modeling complex validation and alerts, 93–94
 - practical issue, 95
 - metadata
 - data input, 80–81
 - ergonomic operation, facilitation, 81–83
 - protocol-subcomponent, 83–84
 - open-source EMRs
 - OpenMRS, 108
 - VAMC VistA, 107
 - patient's information, 75
 - pitfalls, 76
 - protocol representation, 77
 - SOAP, 75
 - user interface
 - automatic interface generation, 78–79
 - hand-coding, 77–78
- Structured Query Language (SQL), 3
- Study calendar
 - event-CRF cross-table, 276–277
 - event scheduling, subject calendar, 276
- Study planning and protocol authoring
 - informatician's role, 288–289
 - recruitment and eligibility
 - determination, 289
 - scheduling enrolled subjects, 289
- T**
- Technical metadata standards.
 - See* Clinical Data Interchange Standards Consortium
- Temporal queries, 319–320
- Terminology. *See also* Descriptive metadata
 - basic taxonomy, 228, 231
 - concept recognition, 239–241
 - content of, 227
 - controlled vocabulary, 228, 230
 - data exchange basis, 238–239
 - folksonomy, 230–231
 - ontology, 229
 - poly-hierarchy approach, 232
 - pure-hierarchy approach, 231–232
 - thesaurus, 229, 232
 - use, 227
- Text data type, 23–24
- Thesaurus, 229, 232
- Thesaurus kernel
 - concepts/objects table, 259
 - inter-concept relationships, 260
 - local IDs vs. content IDs, 258–259
 - rapid search of terms, 259–260
 - terms/synonyms table, 259
- Toxic shock syndrome, 57
- U**
- Unified Medical Language System (UMLS), 234–235, 246–247, 261
- Unified Modeling Language (UML), 47–48, 138
- Unique identifiers (UIDs), 63

Universal database (UDB) concept
 cycle detection and elimination, 354–355
 hidden schemas, 355–356
 weaknesses, 353–354
Universally unique identifiers (UUIDs),
 20–21
User-defined data type. *See* Custom data type

V

VAMC VistA, 107
Virtual medical record (vMR)
 cross-EMR software development, 139–140
 HL7 v3.x Reference Information Model,
 138–139
 mapping guideline data element, 140–141
Visual Query By Example (VQBE), 295,
 305–306

W

Warehouse schema design
 enumerated data types, 298–299
 object data types, 299
 vs. production schema, 298
Workflow

 BPEL and BPMN limitation, 168–169
 business process execution language, 165
 business process markup notation,
 166–168
 characteristics, 158–159
 design implementation, 164–165
 ERP, 173–174
 flexibility, 171–172
 HTTP protocol, 160
 identifying and implementing reusable
 functionality, 162–163
 Java-based workflow code frameworks,
 170
 model, 157–158
 multi-tier client-server approach, 160
 performance, 163–164
 pure visual programming metaphor
 limitation, 169–170
 rapid reflex reaction, 172–173
 software framework, 159–160
 web service basics, 161–162
 WF essentials, 170–171
Work flow and software-
integration, 116–117