Managing Systems Development 101

A Guide to Designing Effective Commercial Products & Systems for Engineers & Their Bosses/CEOs

James T. Karam

The Technical Manager's Survival Guides, Volume 2
Marcus Goncalves, Series Editor

The Technical Manager's Survival Guides

Volume 1: *Team Building,* by Marcus Goncalves

Volume 2: *Managing Systems Development 101*, by James T. Karam

Library of Congress Cataloging-in-Publication Data

Karam, James T.
Managing systems development 101 : a guide to designing effective commercial products & systems for engineers & their bosses/CEO's / James T. Karam.
p. cm.
Includes bibliographical references and index.
ISBN 0-7918-0258-2 (alk. paper)
1. Systems engineering--Management. 2. Project management. I. Title.

TA168.K3567 2007
620.001'171--dc22

2006100873

Table of Contents

Table of Figures v

Table of Tables v

Preface vii

Acknowledgments ix

Introduction 1

Chapter 1 Project Systems Engineering 101 5
- Design Requirements 7
- Verification & Validation 11
  - Reviews 12
  - Analysis & Similarity 15
  - Test 16
- Barbie® Dolls 18
- Change Management 19
- Third Time's the Charm 20

Chapter 2 Program Planning 101 23
- Noah's Principle & Earned Value 32
- Scheduling Morality 42
- Management Reserve 44

Chapter 3 System Evolution 47
- Bid & Proposal 47
- Architect for Fault Tolerance 50
- Make It Work, then Robust. Only Then, Make It Better. 52
- Branching is a Necessary Pain 53
- Numbers are Better than Judgment 54
- Customers Need Managing Too 55
- Closing Out 56

Chapter 4 Often Forgotten Programming 101 57

Chapter 5 User Interface Design 101 63
- Clickable Mockups, Often in Lieu of Specs 64
- Admittedly Biased Design Practices 65

Chapter 6 Presentations 101 73

Chapter 7 Find & Flush the Full In-Boxes 77

Chapter 8 Continuous Improvement 101 79
Categorizing Defects 80
Engineering Metrics 88
Production & Service Metrics 90

Chapter 9 Performance Ranking 101 95

Chapter 10 Incentive Criteria 101 99

Chapter 11 Matrix Organization 101 103

Chapter 12 Tailor Your Behavior to the Software, not Vice Versa 107
I've Never Found the Software that I'd Rather Write than Buy. 108

Closing Thoughts 113

Additional Reading 115

Index 119

About the Author 123

Table of Figures

Figure 1.1 Key System Engineering Elements 6
Figure 1.2 Decomposition Hierarchy 8
Figure 2.1 Ditch Digging Project Plan 24
Figure 2.2 Estimating with Factors 31
Figure 2.3 CPI Likelihood 34
Figure 2.4 Cumulative Earned Value 36
Figure 2.5 Earned Value Indices 37
Figure 2.6 Incremental Earned Value 39
Figure 2.7 Integrated Earned Value Status 41
Figure 5.1 GUI Illustration 66
Figure 6.1 Horse Charts 76
Figure 7.1 Full In-boxes 78
Figure 8.1 Bug Quantity 89
Figure 8.2 Bug Aging 89
Figure 8.3 Work In Progress (WIP) Defects 91
Figure 8.4 Install Defects 92
Figure 8.5 Mature Product Post Install Defects 92
Figure 8.6 New Product Post Install Defects 93
Figure 9.1 Merit Pay versus Rank 98
Figure 10.1 Individual Performance Incentive 100
Figure 10.2 Group Performance Incentive 101

Table of Tables

Table 2.1 Project Planning Granularity 25
Table 8.1 Defect Severity Classes 81
Table 8.2 Defect Urgency Codes 85
Table 8.3 Known Issues 87
Table 11.1 Boss Duality in a Matrix 103

Preface

I have had the good fortune to be associated with the development of large-scale systems for over forty years. These are products that are developed by more than one team, working in parallel, which must be interfaced and integrated together. The point is not so much their physical size but the need to manage and integrate multiple efforts simultaneously. Experience suggests that a single good lead engineer can indeed keep a design all in his head and direct a handful or so of engineers. While that works for many games, web applications, and IT projects, it does *not* work for systems. There are just too many people involved, in more than one team, and often not even co-located.

I was particularly blessed to start my career as an R&D officer in the United States Air Force (USAF) in the timeframe when systems engineering was being formalized well by the Department of Defense (DOD), and the Air Force in particular, based on their good and bad experiences in the late fifties fielding Intercontinental Ballistic Missiles (ICBM's). As I moved out from aerospace into commercial developments, there was a learning curve on my part regarding how much of those aerospace processes and formalism were relevant in this seemingly different arena. I soon concluded that those processes were key for *any* successful system development. Only the formalism was negotiable or tailorable.

I frequently found myself resurrecting some common threads of advice and direction as I moved among several industries and company organization types. It did not seem to matter what we were making, or whether it was a large multi-national corporation or one with the founder still in sole control. The engineering management issues were eerily the same. I would pull out an earlier presentation or document, tweak a logo and a bit of text, and influence a new set of staff. This book is a heavily edited and expanded compilation of those lessons re-taught over the years.

You will find the advice is invariably basic, hence the titles ending in "101". The management problems encountered were because of a failure to understand or enforce those basics, and their enforcement is *not* easy. In effect, experience says that your focus should always remain on these basics.

I have intentionally tried to make this book easy to browse using a somewhat unique style that evolved over the years. Most chapters use a bold-type opening sentence in each paragraph. You can get the key assertions by just skimming them. Those claims are elaborated in the

rest of the paragraph. If the reader is familiar with systems engineering terminology, that is probably sufficient. If not, I have often followed with subsequent indented paragraphs that elaborate further.

This book is likely most valuable to young engineers who are moving out of their academic specialty into engineering or project management, about which they probably were taught very little that was practical. And, yes, I shoulder some of that blame myself since there is a stint of teaching graduate engineering school on my resume'. The book is also intentionally succinct. While we usually explain our rationale, rather than just assert, our intent is to provide the reader with cogent advice that they can quickly absorb and effectively apply. As such, it should also serve as a useful quick reminder to more senior professionals, typically when they have been given a broadening assignment that forces them into new professional terrain.

Acknowledgments

David Lapczynski and Dr. Milton Franke were invaluable in their insightful review and comment on several drafts of this book. Dave is the COO at Cubic Transportation Systems and was a great last boss as well as a good friend. At his behest, I would like to beat a dead horse and re-emphasize the importance of detailed, resourced schedules for managing projects or product developments. Milt was, in effect, my first boss as he was my major professor on my Masters thesis at the Air Force Institute of Technology, where he still actively teaches, and is likewise a life-long friend. I was blessed with working with many true professionals all of my career, but none better than these at the start and end.

I mainly want to thank my wife Alicia for enduring almost thirty years of marriage while retaining such a gracious and loving spirit. She is my best friend of all.

Introduction

So, are you a young engineer that has been asked to become a lead for a team of specialists to work on a product or project that requires many different skills or even several teams? Have you been a lead engineer but have now been asked to be a manager of your department? Have you shown both the inclination and the capability to broaden out of your specialty to become a project or product development chief engineer or manager? Have you managed projects or departments and now you have been asked to manage those managers? In all these cases, you are confronting topics daily that they never taught you in school as you find yourself involved with managing the engineering of what are called "systems".

Managing the development of large-scale systems can be both fun and satisfying. The U.S. Department of Defense (DOD), notably the Air Force (USAF), codified the methodology of such management in the late fifties and sixties in MIL-STD 499 and its ilk. They took their lessons learned from fielding Intercontinental Ballistic Missiles and the like, both good and bad, and embodied them in processes that continued to mature. Many engineers spent at least some of their career in aerospace and this systems culture. However, since "peace broke out" in the early nineties, this opportunity for systems on-the-job training (OJT) has substantially diminished.

This book addresses many of the key topics you will face in your expanded responsibilities. There are good textbooks on the topic of systems engineering, but most still focus primarily on the very large systems of systems typical of aerospace and defense. Further, as textbooks, they tend to focus understandably on the generic processes involved, primarily regarding the earlier phases of development. Regardless, several are cited in a closing section as candidates for additional reading. Instead, this book focuses on specific practical advice to use when executing those processes in commercial environments. In effect, our focus is on the practical mechanics of management. As such, it can also provide an incisive refresher of useful tricks of the trade even for professionals in aerospace.

While large commercial systems also existed, they were mostly the domain of mainframe computer developers until the eighties with its advent of the ubiquitous personal computer (PC). Then the nineties saw the introduction of the World Wide Web (WWW) and a plethora of personal and business software applications of all sizes. Further, PCs became so powerful that many, if not most, applications that used to require large computers or, more commonly, highly specialized and

customized electronic hardware could now run on these relatively cheap machines.

Almost every capital goods industry saw their hardware commoditized to some degree with their products' functionality provided mostly by software. This commoditization of hardware was a watershed event as it meant that software development would become a critical asset (or heartache) for most every industry and product. Moreover, large systems were routinely created using a collection of PCs, some evident and some embedded, but PCs nonetheless. So, where did developers learn to put such commercial systems together? Folklore said that aerospace processes were gross overkill with an excessive focus on paperwork.

In addition to the regulated industries like nuclear and medical equipment that had done so previously, most companies in all industries formalized their system development processes in response to the pragmatically mandatory need to get themselves certified to the ISO-9000 quality standard in the nineties. Many made the mistake of over-promising, particularly with respect to the paperwork, since they proposed to behave like they thought someone might have expected, rather than what they had always done. Either they drowned in their own paperwork, or, more commonly, quickly lapsed into old habits and prayed an auditor would not show soon. (The proper solution was to edit the procedures and processes to reflect what was reasonable. Generally, auditors do *not* tell you what you should do, but only if you are complying with what *you* said you should do.)

As one who stumbled through some of those choices, my conclusion quickly became that, while one should tailor the formalism in a commercial environment, systems are systems, and the aerospace system engineering process basics remain the key to success anywhere. While somewhat facetious, the section titles typically end in "101" because the basics are where your problems, and their solution, lie.

Chapter 1 starts with a review of the key elements of the project systems engineering process. While still the way of life in aerospace and defense, many engineers in commercial enterprises lack exposure to even the terminology of systems development. This initial chapter provides that context along with practical advice regarding execution. Project/program planning is addressed in Chapter 2, as these plans, in effect, become the internal contracts between the various development groups and their management and customers. In fact, it is hard to even claim that one *is* a manager without a plan, much less actually manage, rather than just react. This section ends with Chapter 3 discussing

several topics to consider pragmatically during the various phases of a program or product's lifecycle or evolution, notably at the beginning and at the end of a project.

The next chapters address some of the key mechanics of managing systems development. Since software is such a dominant part of any system nowadays, we start Chapter 4 with a set of very basic design practices that seem to be ignored or forgotten by developers. These topics *were* taught in school, probably in their introductory courses, and staff usually resent being reminded. However, they recur so often that they should remain your focus. Chapter 5 recommends using clickable mockups to facilitate timely development of graphical user interfaces (GUIs) in products. While admitting that they represent just one particular religious bias, we also include an example of GUI design practice rules. We said "religious" because, like many other issues, there is no technical right or wrong involved, just a preference. Nevertheless, the benefit arises to your team because you state your belief, almost independent of its specifics.

Chapter 6 moves away from managing software to using software to make presentations. Every manager is also, some would say mostly, a salesperson. While presentation style would seem to be the ultimate religious preference, we recommend that you become a zealot. Very simple rules are recommended, and they work. Chapter 7 implores and explains how to find and empty all the full in-boxes in your span of control. Nothing you can do will improve responsiveness more. Then, the process of Continuous Improvement is advocated and explained in Chapter 8, with practical examples from all operational departments.

The next set of chapters address people-related topics since people are your means to success. Chapters 9, 10, and 11 address performance ranking, incentive criteria, and matrix organizational structures, respectively. These provide a succinct practical guide to these topics whose mechanics are rarely dealt with, except by osmosis.

Finally, Chapter 12 offers success in improving your productivity with tools, provided you adapt your behavior to them, not vice versa.

Closing remarks refresh our key advice. Candidates for additional reading conclude the text.

Chapter 1 Project Systems Engineering 101

Systems engineering is nothing new but rather a methodical perspective to organizing sound engineering practice in an auditable manner, even when only self-audited. As shown in Figure 1.1, one can group engineering activities into five main categories: requirements, implementation, verification, validation, and record/evolve. While reasonable professional practice in any case, members of regulated industries *must* document all such activities to enable external audit of their effectiveness and integrity.

This chapter presents an overarching design process perspective and terminology, particularly for those readers with minimal exposure to aerospace and defense. Interspersed throughout are pragmatic guidelines and recommended detailed practices. The design process presented is a classical “linear” or “waterfall” scheme, which admittedly has lost its cachet, particularly among academics and large-scale systems of systems practitioners. However, it still represents the foundational basics that will be central to your commercial success. One would typically formalize a procedure and associated internal forms for each box shown in Figure 1.1, e.g., as part of an ISO-9000 certification.

Administratively, the first step in the systems engineering process is the formal authorization of a project/product. Part of that authorization is typically a project plan, which also provides a summary of resources required and schedules. A subsequent chapter discusses planning in more detail. Engineering has likely been involved with a project or product even earlier than this formal authorization event, typically spending sales and/or marketing budget supporting their development of draft specifications, conceptual prototypes, focus group mockups, and the like. However, most companies understandably require a formal authorization event before any non-trivial sums are spent, usually whenever budgeted funds are first provided directly to engineering.

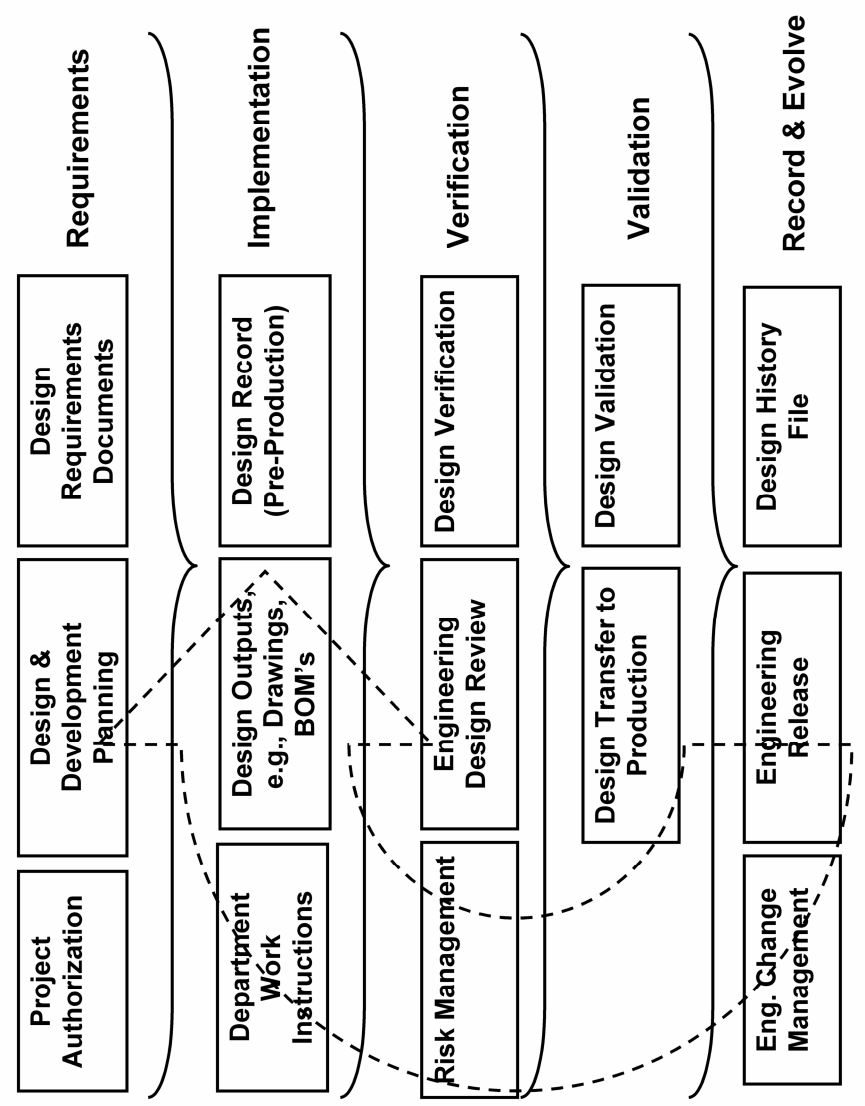

Figure 1.1 Key System Engineering Elements

Design Requirements

Functional requirements start the systems engineering technical process. Functional requirements have a **"black box" perspective.** That is, one should not be able to ascertain anything about a particular implementation. **"Form, fit and function"** (F-cubed) is another common descriptor. As this term implies, a functional specification addresses the inputs, outputs, transfer functions, environments, shape, other physical interfaces, signals and/or commands, other software or electronic interfaces, and the like.

> "Black box" is a common technical slang that implies the viewer is unable to see inside the box. As such, all one can see is how the box behaves in processing its inputs to produce its outputs, just the functional perspective we need in these specifications. For completeness, "white box" means you can see all the internal details. This term is commonly used to describe software testing where one has had access to the creator's source code.
>
> **"That's a solution, not a requirement"** is probably the most common remark you will have to make when reviewing specifications. Again, it seems to be part of the engineering psyche as it is independent of industry and even experience. Since these functional specifications (or design requirements documents, or whatever your company's nomenclature) are often contractual, it is in your self-interest as the developer to retain as much design freedom as possible.
>
> **Commercial customers love to specify solutions also.** Gently push back and recast as a requirement.

Ambiguity in a specification is always to the buyer's advantage. Instead, as a developer, you need as much functional specifics as you can possibly define. Naive staffs seem to think that if requirements are vague or silent, then they get to define what was meant after the fact. Just the opposite is true and is the major cause of feature-creep that has killed many projects, or at least made them painful for the developers. Remember, if the buyer does not believe that you could easily convince a third party that you were in compliance, they retain the ultimate control because they have yet to pay for your product or services. The Golden Rule, "Whoever has the gold, rules", only applies if they believe they would win in court.

This functional specification is the key contract you are making with your bosses or your customer. Developing these is not an easy proposition, and it is so tempting in the honeymoon phase of a project to give in to expediency and get on with the fun of making something. There has even been a recent culture arise in the software community to rationalize that defining requirements in advance is so difficult that one should not even try, but instead should just iterate a design to success. It *is* hard, but you will invariably rue the day that you did not do it. It *can* be done. People have been doing it for years in aerospace and other industries. Moreover, the painful experience with iteration is that it is often a code word for "throw it away and start over". Most such projects will not survive.

Functional requirements are then typically decomposed. Most systems in practice must be implemented with an interacting combination of several peer black boxes. Thus, it is common practice to develop functional requirements for each of these subordinate entities. Notice that this is still a black box perspective, but the requirements have been allocated from the superior entity. Note also that while each subordinate only addresses a subset of the superior's requirements, the mere task of **decomposition introduces new inputs, outputs, and environments** for the subordinate. Each has to interface to its peers, and invariably each has an environment that may be somewhat more stringent that the superior. For example, a printed circuit board is typically exposed to temperatures that are worse than the overall due to peer heating.

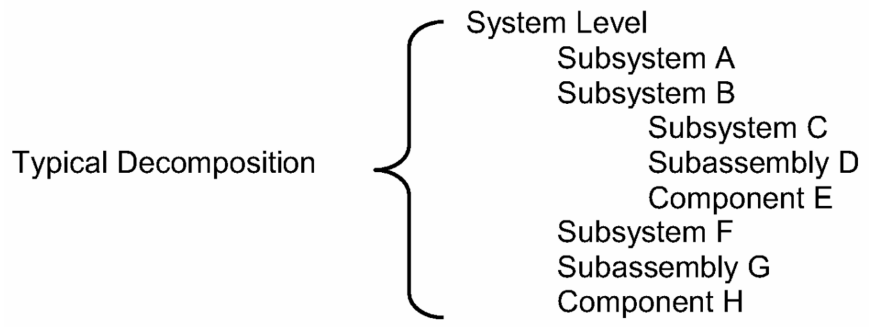

Figure 1.2 Decomposition Hierarchy

Defining some terminology used in Figure 1.2, a subsystem is simply a subordinate system, typically separately specified, developed, and verified by an independent group. A subassembly is just a collection of components, typically that cannot function stand-alone. A component is just a piece part, e.g., a resistor, a connector, a chassis, whatever. Note that each level of assembly can be a mixture of all of these types.

Decomposition is the black magic in system design. Do you split things into, say, four or seven subsystems? There is no *right* answer, but the best advice is to keep interfaces as simple as possible. The trick is to minimize the amount of information that one has to pass among subsystems. In this era of cheap computing, try to make each subsystem as self-contained and self-sufficient as possible. Resist the temptation to pass along information just because you can.

Most of the showstopper development issues that subsequently surface will be due to a failure to understand fully or, worse, to agreements to disagree on these internal interfaces. Bugs are typically fixed in days, but interface incompatibilities take weeks or months to resolve. Managing interfaces between subsystems commonly uses dedicated design documentation. Historically called Interface Control Drawings (ICDs), their content is often managed by Interface Control Working Groups (ICWG's) made up of participants from both sides of the interface as well as usually some representatives responsible for the overall system. Most commercial projects do not spend enough time on this activity. The extreme formalism and dedicated staff of aerospace is probably not warranted, but appropriate definition and documentation is essential.

Functional specifications are the criteria for subsequent design verification. This design verification is often called **"qualification"**. These functional specifications enable an independent party to develop qualification test plans and procedures including pass/fail criteria. Such is often required in parallel with the actual design implementation since test planning, fixtures, procedures, and software may be a non-trivial development within themselves. Further, the black box view of such tests invariably brings out missing or incomplete features overlooked when one just tests the integrity of a specific solution.

The top system-level functional specification is the criteria for formal design validation. Regulators invariably require such validation

by someone other than the system's developers. By definition, validation is a demonstration by a second party to confirm the objectives of a verification performed by the development team. While this seemingly duplicates the developer's verification at the system level, the difference in perspective, usually based on an independently developed test plan/procedure, is worthwhile.

Lower-level functional specifications are the basis for procurement of design services. While desirable as the basis for design verification, such specifications are mandatory if one is to procure a non-catalog design from an outside entity. Note that if one does not produce such lower level functional specifications for internally designed entities, one must instead perform the simultaneous verification of several unverified peers at some higher level of assembly that *does* have such a functional specification. If a design has any significant complexity, trying to resolve defects and errors among unverified components is quite time consuming and sometimes impossible due to ambiguities.

> **Lower-level specifications are also essential if reuse of the subsystem is anticipated**. If you are developing systems by tailoring somewhat standardized subsystems, you particularly need a detailed definition of what they currently do so you will know how to reasonably define and charge for any needed bells and whistles for each new customer application.

Product specifications are the basis for procurement of production copies of the qualified designs. These specifications fully define the requirements for production articles. As such, they are no longer a black box view but **describe the chosen solution** in detail. These may not need to be separate documents if the drawings and other technical data fully describe the characteristics needed to produce and verify. However, it is also common practice to collect the non-bill of material and non-construction information in a textual document.

> **Specifications that define the solution are what most engineers find comfortable to write,** probably because they are written after the fact when more is known. Unfortunately, such does not provide any guarantee that the real functional requirements have been met. It just describes what they built.
>
> **Product specifications are invariably written in terms of tolerances, whereas functional specifications are written as bounds**. For example, a product specification might say the item weighs 24 ± ¼ pound whereas the functional specification would say it needs to weigh less than or equal to 25 pounds.

Product Specifications are the basis for verifying the integrity of production articles. This production verification is often called **"acceptance"**. Note particularly that this is *not* re-verifying the design, but rather its continuing production execution. As such, acceptance inspections and tests are designed solely to verify errors in production: cold solder joints, mis-oriented or missing components, weak insulation, etc. For example, acceptance testing at end user operating environments may well be too benign to induce the stresses needed to weed out weak components and assembly shortcomings. However, as with qualification, if product specifications do not exist, one must defer the acceptance of an entity to some higher level of assembly that is specified and verifiable. Deferring such testing may lead to higher overall costs of production. One historical rule of thumb is that it costs a factor of four more to discover and fix a defect at the each higher level of assembly.

> **"Fail early" is a useful mantra to adopt.** There is often a tendency to defer substantial testing since it sounds like you would save money by not duplicating a test at each higher level of assembly. For example, one will encounter companies who did not want to pay for substantial supplier test fixtures and time. How they could then hold their suppliers accountable for quality is beyond me. This mantra is likewise applicable for qualification testing as well. The sooner you find a bug, the cheaper it is to fix.
>
> **You can save a lot of money by not duplicating functional tests per se as a part of acceptance.** Remember, your primary objective in an acceptance test is to find errors that are unique to this particular serial number. It is often reasonable to use selective functional tests to detect defects in production and assembly as such may very well be the most expedient screening mechanism, but the objective is different.

Verification & Validation

Verification can be by inspection, analysis, similarity, or test. What is important is that one confirms the integrity of the design and of the product. While qualification testing is common, it may be unnecessary if the design is very similar to another previously verified or if well established analysis techniques are applicable. Acceptance is usually by inspection or test.

Regardless of the method, documented evidence of the activity is essential. Prettiness is not the issue. Handwritten notes in an engineering notebook or memo for the record are perfectly adequate. However, compiling such evidence for regulatory audits may be more of a burden than producing them originally in a more organizable or fileable form. Said another way, do not over-promise the form of documentation, but rather focus on its organized retention and accessibility.

Reviews

Design reviews are one of the most common forms of verification by inspection. These do not have to be formal meetings with all stakeholders present in a single room. Simple peer reviews are much more common, such as a software code walkthrough or an engineer's check of a drawing made by another designer. Walk around "desk review and signoff" is also common. The main requirement for an activity to constitute a review is the involvement of at least one party who has no direct responsibility for the design under review. There is at least an implicit requirement that this independent reviewer is competent, typically an objective peer or a functional (not project) supervisor. In addition, some evidence of resulting action items (or the lack thereof) is minimally required. These can be as simple as annotations on a sign-off sheet. Meetings that are more complex will also typically involve minutes capturing any presented materials and summarizing the key discussions of the review. Nevertheless, in very complex programs, there are at least five formal reviews, sometimes called SDR, PDR, CDR, FCA, and PCA. (Note that this aerospace terminology has evolved, but the process basics are the same.)

> **If there is only one feature of aerospace system practice that you can adopt, it should be *design reviews***. The most notable results from reviews invariably arise more from differences in perspective than from simply detecting mistakes. Aerospace has the advantage of a culture of smart customers performing excruciatingly formal reviews. The real reviews were the internal dry runs, in order to make sure your development team was not embarrassed by these customer reviews. The dry runs were often rather brutal and demanding, but it was not personal. The main point is that these internal reviews invariably produced substantial observations. They are worth the effort. However, do not confuse these reviews with customer reviews where you are trying to prove the system will work. In these internal reviews, you are trying to prove that they will not.

The most difficult part of establishing meaningful design reviews is establishing the premise that this feedback is professional, not personal. Many commercial developers, particularly software programmers, just cannot accept this concept. They view themselves as the expert, so it is particularly egregious to have management involved. One may lose as much as ¼ of the staff when establishing this practice, even when the reviews are mainly by peers, but have no regrets or hesitation. If they cannot explain and defend their design, they will never be useful contributors to large-scale systems.

A military concept called "completed staff work" provides a sound basis for such reviews. One commonly encounters engineers mostly wanting to describe their chosen solution. The most effective question in a review is usually "why?" The idea behind completed staff work is that you should prepare 3 to 5 alternative solutions, evaluate their pros and cons, and explain your recommendation's rationale. There are three keys here: a.) more than one solution, b.) your recommendation, and c.) its rationale. When your bosses choose an alternative, it is invariably because of a difference in perspective, not that they did not listen or that you were wrong. The only time you should feel a bit embarrassed is if they come up with an alternative that you did not even consider.

A System Design Review's (SDR) objective is to concur on the system's top-level functional specification. Typically, conceptual designs and results from feasibility studies are also reviewed to develop confidence that at least one viable solution exists so that it is prudent to initiate preliminary design.

A Preliminary Design Review's (PDR) objective is to concur on the decomposed functional requirements. As the name implies, preliminary designs and/or the results of prototypes as well as initial risk management activities are also typically reviewed. However, one is only approving the hierarchy of functional specifications as to their appropriateness, consistency, and completeness. In effect, you are approving that it is prudent to begin detailed design activities.

Focusing a PDR onto the specifications, rather than onto drawings, Graphical User Interfaces (GUIs), and the like, will probably be the hardest culture shift in a commercial environment. If you thought writing those truly functional specifications was difficult, getting your customers to understand that those specifications are what they should be

controlling is even harder. However, you will both be the better for it. You will have more design leeway, and they will have more control over what they really should be controlling. Moreover, you will both have a legitimate basis for declaring victory.

A Critical Design Review's (CDR) objective is to concur on the design outputs: detailed design drawings, bills of materials, product specifications, test fixtures, software source code, and the like…everything needed to procure and produce articles representative of production that are suitable for use in qualification activities.

A Functional Configuration Audit's (FCA) objective is to concur on qualification. All evidence of the inspection, analysis, similarity, and test activities are methodically assessed to confirm that all functional requirements have been met. A traceability matrix is often used to document completeness, although any methodical process may be used to assure it is prudent to release the design outputs for volume production.

> **Traceability matrices are often impractical given today's software design practices**. In the old days, most design used something called "functional decomposition". The result was that you could indeed trace a single high-level function down into a single location in the software tree. One of many problems with this approach is that it leads to excessive (almost?) redundant code. Nowadays, there typically will be several low level functions distributed throughout the system needed to provide a single high-level response. A matrix that is attempting to make a simple two-dimensional mapping of a requirement to some low level test has lost its relevance.

A Production Configuration Audit's (PCA) objective is to concur on manufacturability. The suitability of procurement documents, production tools, work instructions, acceptance test procedures, and the like are confirmed to result in components, subassemblies, subsystems, and systems that are fully compliant and consistent with the design outputs that were previously qualified.

Regulatory entities, like the FDA, usually leave it to the discretion of management to determine the number and timing of formal design reviews. Typically, these would be specified as elements of each project plan. While it is theoretically possible to run a very simple project with no formal reviews, any project must somehow demonstrate that it has met the objectives of all five of the formal reviews cited above.

Analysis & Similarity

Technical analyses are a second broad class of design verification activities. All the classical types of engineering analyses may be involved: stress calculations, circuit timings, state diagrams, cost estimates, tolerance stack-ups, statistical assessment of clinical data, etc. When one's confidence in the accuracy and precision of the analysis method is combined with its predicted margin, often such analysis is adequately prudent verification. That is, no further testing is required.

***Risk* analysis is a special subclass of verification and is particularly important in medical devices.** One must methodically assess the product as to the likelihood and to the severity of occurrence for hazards and risks under all reasonably foreseeable circumstances, both for normal and unplanned usage. Typical tools include Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA).

For those risks deemed unacceptable, specific risk mitigation actions must be planned, executed, and verified. Except for the simpler projects that can incorporate these elements as part of the total project planning, separate risk mitigation plans and verification are usually provided to insure the requisite focus on safety related matters.

Compliance of the design outputs with company practices must also be verified. Examples include compliance with coding style standards, derating criteria, drawing style and dimensioning practices, software design practices that assure extensibility and serviceability, etc. These would typically be invoked by inference and verified by inspection; these are not usually cited explicitly in functional specifications. Regardless, one must be careful to invoke them explicitly in design procurements.

> **Well-documented design practices are particularly helpful in guiding younger or newer staff**. It is very worthwhile to try to capture some of the folklore and experience of your company. Lessons learned cannot be leveraged unless captured and taught. Later chapters include several examples.

Qualification may also be simply determined by an assessment of similarity to an existing qualified design. Typically by inspection and analysis, one must confirm both the technical similarity of the two designs and the qualified and satisfactory usage status of the existing design.

While common in aerospace, **qualifying by similarity is not that common commercially**. Such can save a lot of time and money.

Test

Test is often erroneously used as a synonym for verification. In fact, testing is only mandatory for validation. Verification is often more effectively and efficiently performed by inspection, analysis, or similarity. For example, it is usually more difficult, if not impossible, to cause hardware and software to represent the limits of tolerance or fault conditions. As such, practical considerations invariably lead to a combination of tests, each of which only addresses a subset of the environmental and functional requirements. Normally, testing is a last resort that only addresses those specific issues where one lacks confidence in the relevance or thoroughness of the other verification methods.

Test to a plan, not just until you are tired. Those functional specifications discussed earlier provide the missing basis for the test plan. The problem with just allowing the developers to test their own design is not that they are prone to cheat, but rather that they are meticulous in testing for all the conditions that they made provisions for in their design… but not necessarily the underlying requirements.

Corner coverage requires balance. Besides the practical difficulty in forcing good hardware to its theoretical tolerance limits, one must also be careful not to simultaneously force all inputs to their extremes. Otherwise, you are testing for a set of circumstances that will be both highly unlikely to ever occur and very expensive to create. Just make sure the combinations of variations are reasonable. Said another way, test for so-called three-sigma cases, not nine-sigma.

Automated test tools, particularly for GUIs, are worth the effort. These tools facilitate the thoroughness needed, particularly for exception conditions. Your staff can concentrate on adding exceptions, rather than boringly, and thus sometimes sloppily, repeating inputs day after day.

Testing with emulators has its limits. When a team has gone to the extra effort to develop or use emulators of their peers, it can also be difficult to get them to let go and start interfacing to the real thing. Timing issues and real data dynamics will have unanticipated consequences. Exception

conditions are also difficult because they often must be emulated, since it is difficult to force real hardware to its limit conditions, but emulations will invariably also be somewhat incomplete.

You will always regret trying to test more than one untested item at a time. Finger pointing arises to a fine art when neither party can prove that their component meets its requirements, particularly with respect to interfaces.

Oversights in test planning are only a problem if you do not learn from them. Despite your best efforts, you will still have defects and bugs escape your factory into the field. No one is omniscient enough to anticipate all exception conditions. Just make sure that every bug found in the field leads to a corresponding change in your test procedures. That is, not only fix the bug, but also fix the test that let the bug escape.

Test to break it, not demonstrate it. Most customer-witnessed testing would be more appropriately labeled as demonstrations, except for the social stigma that would accrue. However, the precursor internal tests should be both ruthless and thorough. Said another way, the demonstrations show that one has met the customer-specified requirements, while your internal testing should be focused on exception and off-nominal conditions to surface more subtle failure modes and mechanisms.

Test early and every step of the way. Where feasible, one should test at each level of assembly, working your way up from the bottom to the top system level. At each level of assembly, over time, one likewise works up the organizational structure. For example, the individual developer or assembler performs some type of unit testing before passing it on, usually to a device level test, then to an Engineering integration test, and eventually to an independent test group. In turn, as noted earlier, validation is then simply an independent test at the system level by yet another independent group.

Keep Engineering responsible for the initial integration testing, at least of complex systems. There is probably no better learning experience for all engineers, young or old. They also need to remain accountable for making their designs work. Unfortunately, some like to try to leave this supposed clean-up activity to others. They will never learn to detect and accommodate exception conditions without this experience. One means to enforce this is by requiring Engineering budgets for original design to include passing these initial Engineering integration tests. That is, they cannot begin to spend the typical bug fixing or sustaining engineering budgetary accounts until passing this

milestone. One means of lessening their objections is to give them a free pass on any bugs that they find and fix at this stage, i.e., do not start counting bugs in your publicized metrics until they handover their design to an independent test group.

To reemphasize, testing usually should be a last resort and should focus on exception conditions. Developers invariably focus on proving that their system can indeed work. Unfortunately, it is often just under their point design conditions. Most of the real world problems, and, therefore typically more than half of most production software, relates to gracefully handling error and off-nominal design conditions.

Barbie® Dolls

Most product-based capital goods industries are Barbie® doll businesses. That is, you get what ever you can for the doll, but you make all your profits from the clothes. In capital goods, the "clothes" are replacement parts and service contracts. As such, development activities should also focus on minimizing the costs associated with servicing a system. With today's technology, it is relatively inexpensive to capture error codes in non-volatile memory so that your service staff can find and pass on what the device thought was wrong as it was dying. Otherwise, you will be faced with the historical issue of could not duplicates (CNDs), retest O.K.'s (RTOKs), and no trouble found (NTFs) back from your field staff as they repaired by remove and replace (R&R). In fairness, R&R is about all that they can do without good error capture and built-in diagnostics.

> **One should rarely buy a hardware maintenance agreement**. As long as there are no moving parts in the product, most products today are very reliable. You can reasonably gamble and only buy hardware maintenance agreements when it becomes obvious that you bought a lemon, or being more polite, an overly complex piece of hardware. Such commonly occurs when one is an early adopter. Otherwise, just pay time and material for Service. While suppliers will often contend that they cannot guarantee response times to non-contract buyers, they will invariably respond as quickly as they can… which is all they will do even with a contract.
>
> **At least you get new features with a *software* maintenance contract**. Yes, you also get the bug fixes that perhaps you were due anyway, but the new features are usually worthwhile. If you find your supplier fails to add substantial new

functionality, then drop their contract, but also do not expect them to answer their phone when you call with a problem. Their first words will invariably be, "Which version are you running? Oh, then please bring yourself current and call back if the problem still exists."

Many vendors use the defacto industry-pricing model of about 20% of the software's list price per year. In fact, the primary reason for software list prices even to exist is to set the price of the annual maintenance fee. You will find that almost every hardware vendor will discount his or her original associated software purchase price to whatever is needed to be the winning bidder, including giving it away for free. However, they will rarely negotiate their software maintenance prices since, unlike hardware, these are rarely cash cows. As noted elsewhere, the good news about software is that you can change it. The bad news is that the market makes you change it to stay competitive.

Consider offering to buy "used" hardware if it is the end of a quarter, or, better yet, the end of the supplier's fiscal year. We were actually delivered new hardware almost every time. This appears to be simply a ploy by suppliers to bypass their "favored nation" purchasing agreements with large customers. Those agreements typically have the supplier promising never to sell the same product for less to another customer without offering a credit to the "favored" customer.

Change Management

If you are in the system development business, the Barbie® doll's clothes are contract changes. With any reasonable complexity, there is little historical precedent for assuming your basic contract will be profitable. It is not an issue of whether you will overrun Engineering, only about how much. Details will follow later in the discussion of earned value. So, how does one do profitable development? The answer is in your contract's changes clause.

Detailed original specifications are the key to changes. Remember, you have to have something specific to change from.

Usually, a superior document prevails when addressing conflicts, but a *subordinate document prevails regarding interpretation*. That is why we stressed the importance of including as much detail as

possible in subordinate functional specifications. For example, if your lower level specification says your system has such and such behavior, when your customer comes back with a request for doing it some other way that is nicer or better or whatever, as long as your specified method is *a* way that meets the top-level specification, you have a legitimate claim for a change.

Be fair, but do not be a pushover. We are not advocating that you "get well with changes" as the saying goes, but we are also saying that one should *not* feel guilty about making the customer pay for his feature creep. There *will* be feature creep.

Third Time's the Charm

Experience suggests that it takes three attempts to get a product right, particularly if it is software intensive. Many do not realize that there was a Windows version 1 and a Windows v2. All that most are likely to remember is Windows v3.1. Digital Research's GEM was originally much better and had most of the initial market share for graphical user interfaces (GUIs) on PCs. However, Microsoft had the resources (admittedly because of their cash cow, MS-DOS) to listen to the marketplace and evolve the product to a market winner. Moreover, despite all the latter day whining, Microsoft's dominance of the word processor and spreadsheet market was indeed because they created a better mousetrap. In the early days, many bought Apple Macintosh's in order to get access to Microsoft's new What You See Is What You Get (WYSIWYG) Word and Excel applications.

The first version of anything rarely involves inputs from real customers. They are primarily based either on wish lists from the company's Marketing department or are some bootleg demo out of Engineering that Marketing thinks must be ready for production as it understandably is in everyone's interest to get something to market quickly.

Strongly fend off any attempt to put a demo into production, even as an initial product. Demos are just that, particularly if they were developed for a big industry trade show. Primarily they lack the exception handling code needed, but unfortunately, such is typically much more than half of the code in a real product.

First products primarily get everyone useful feedback from real users. It is not just about the GUIs, but mainly about what features are really used and need enhancing and which are bells and whistles that can be allowed to wither on the vine for a while. In addition, you will be

inundated with exception conditions that your developers never considered. The first hardware designs also are invariably not cost effective to produce, from both a manufacturability and testability perspective. They did not realize it, but these first customers were really just beta testers.

Mainly, the second products are producible, profitable, and reliable. These second products then get feedback from enough end users to create a third, robust set of features that can dominate a market, assuming good execution. They also usually include first attempts at user configurability to try to get the developers out of the expense of customizations, or to assuage customer pleas.

> As an aside, when developing these second designs, one will invariably find that the dominant effect on manufacturing costs is piece parts count. Use manufacturing technologies that minimize them. The dominant effect on electronic reliability is invariably parts' temperature, since reliability is a function of the fourth power of junction temperature. Derate your parts, and run them cold.

Finally, if you have been listening to customers, the third time is a market winner.

Incumbents know their marketplace, so they can skip steps. While it would be nice to think that they only needed one step, their first attempts are still often not very producible, because they tend to be dominated by engineering, and/or they tend to lack configurability, using the excuse of a rush to market.

Incumbents know the myriad exception conditions experienced in their applications. More than the functionality seen by end-users, these exceptions are the unique lessons learned that they could leverage to maintain their market edge.

Incumbents disappeared from the market mainly because they could not let go of building specialized hardware. The problem was not being a Smith-Corona failing to recognize the advent of word processors that would displace their typewriters. The problem was being a Wang or a Prime who would not introduce versions of their application running on a PC until it was too late. They, and many others of their ilk, had dominant market share, but they just never learned to compete with themselves. If you do not learn to compete with yourself, then someone else will.

The large dashed arrow in Figure 1.1 recognizes the inherent iterative loop in the overall development process just discussed. Therefore, our "linear" or "waterfall" process was implicitly iterative, assuming the first version was enough of a commercial success to justify another loop, based on feedback from the first pass through the design process. The process is considered a waterfall because, conceptually, each step is completed before the next is begun. In practice, there is always some overlap and even iteration backwards as needed, e.g., when architectural problems are encountered.

More elaborate system engineering process models were evolving by the eighties, such as the "spiral" developed by Boehm and the "Vee" developed by Forsberg and Mooz. These elaborations tend to primarily apply to systems of systems, typified by aerospace and defense, where multiple iterations occur over many years before a "production" article emerges. Similar iterative design schemes have arisen in the software development community. As an admitted overstatement, these schemes seem to advocate that requirements are so hard to determine that one should just make a reasonable first cut and then iterate your way to success. In effect, they seem to rationalize a build-and-redesign, rather than a design-driven-by-requirements process.

Regardless, while most projects tend to implement in phases, the author has never seen anyone successfully architect and design in phases. To be applicable to commercial systems, one then will have to be very cautious of these other development strategies to assure that a sellable, useful product will result from each iteration. Again, following the theme of staying focused on the basics, the simple waterfall model presented herein will invariably suffice as a laudable objective.

In conclusion, the top-level functional requirements specification, the design outputs needed to support production, evidence of validation, and evidence of risk management are about the only mandatory items for any project, large or small. Each project manager has the prerogative to define which of these elements are suitable to combine for their specific development. For example, SDRs are often combined with PDRs for routine projects. Regardless, **all of these objectives must be demonstrably met. It is only their form that is subject to management judgment**.

Chapter 2 Program Planning 101

The crime of management is not being late or overrun; it is not knowing. According to Merriam-Webster, to manage is "to handle or direct with a degree of skill: as a: to make and keep compliant". Thus, to manage, one has to have a plan, i.e., something to which one can attempt to keep things compliant.

Most projects of any substance warrant a legitimate schedule that is appropriately resourced. Nowadays, most teams use Microsoft Project® as their planning tool. There are other tools around; Primavera® is notable for its more thorough tools for dealing with shared resources in a multi-project environment, but, as with other areas, Microsoft has evolved sufficient functionality into Project® that it drove most historical but higher-price providers out of the market.

This chapter assumes some familiarity with the terminology of formal planning tools. However, we will digress for a while to establish some of the basic concepts for those that have so far avoided this practice. The most useful view of a schedule is called a tracking Gantt. Figure 2.1 shows an example of a simple plan.

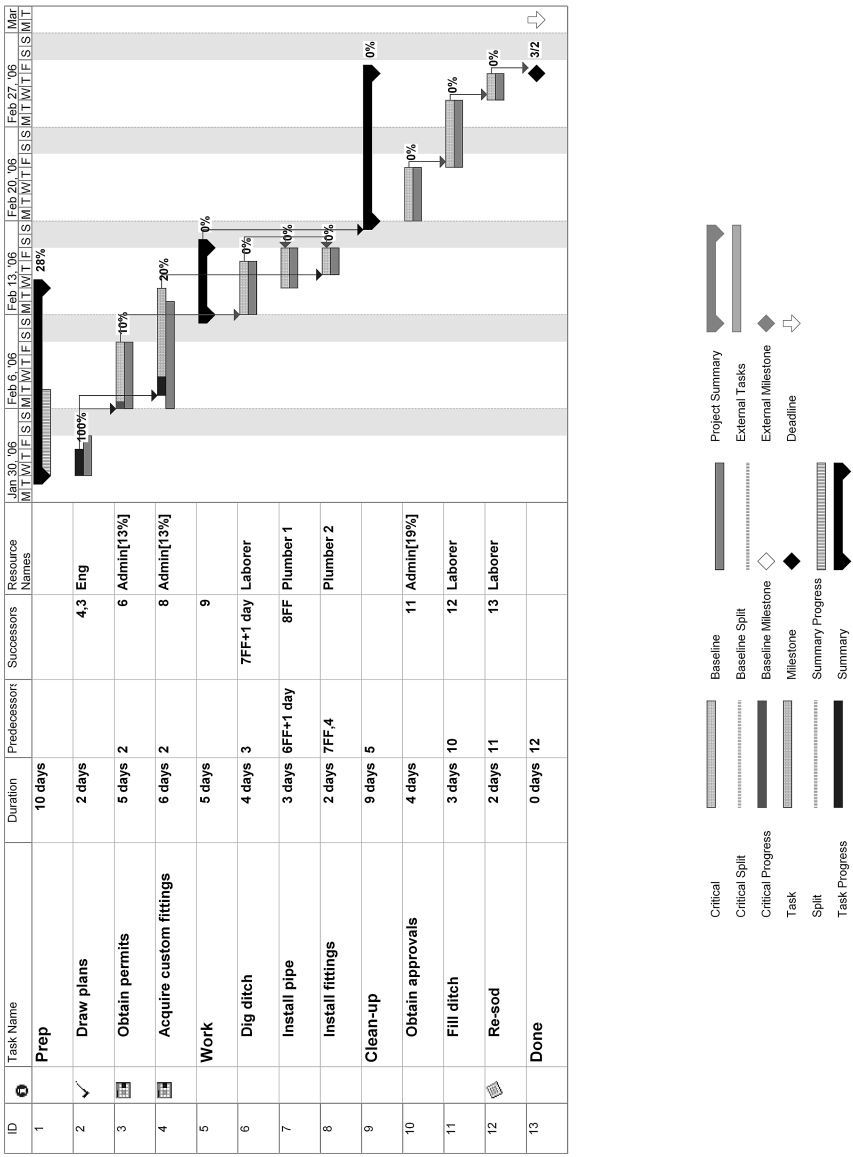

Figure 2.1 Ditch Digging Project Plan

Let us define some basic terminology. Summary tasks are just that; their span encompasses their subordinate tasks. In the old days, the subordinates were often called “hammocks”, hopefully for self-evident reasons. Tasks are linked in various ways: finish-to-start (FS, which is the most common but often unrealistic), start-to-start (SS, typically with a lag that you define), or finish-to-finish (FF, again usually with some lag). A task is “critical” if it lies on the path through the schedule that has the longest overall duration. Split’s occur typically when one has to pull some resource off a task so that no work can be done for some interval. Milestones are events with no duration or resources, typically used to report to senior management. Deadlines, unlike milestones, do not move with changes in your plan. A baseline is just a snapshot of your original plan to which you can compare your actual performance.

One *manages* starts, not finishes. You react to finishes. Having a plan will not mean that your need to react disappears, but your fire drills will at least be different, and most should not be a surprise. As such, when monitoring a program, pay much more attention to how one is achieving planned starts since you will rarely see tasks that take less time than originally planned. By focusing on starts, you are achieving several weeks of a head start on any associated difficulties.

In the example, even though the administrator started getting permits per the plan, a day was actually wasted since the engineer finished early. The administrator also was late ordering the custom fittings, but that does not affect the overall schedule because that task is not “critical”.

Use varying granularity of detail. Constant granularity plans are invariably wasted efforts. Future fine-grain details are overtaken by events (OBE). Coarse near-term plans preclude detecting and resolving resource conflicts. Some guidelines include:

Table 2.1 Project Planning Granularity

Interval	Resources	Granularity
now + 3 months	name	man-days (md)
3-9 months	skill	man-weeks (mw)
remainder	dept	man-months (mm)

…where quantities should range from two to twenty, but typically from five to ten.

Customers often demand constant granularity, but push back hard. You will notice in the example that the summary tasks are not resourced since they have been expanded. However, in the early phases of a larger project, the plan would mostly consist of such summary tasks (that is, these are mostly not yet expanded into finer granularity) which have been resourced grossly per the table above.

Create a "rolling wave" of detail. This simply means that once a month or so, you will need to add another month's worth of detail for the near- and for the mid-term. These are only guidelines. Depending on what you are decomposing, it may make sense to do a couple of months on some jobs and to wait on others. Judgment is still required.

First, re-plan uncompleted work. Unfortunately, you will invariably be resource-bound in the near-term. Further, at least some of the incomplete work will invariably remain on your critical path. Thus, you will need to address this work first.

Re-plan with no-earlier-than (NET) start dates and/or "resource" links. Both Microsoft Project® and Primavera® will automatically reschedule un-started work to start from today. More commonly, you will be resource bound. The easiest way to handle near-term resource conflicts is to artificially link tasks that you have assigned to the same resource. Then they will all re-schedule themselves.

In the example, the finish-to-start link between filling the ditch and re-sod is such a resource link; it is the result of having only one laborer available. If you had more resources, this really should be a finish-to-finish link with a small lag, for example, as we did when we added a second plumber to be able to install the fittings in parallel with the first plumber who was still laying pipe.

Do not schedule more than two simultaneous tasks per resource. Most tasks have times where one is waiting on others, or for a better idea, or whatever. Thus, most individuals can handle two tasks at once. A few can handle more, and some can only handle one. Regardless, you should seriously question any plan that routinely has the same resources doing more than two tasks at a time.

***Never, never ever* hit the automatic resource-leveling button**. Only you or your managers know which staff are interchangeable, just how productive each is, etc. This is now

less critical since the latest planning software versions have an undo button, but it remains good advice.

Settle for leveling your resources so that the person-hours per month start with a one. While ideally you would shoot for about 150 person-hours each month, you will find most of the benefit results when you get everyone somewhere between 101 and 199. Let the 90/10 rule work for you, i.e., you will invariably get 90% of a result for 10% of the effort. You will find that this is difficult enough, and you need to be spending most of your time on making the plan happen, not just making the plan.

Usually, decompose using a hammock. Hammocked tasks are simply a set of subtasks (at least one of) whose start is linked to the parent's start and which ends with (at least one) link to the finish of the parent. Doing so allows you to retain visibility into your baseline plan. You can collapse the sub-tasks and the original plan appears unchanged, even though there invariably are some other useful links coming out of the sub-tasks that were not present previously.

Microsoft Project® calls these hammock parents "summaries". You will notice in the example that we actually start digging the ditch before the summary "prep" task is complete. That is because Project® automatically makes a finish-to-finish link from the latest sub-task to the summary. Elsewhere we physically linked "clean-up" with "work". Otherwise, we would have to stay on top of which of the "work" sub-tasks was latest.

Beware of the student syndrome. Many staff members likely *are* overtaxed and will invariably delay working earnestly on a specific task until the deadline is imminent. This is one reason why 20 person-day tasks should be the exception. Further, long tasks are inappropriate for the 50-50 rule discussed later.

That is probably why we have already lost a day of schedule in the example. It means you will always be facing the quandary of balancing realism with challenge, as it is easy for someone to prove that he or she can be late.

"Vertical waterfalls" are particularly suspicious. These arise when several linked tasks are supposedly going to complete on almost the same day. This rarely happens, since you are rather good if the first actually occurs. All the rest will dribble to the right.

Throw every resource you can at the critical path. Nothing dominates cost as much as overall project duration. The fact is that work for the entire team will invariably expand to fill the time allowed, so the old saw is true…**time is money**.

> **Let us say this again, throw every resource you can at the critical path**.

Be careful to start as many non-critical tasks as you can. Avoid falling into the trap of starting jobs as late as possible. Forget that such a start-date type even exists. One or more of them will also get in trouble and become critical as well.

> There are several date constraint types: no earlier than, no later than, as early as possible, as late as possible, must occur on. It is strongly recommended that you also avoid the latter. It is fine to set a deadline that will show on the plan, but deadlines are not constraints, just objectives.

Tasks should mainly be expressed in terms of outputs, products, or functions…not skills or phasing. These can be drawings, GUI screens, PC net lists, specifications, design reviews, etc.

> A common, but essentially untrackable, engineering task list will say things like conceptual design, preliminary design, detailed design, etc. Those managers are praying that someone will let them get away with managing a level-of-effort, say, six EE's for the month of June.

There are legitimate levels of effort (LOE), but often they just mask an inability to plan.

Other Direct Costs (ODC) need to be scheduled as well. Do not forget design tools and long lead items. Material and travel are often rather spiky in demands, not LOE.

The 90/10 rule applies to planning also. Do not polish the apple. You *will* get 90% of the result with 10% of the effort. Just make sure you have reconciled your main resource conflicts and have applied all the resources practical onto your critical path.

Nothing said so far is changing your baseline program plan. That is, you will be re-estimating when you believe tasks and hammocked sub-tasks will occur, but you are not rescheduling when they were planned in the baseline. Typically, programs may be so overtaken by events that a semi- or annual re-baselining is prudent.

Good plans minimize finish-to-start links. Few things in the real world have a finish-to-start relationship. Start-to-start and finish-to-finish links with lags are much more realistic and will radically reduce an overall schedule. As in our example, you do not have to finish digging a ditch before you start laying pipe, although you cannot finish until you do. In engineering, staff rarely waits to start a detail design until a specification is approved or a preliminary design review is held. However, it surely would be good practice to have at least a draft specification available before proceeding and a signed specification before release to manufacture.

Good plans have many vertical lines. Common practice is mostly horizontal, excruciating detail in long strings of tasks linked finish to start. These invariably represent a functional department detailing out internal tasks over which they already have maximum visibility and control. Vertical lines commonly represent dependencies between groups, which is where one invariably loses weeks on a schedule, versus the days lost on horizontal tasks. Particularly as you move to engineering that is more concurrent, these vertical lines indicate commitments and/or opportunities for feedback. Further, vertical dependencies to customer actions/events are invariably the basis for legitimate claims… *if* reasonably documented (say, in your program plan?).

Project plans should start with proposal plans. Perhaps a bit of a digression, but a program's resource estimates and schedules are much more believable in a proposal if they actually are linked together using a program-planning tool. It is very questionable how anyone could have confidence in their proposal estimates without time phasing their proposed resources and thus defining their schedules.

Proposal plans do not need unbelievable detail. Using our earlier groundrules, one would expect most proposal tasks and events to be in terms of person-months, not person-hours, days, or even weeks. Tens of pages of spreadsheet detail that is not time-phased or linked to a defendable basis-of-estimate (BOE) seem specious. Where feasible, one can use prior project actuals and some scaling rules for most new development proposals. Some companies get very fancy using parametric estimating tools, but those are usually more amenable for estimating factory and service tasks.

Engineering estimates should mostly be scaled from prior experience. For example, product design estimates can start with a drawing tree from the most similar, recent product. Each element of that tree is then marked up with "not applicable", essentially "usable", needs "modification", as well as identifying titles/descriptions for totally "new"

drawings. Then one uses person-hours per drawing category factor for each of the “tweak”, “modify”, and “new” drawings to capture the bulk of the mechanical design costs (not just the drafting time). That is, you are scaling your actual historical cost experience as the basis of estimating your new costs. Similarly, one can take all the computer screen-shots from a similar prior job; mark up and identify those needing “tweak”, “modify”, and “new”; and then use a person-hour per screen to define the specification, coding, test, and integration of new software. (Note that we are not proposing lines of code for the software BOE. History has shown it to be an unsuitable measure since it is more a function of style, than function.) Even if one cannot base the category person-hour factors on detailed, auditable history, these basis-of-estimate discussions will at least be on a much more meaningful level than purely judgmental extraneous detail.

The remainder can be factored from these direct estimates or are LOE. Most major development activities have some meaningful product indicators: person-hours per drawing, per GUI screen, per whatever. Others may legitimately be LOE’s, such as program managers (although not necessarily full time for the entire program period). As an example of the remainder, it is commonly found that configuration management was xx percentage of hardware engineering and yy percentage of software.

> As a real world example, Figure 2.2 illustrates the excellent correlation between a Systems Engineering group, labeled non-variant specific (NVS), and the underlying Engineering groups, labeled variant specific (VS) that they supported. VS groups would include mechanical design, electrical design, software programming, etc., that is, groups that have many discrete, clearly defined engineering products. The data compares the person-hours expended each month by the VS and NVS groups over a 5-year period. In this case, the fixed-cost term, the LOE, is seen to be 900 hours per month with a variable factor of 56%. So this group would be estimated as an LOE plus 56% of the loading in your plan for the directly estimated Engineering groups that produce drawings, software, etc.
>
> The R2 = 0.83 shown on Figure 2.2 is more correctly σ^2 which is the correlation coefficient for the fit. Anything above 0.5 is often usable, with something above 0.8 being a good practical fit. Experience also suggests that you will have enough cost data scatter that you should settle for simple curve fits. One should mainly try linear fits on either linear, log, or semi-log axes. This type of cost correlation has been found to estimate

well for technical support groups like configuration management, drawing check, drawing release, etc.

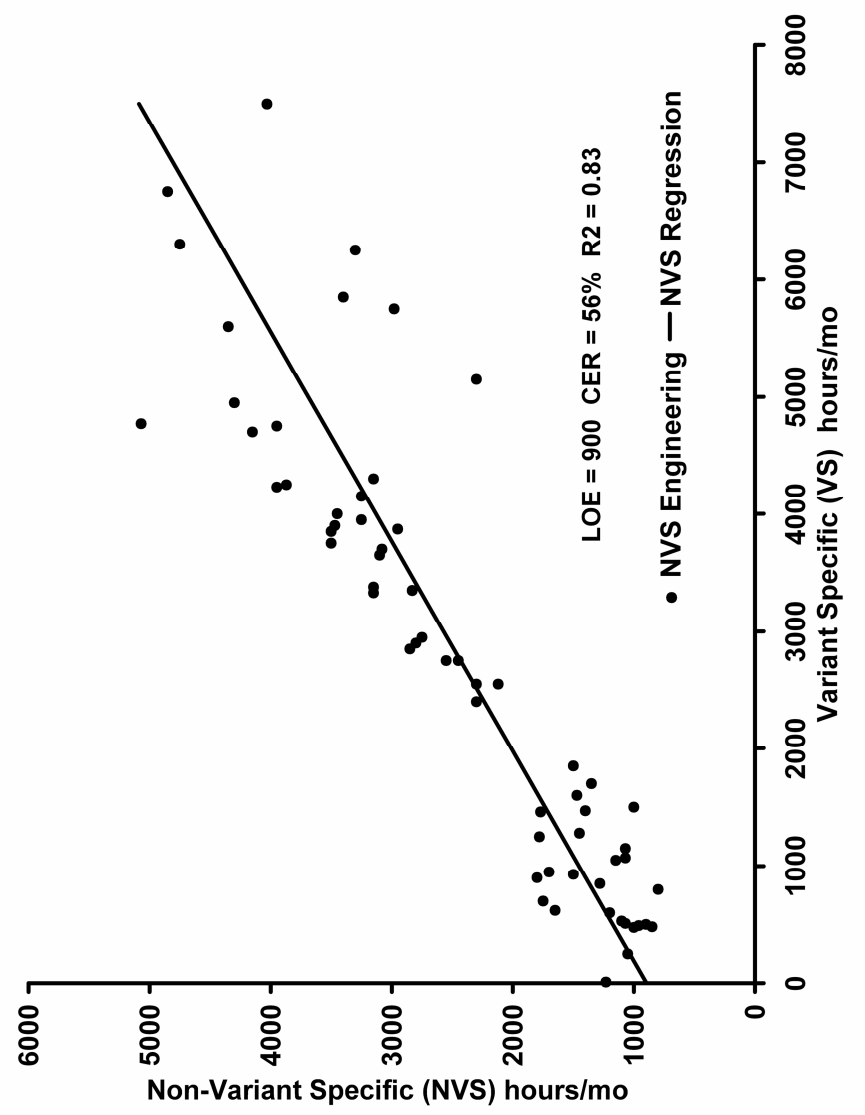

Figure 2.2 Estimating with Factors

These are "will cost" estimates, not "should cost", and that is just fine. Back in the sixties and seventies, the U. S. government procurement staff attempted to determine what something "should cost". Supposedly, this contrived but magically efficient contractor could be used as a benchmark to judge bid submissions. Having worked on both sides of that negotiating table, both the contractors and government should be quite happy with solid estimates for what something "will cost". The competitive process itself will drive down the price more than sufficiently. If anything, contractors will kid themselves regarding their costs in order to win a bid. The only place "should cost" should enter the picture is if the procurement is with a sole source. Even then, just use the bidder's own history to determine his "will cost" using the methods outlined above.

Noah's Principle & Earned Value

Predicting rain doesn't count, building arks does. When you are called on to present your status, remember it is *not* sufficient to be a reporter. What your bosses are looking for is a summary of your proposed corrective actions to recover or, at least, to reduce the hemorrhaging.

"Earned value" is a key management tool. Unfortunately, it requires a rather detailed project cost accounting system that is often not available in commercial companies. The idea is that you compare your cost and schedule for actually doing the work with your baseline plan. Most companies do not have a mechanism to collect costs on a task-by-task basis, although some do so at various summary levels. By the way, that is another reason to use the Microsoft Project® summary task feature. Earned value lets us distinguish true status. That is, not just what was spent, but what you have to show for it. As such, it is well worth the effort to work with your finance department to get access to whatever information can be made relevant.

Simply comparing expenses-to-date with budget-to-date can be grossly misleading. In earned value terminology, you would be comparing the actual cost of the work performed (ACWP) to the budgeted cost of the work scheduled (BCWS). Hopefully, restating the issue this way makes the problem apparent, i.e., you are comparing an apple to an orange. Somehow, you need to know how much work actually was done, that is, you need a third piece of information called the budgeted cost of the work performed (BCWP).

> Otherwise, you could have a manager who is underrunning his/her budget (ACWP < BCWS) because he/she is far behind

schedule (BCWP << BCWS), but whose productivity is terrible (ACWP >> BCWP). Likewise, you could have another manager, who is way ahead of schedule (BCWP >> BCWS) and is very productive (ACWP < BCWP), but is likely overrunning his budget (ACWP > BCWS). To uncouple productivity issues from schedule, some indicators have been found helpful.

There are two key indicators from earned value calculations: CPI and SPI. Cost Performance Indicator (CPI) is the ratio of the budgeted cost of the work performed (BCWP) to the actual cost of the work performed (ACWP). Schedule Performance Indicator (SPI) is the ratio of the budgeted cost of the work performed (BCWP) to the budgeted cost of the work scheduled (BCWS). CPI is what one should mainly care about because SPI shortfalls can usually be made up just by adding resources.

The budgeted cost of the work performed (BCWP) is unfortunately difficult to obtain from most commercial enterprise financial systems. However, it remains the key to earned value, so do not give up so easily. You will invariably have to settle for reconciliation at fairly high levels, e.g., at the department/skill levels where the existing financial systems track the actual costs of the work performed (ACWP) and the budgeted cost of the work scheduled (BCWS). You can obtain BCWP from your planning software, e.g., Project® or Primavera® or the like. Formal certified earned value systems require that these all be linked in an auditable manner, but most commercial companies just will not bother to do so. However, they will usually allow costs and budgets to be tracked by major project. You can calculate practical indicators from this existing data, which will be invaluable in assessing your true status. Remember, the crime of management is not overrunning, it is not knowing where you really stand.

You will rarely find an Engineering CPI greater than one. Most good programs will overrun with a CPI in the 0.9-ish range, with the typical complex project running in the 0.8 range, and not just a few poor programs with a CPI of 0.5 or less. (For the record, projects that the author managed were mostly good, with a few typical.) Figure 2.3 shows the likelihood of achieving a given CPI based on 81 major defense programs in the period 1950-1980. This is a recast version of the work by Norm Augustine in Augustine's Laws, published by the AIAA in 1997 as his Figure 48 on page 247 of his 6th edition. It includes both development and production costs corrected for quantity changes and inflation. Since production cost estimates are invariably no worse than development costs, this chart still provides a useful indicator for

development and is quite consistent with the author's personal experience. This does not mean that one should accept this performance as immutable, but it is indicative of what you will face in the real world. These programs were run and executed by good professionals using the best tools at the time. No panacea has been found in the meantime. In fact, due to the prevalence of software nowadays, one could argue that your chances are worse.

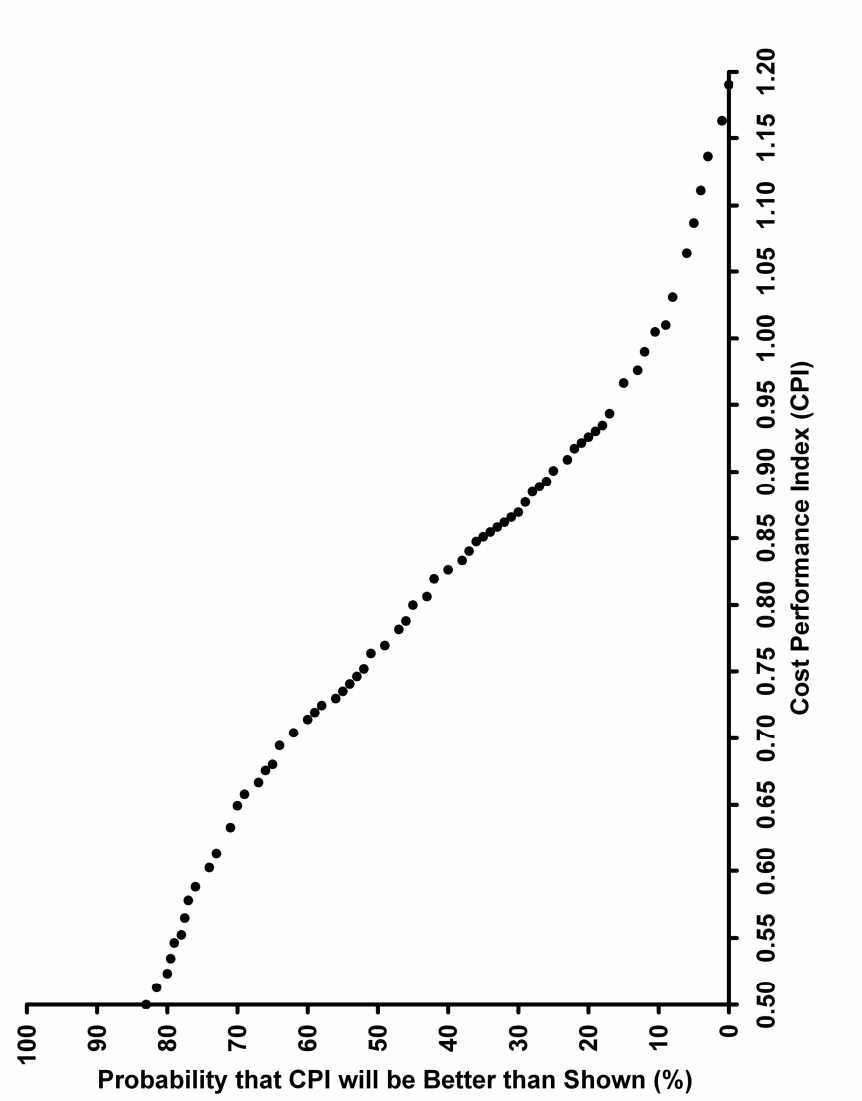

Figure 2.3 CPI Likelihood

Look for trends in incremental CPI and SPI, not cumulative. The problem is that once you are well into a program, cumulative indices are dominated by your entire history. As such, your recent performance is masked. Instead, plot monthly incremental values.

Use the 50/50 rule to claim earned value. Percent completion estimating is a mostly academic but time-consuming exercise in practice because the percent complete invariably equals the percent planned until 90%, where it sits until the task is done. Instead, simply claim 50% of the value when you start and the remaining 50% when you are complete. For some reason, staffs do not seem to kid themselves nearly as much regarding whether they have started or finished. Moreover, it is a lot less work to track. Finally, as long as you followed our task duration advice (nothing < 2 or > 20), it provides just as good of an overall view of progress. Besides, we want to motivate legitimate task starts for all the reasons discussed earlier.

Update your status weekly. Otherwise, the plan quickly becomes just a cartoon for management. Monthly is still adequate for substantial replanning like adding granularity, but start and finish status and resource assignment adjustments need to be timely.

Figures 2.4 to 2.7 are representative examples of the earned value status of a program. In Figure 2.4, cumulative person-hour results run from the bottom left to the top right. The original baseline plan (BCWS) is shown as a long-dashed line with no symbols. Actuals to date (ACWP) are shown as a solid line with no symbol; while the budgeted cost of the work performed (BCWP) to-date is shown as a short-dashed line with circular symbols. The current plan for the future is also shown as a solid line with no symbols since it does not overlap ACWP in time. Figure 2.5 similarly shows the cumulative and incremental CPI and SPI earned value indicators to date.

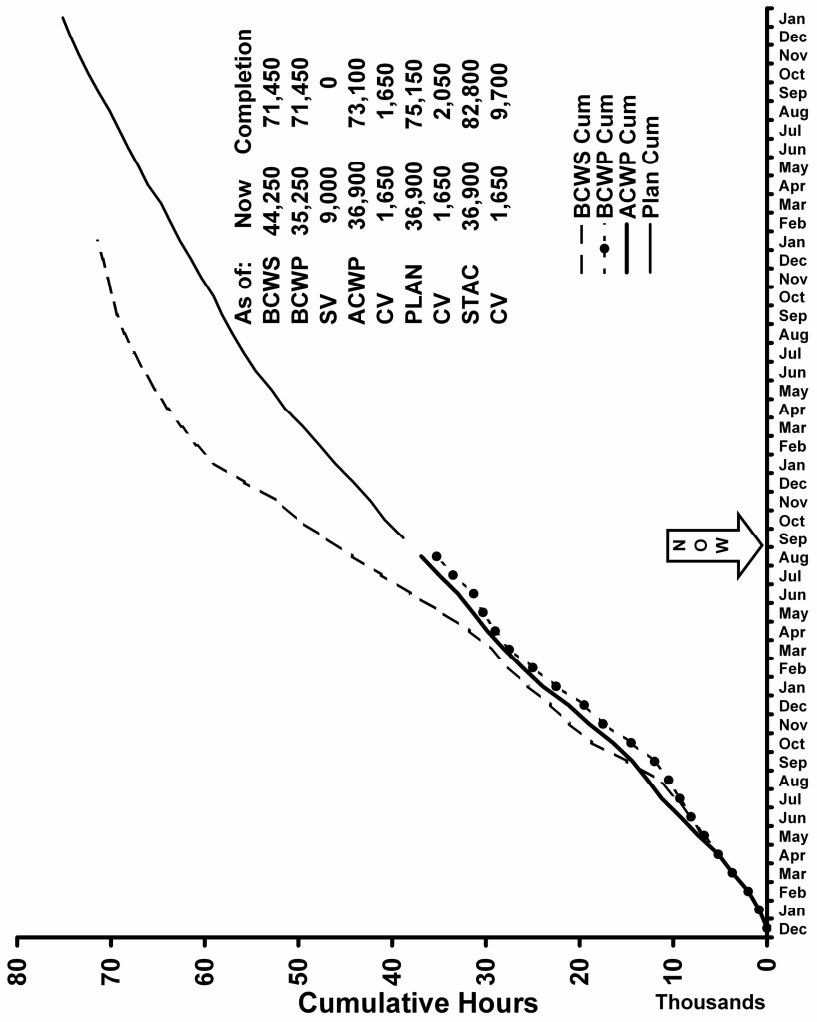

Figure 2.4 Cumulative Earned Value

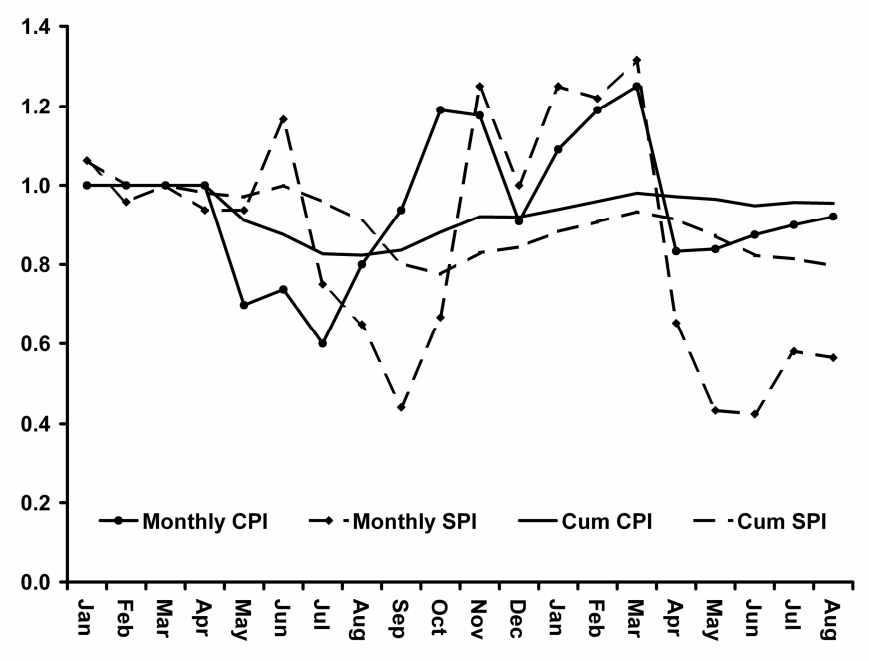

Figure 2.5 Earned Value Indices

The overall Cost Performance Index (CPI) is quite good, i.e., CPI = BCWP/ACWP = 35250/36900 = 0.96, but it has recently been worse, running in the 0.8 to 0.9 range. This manager is projecting a small overrun, i.e., a final CPI = 75150/73100 = 0.95, but he/she is unlikely to achieve that at the current pace. For completeness, the overall current Schedule Performance Index (SPI) = BCWP/BCWS = 35250/44250 = 0.80, although recently it has been running in the 0.5 range. As noted earlier, incremental indices are more indicative of recent performance.

Regardless, a low SPI is not necessarily a problem because it is apparent that the intent of the current plan is to delay the program. Such can occur to match the available work force resources, or often to accommodate some other project's delays, say if this project assumed the other was taking the lead on some key development. As a management survival hint, try to minimize being the first to do anything. Remember that in the system business, your main value added is making the combination of constituents perform a useful task, not inventing a new constituent. Getting the combination to work is hard enough. Unnecessary invention just adds unnecessary heartache.

Statistical Total at Completion (STAC) is probably a worst case for any program because it assumes that no one can do anything to improve his or her performance to date. It just extrapolates the current individual task CPI's to the end of the contract. It is only conservative in that it also assumes they will not have worse CPI's for individual tasks than currently. On the other hand, one will rarely see a program that ever did the remaining work for less than originally budgeted. Thus, one should be very suspicious of any manager who predicts they are going to get back on budget by magically doing the to-go work with an incremental CPI greater than 1.0. Said another way, one should be suspicious if the managers claimed that they could get well in the remaining interval. The best he or she is liable to do is hold to his current variance of 1650 hours. Alternatively, if they say there is nothing they can do but meet the statistical TAC, and let the overrun grow to 9700 hours, then he/she is not really trying.

In the example, even though the overall values look pretty good, there must be some tasks having substantial to-go effort with currently very poor CPI's, since the statistical total at completion (STAC) is much larger, projecting a final CPI of 71450/82800 = 0.85. If one were to guess, looking at the time phasing of the program, one would suspect they are just beginning testing in earnest, with probably a very poor CPI on very little test work to date. For example, they have probably only drafted test plans and are having difficulty getting them reviewed and completed. If so, one could almost believe that project manager, assuming he/she can keep their monthly CPI closer to 1.0.

For those who prefer absolute values instead of indices, the current cost variance (CV) is only 1650 hours in the hole, out of 44250. The manager is projecting that to worsen by another 400 hours for the remainder of the project. Similarly, the current schedule variance (SV) is 9000 hours in the hole. By definition, the schedule variance is zero at completion.

Figure 2.6 shows the incremental person-hours per month for various categories, using the same graphic styles as on the cumulative curves. These monthly values are much more indicative of how the project staffing levels compare to their plan. One would need extraordinary eyesight and mental acuity to be able to derive these values from the cumulative curves. In this example, it is clear that a conscious decision has been made to hold the staffing levels to about 2000 hours per month and to slip the schedule accordingly.

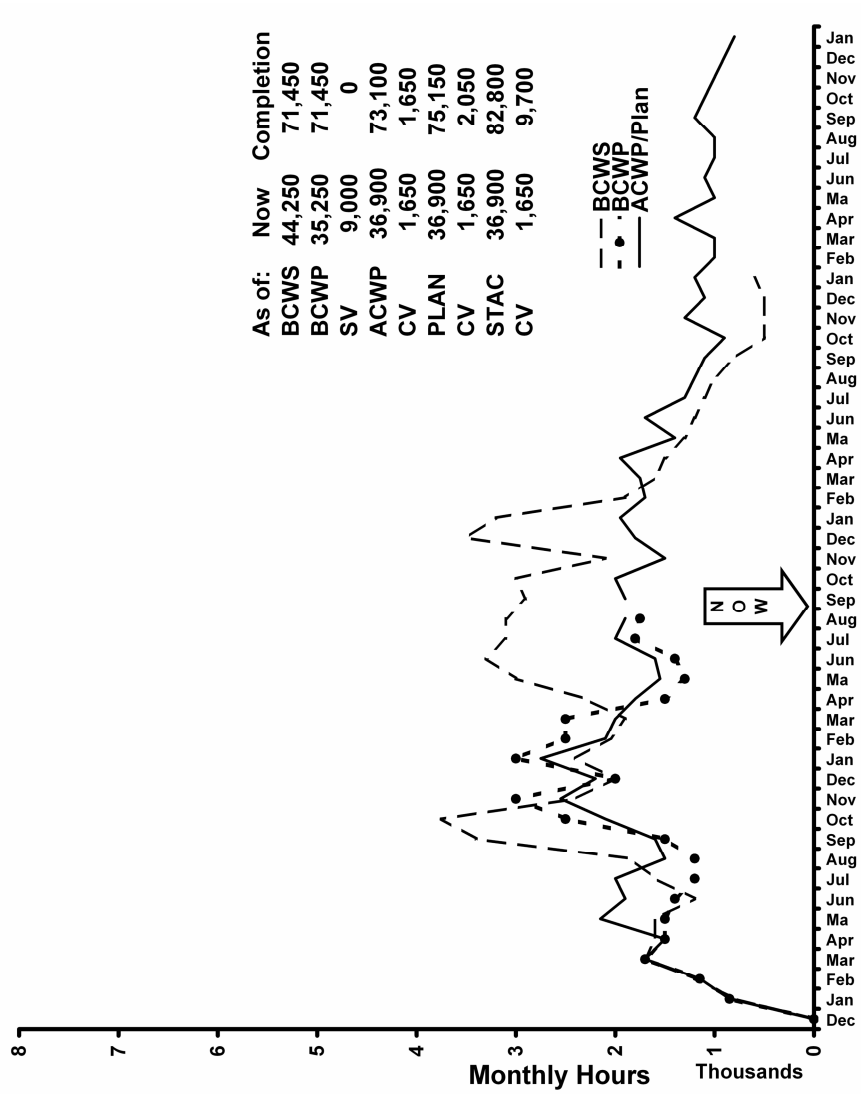

Figure 2.6 Incremental Earned Value

By the way, the spikes in the actuals are invariably an artifice of data collection, such as whether it was a 4 or 5-week accounting month or whenever staff got around to updating their progress. Likewise, the spikes in the planning are often because of some milestones where a multitude of tasks are anticipated to complete, followed by a spell of noses to the grindstone. If your bosses are bothered by this noise, or worry unnecessarily about its consequences, then it is common to use two-month running averages to smooth them. Such also indicates that weekly results should rarely be consulted, as they are mostly noise, not data, and definitely not information.

In practice, all these curves are typically combined into a single chart as in Figure 2.7 for discussions at a monthly project management review, but this combination can be rather daunting at first view.

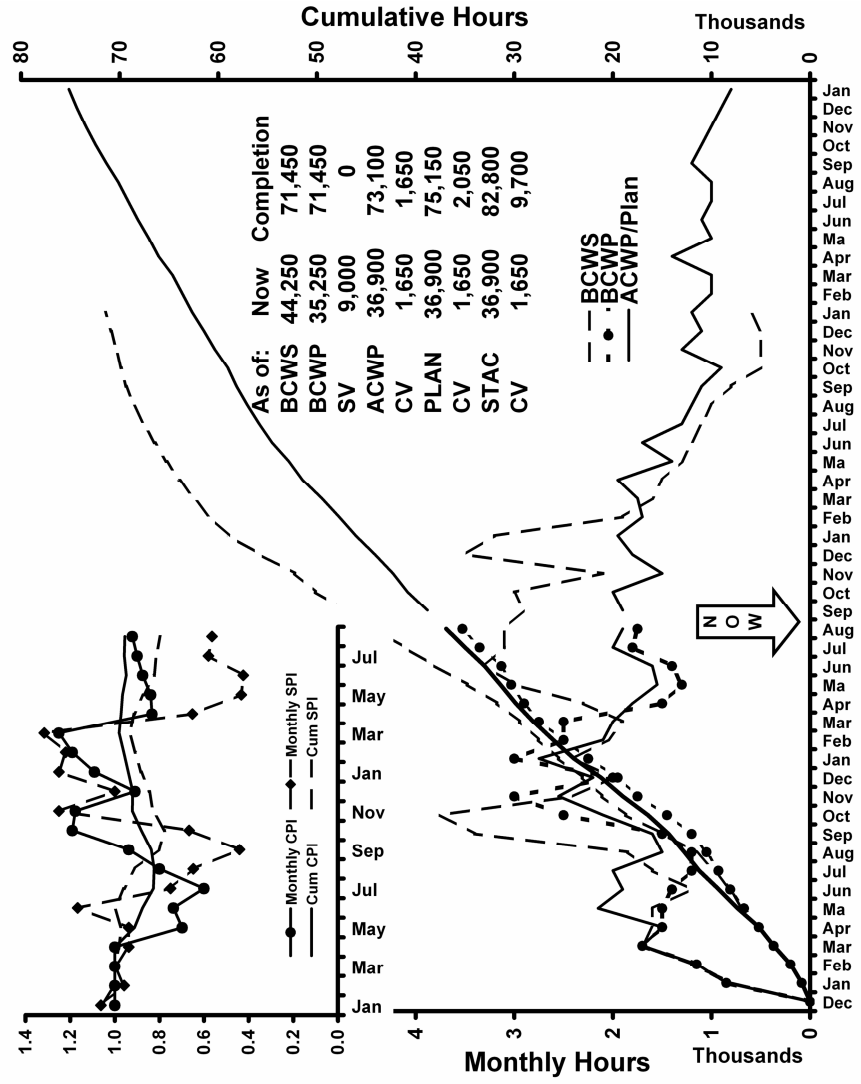

Figure 2.7 Integrated Earned Value Status

Use budgeted labor rates, not actual dollars, when assessing earned value. This example intentionally discussed earned value in terms of person-hours, not dollars, since project managers can only control hours, not rates. However, most senior managers prefer to express everything in terms of dollars. If so, then one should assess a project manager's cost control in terms of the project's originally budgeted average labor rates, and, of course, other direct project costs such as material, travel, and the like. This is often called the "performance variance".

Average labor rates should be used because project managers cannot control who is assigned to their program. When companies use actual payroll costs to track earned value, one always hears from the managers who complain that they have been saddled with more expensive personnel… although that also tends to mean that they are benefiting from more experienced and most efficient staff. Conversely, you will never hear from the managers who are benefiting from the dollar cushions in earned value provided by assigned staff that earns less than the average.

> **Know your resources!** If your financial system will not allow the use of average rates, then make sure you know what each resource is costing you. Note that your more expensive staff members are also likely to be more efficient, so do not make the mistake of blindly trying to obtain the cheapest resources in an attempt to save budget. Such would mainly just adversely affect your schedule.

Rate variances are invariably the responsibility of managers outside of the project team, i.e., senior functional departmental management. Labor rates tend to be dominated by the overall business activity in each major department and their elected capital investments. As such, these are rarely controlled by the middle managers determining the success of any one project.

Performance and rate variance can always be added to provide a total cost variance, if desired. However, their separation presents a clearer picture to senior management regarding appropriate responsibility for management and control.

Scheduling Morality

No, morality is not something you can schedule, but you are liable to find yourself questioning your own rather frequently when you make and,

particularly, when you revise schedules. You will have to balance several dilemmas.

Keep your commitments, as best you can. Remember Noah's Principle. The only truth you know is in the past. You can still change or at least influence your future. Oversights or feature-creep should be your primary explanations for increases in scope or for delays.

Some things just take time. Some tasks are like gestation periods. Carefully assess if more resources can really help. Fortunately, most delays can be offset by adding resources, which is mostly within your control. Be careful though. If the new resources require training or orientation, adding resources commonly will slow down progress in the near term. Nevertheless, particularly for critical path tasks, take the hit, as you will invariably gain overall.

Stay (barely?) achievable to counter the student syndrome. You must keep your schedule objectives achievable, albeit requiring a bit of a stretch. Remember, almost no one ever completes a job early, not because they are lazy, but because they invariably have other jobs to also be done. On the other hand, your schedule must remain believable to the executing staff. Otherwise, they will just pay you lip service. It is easier on your emotions when senior management acknowledges off-the-record that you are playing this game, but do so regardless.

Do not make frequent small changes. While you need to keep your plan current, at least monthly, only a masochist would frequently report minor overall variances. First, you are liable to be just seeing reporting noise that will balance out next month. Failing that, there is always a task or two that can absorb these minor increases or delays in your plan for a bit. On the other hand, if some external event having major impact does not occur at least quarterly (and it often will), then fess up. Waiting longer is not realistic. Be particularly on the lookout for delinquent customer events, which you should be able to note from your plans if you have followed our advice encouraging lots of vertical lines.

Decompose, rather than consider it a blank slate. Your summary tasks are likely more legitimate, particularly if they were based on decent estimating rationale. As noted above, feature creep or major oversights should be your main explanations for changes.

Always schedule top-down. The problem with bottoms-up estimates is that no one ever rounds down. You will commonly find that bottoms-up estimates almost precisely double what the customer has in the way of funding, while luckily very few programs overrun that much.

Put all your reserve in one place and manage it. The other problem with bottoms-up schedules is that everyone has a bit of cushion, and you, the manager, have none. Reserves will be discussed in more detail in a minute, but your job is to keep all the individual tasks free of reserve.

Express your own view as the project manager, not your boss's or your staff's. Bosses and customers obviously do not like to hear about delays or overruns. Your main mitigation strategy will be continually demonstrating your adaptability in working around these issues as they occur. Unfortunately, your staff is simultaneously pressuring you to be more realistic, defined by them to mean relief from meeting the commitments they made long ago. You should strongly defend them from feature creep and external changes, but that is about all.

Management Reserve

Any good project manager needs and has a management reserve. Usually, it is created by holding back five to ten percent of the budget authorized by management when he/she negotiates the original project plan with the functional managers. If the latter are smart, they will not complain because they will be the ones who likely spend it all before the project is done.

Management reserve is *not* for covering overruns but for maintaining the integrity of these internal contracts as features creep. A project manager has to have some source of funding to authorize tasks and/or features that no one was omniscient enough to anticipate. Otherwise, these internal contracts rapidly lose their validity. The project manager may try to cajole or bully the functional manager to swallow this new task within existing planning and budget. Even If the latter agrees, they both are the worse for it since neither will ever really know what it cost them to do the originally planned tasks. For example, some might claim that their person-hour per drawing estimating factor is faulty, when the real problem is that they ended up creating quite a few new drawings that were unplanned. Alternatively, if the functional manager demurs, these plans no longer reflect a negotiated agreement and become simply out-of-date scorecards for both to be beat up by Finance and senior management.

Feature creep is an unpleasant fact that needs management, and that requires funding for the staff doing the work. Creep can be much more than the commonly understood cosmetics, GUIs, and report formats. The classic case was in the early seventies with the Lockheed C-5 cargo airplane development. There were several papers written at the time discussing that project's substantial overruns. Lockheed

explained it in terms of things that were “known”, were “known to be unknown”, and the fatal “unknown unknowns”. The latter quickly became known as the “unk-unks”. Their point was that they had planned for the first two categories. The “knowns” were mostly just work. The “known unknowns” had extensive, explicit risk mitigation plans and tasks in place. However, the unk-unks were their downfall. These led to substantial additional tasks costing the company dearly. (The best thing that ever happened to Boeing was probably losing this C-5 competition and, instead, applying their Air Force funded R&D toward building the 747 commercial airliner.) Actually, the C-5 has turned out to be a rather good plane once the Air Force gave up on some of their more aggressive requirements, such as trying to land this monster on a 5000-foot dirt runway.

Management reserve must be controlled by management, not the project manager per se. Typically, a manager reports any so-called management reserve transactions during periodic reviews with senior management, although some companies require prior approval. Mainly, they should be looking to assure that this funding is for a legitimate oversight, not an overrun.

However, the project manager should be measured as if he/she was expected to spend all the reserve. That is, management reserve is *not* potential profit. If senior management behaves as if they consider this profit, then the project managers will just hide their reserves. Typically, this is done by squirreling away funds in the accounts that will be spent last in the project, such as warranty and Customer Service. Unfortunately, now both the project manager and the Customer Service manager unofficially have to agree to this practice of hiding reserves or further budget battles will ensue. Do not make your managers hide their reserves. Instead, manage them.

Chapter 3 System Evolution

As one journeys through the lifecycle of a system development, some phases have more issues than others. While one might expect the bulk of the work to cause the bulk of your problems, such is mainly just hard work. The trauma, as in life, mainly occurs during the beginning and the end.

Bid & Proposal

Badly bid jobs are stillborn. No amount of cleverness can fix them. There are two primary sources for bad bids: fear of losing and optimism regarding development complexity.

There are worse things in life than losing a bid. Unfortunately, winning with a bad bid is the primary one. Companies are particularly fearful when they are the type that mostly lives from one large project to the next. The assumption is that they must win or substantially reduce the scope of their company. No one likes big layoffs, so most are willing to gamble that they will not lose that much on the basic job while eventually getting well on either changes or long-term service agreements. Sometimes it works out; often it does not.

If you never lose a bid, you are leaving too much money on the table. This is sort of a corollary to the prior one as it usually results from fear. In practice, this also means you are depending on price alone to win a competition. In reality, there is some basis to that premise since, despite procurement rules claiming also to weight technical factors, it is very difficult for a public agency not to select the low bidder, as they almost have to prove it was technically non-responsive, not just worse. Therefore, if you never lose, it means you are cutting your prices very thin. On a development contract, it may still be reasonable to do so, since winning may assure you many years of parts and service business; just try not to kid yourself regarding your initial cost risks.

Be honest regarding your capabilities if much development is required. The other common cause of larger overruns is underestimating the complexity of development. This is particularly common when this is a new product for your company. Stated another way, incumbents can know too much. While that may provide a new entrant an opportunity, the new entrant should view the bid as a calculated risk to open up a market.

You *can* develop new markets, but incumbents have inherent advantages. Make sure they are either resting on their laurels or milking their cash cow. In those cases, you actually stand a good chance of succeeding without a large overrun. Unfortunately, it is still a big gamble, as there is no evident way for you to know truly that such is the case.

Avoid building a house of cards with multiple projects sequentially dependent on each other for developments. Alternatively, at least recognize that you are greatly compounding your risk, particularly as regards schedule. You will still likely benefit by reducing the overall development costs, since you can leverage this multiple usage, but the price you pay will be delays in the downstream projects.

Cost and price are distinctly different issues. Senior management will invariably claim that they know the difference, but they will rarely admit that they know they are buying in. Instead, they may berate you to accept a "management challenge". As long as you have sound basis of estimate, do not do so, but stick to your facts. If not, you probably deserve your fate.

Overheads are real costs, but allocating them realistically is not trivial. Overhead is a category of expenses that accountants use to attempt to address all their costs without having to actually collect minutia regarding what each staff person worked on every single minute of the day. For most salaried staff, it is difficult to get them to report their time on more than a couple of cost collection or charge numbers per day (although such is mainly an issue of laziness or disdain since hourly workers do so routinely).

Allocations are inherently imperfect. A classic example is the quandary associated with procurement overheads, which can lead to the classic $1500 toilet seat. Two obvious choices are to allocate the costs associated with vendor interaction, catalog research, writing purchase orders, etc. in terms of price or quantity. If you have ever filled out forms and processed requests for buying anything, you know there is almost as much effort buying something costing $25 as something costing $2500. The key word was "almost", since there will be some added effort to buy something costing, say, $250,000. A common distinction is to create one overhead pool dealing with purchase of catalog items and another dealing with subcontracts. Since the latter are custom items, they all tend to be more expensive. By going to two overhead pools, it is usually reasonable to allocate costs based simply on the

quantity purchased in each pool. Even so, if you use quantity as the allocation basis, then you will invariably create a $1500 toilet seat.

Some organizations still prefer to allocate based on purchase price. This leads to the opposite problem where the expensive items bear almost all the cost burden. Using that scheme solves the $1500 toilet seat issue but can drastically mislead managers regarding their true costs, e.g., when considering a new proposal or when trying to project impacts of future business alternatives based on “actual history”.

Incremental pricing can be considered to enter new markets. Incremental pricing attempts to ignore overheads and estimates the organization’s out-of-pocket costs for a project. The question that you must carefully ask is whether you have really captured all the incremental costs, particularly regarding efforts required by staff normally covered by overhead pools. If no one else in those groups is going to be hired because of the project, then ignoring such overheads can provide a lower bound of what one might bid for a development contract, typically providing an opportunity for substantial growth into new markets.

A common variation of incremental pricing is the strategy for selling commodity items, like PCs. Most customers will violently object to paying any substantial premium over what they would pay directly to the manufacturer of a commodity. For a variety of materials, letting the customer buy them direct is a practical solution. However, many items like PC may sell like a commodity, but if you ever run a service department, you will find that they are far from truly interchangeable. It is therefore often in a company’s self-interest, typically to minimize service and support costs, to provide PC’s for sale at competitive prices. If so, it can be reasonable to demand that the customer purchase replacements from you so that you can control the product variation.

Just do not kid yourselves; someone has to pay those overhead expenses. They are real costs. It is only the allocation schemes that often make them seem unrealistic.

Almost all staff should charge directly to a specific project or product. Even if they are lazy or disdainful, as noted earlier, their judgmental allocation will invariably be more

realistic than the usual simplistic mathematical allocations routinely used by finance departments.

Do not ever think you can drive a competitor out of the business by cutting your prices. All you will do is flush both of the companies' profits down the toilet. Once a lower price point is set, it is almost impossible to get it back from customers. Companies with their founder's still in control are particularly susceptible to trying this ill-fated strategy.

Answering four questions are paramount for research proposals. Back at the Defense Advanced Research Projects Agency (DARPA), we always looked for these answers before proceeding…

What is the trick? What is unique regarding your approach? What is going to give you a chance to succeed when others have had difficulty? Why should we fund your idea instead of that from any number of other bright, hardworking people?

What is the plan? No one really believes that you will actually be so lucky as to completely follow the plan, but if you cannot even lay out a plan, timeline, major tasks, key decision criteria, etc., you will spend most of your time wallowing around in mostly Brownian motion trying to start. In addition, without a plan, it sounds like a proposal to "send money so these bright people can muck around and try to come up with something useful".

Who cares? What customer or user community will specifically and tangibly benefit from your success, and why is it important to them?

So what? We will stipulate that you do everything you claim, so how much money, time, cost, or whatever will the customer save? Be quantifiably specific, or it is just science for science's sake.

Architect for Fault Tolerance

Inherent fault tolerance is highly desirable in choosing among system decomposition alternatives. Precluding single-point failure modes is a common design criterion in many circumstances, particularly as it relates to unsafe conditions. Failing that, one can design such that the device or system will "fail safe", i.e., if the component or device fails, it does so in a manner that leaves it in a safe condition. More

commonly, some form of redundancy is used when continuous operation is mandatory. In most commercial practice, we are not talking about formal fault tolerance, but simply are assuring that no harm is done and that most failures are masked from the end user.

Simple brute force redundancy is not that simple, as it requires some means to detect confidently that the primary device has failed as well as some means to provide the secondary device with all the history or existing knowledge of the primary device. Simplistic failure detection schemes can lead to excessive false positives. In turn, fear of false positives then can lead to schemes that vote, e.g., additional redundancy or detection mechanisms so that you can more confidently assure that a failure has actually occurred. Further, simple redundancy literally has half your capital investment sitting idle most of the time. Worse, it makes the transition from primary to secondary device into a non-trivial, time-critical special circumstance.

Load sharing is a more useful fault-tolerant architectural scheme. As the name implies, load sharing is inherently fault tolerant since the additional device(s) are routinely performing the same functions as the primary device. Its complexity results from needing to synchronize continuously among peers, rather than just upon transition from primary to secondary. However, once you have done so, fail-over is inherent, i.e., no special actions are required, and adding additional capacity is also inherently simple.

Store-and-forward is an architectural scheme to protect against the commonly occurring loss of communication between subsystems. Information is retained in multiple locations as it is migrated from its source up the subsystem hierarchical chain to its final repository. The idea behind this scheme is that each entity in the hierarchy is then able to exist and operate on its own if communication with peers and superiors is temporarily lost. You should always plan on this communication loss occurring. You just need to be careful that the source does not actually delete its information until the ultimate recipient has successfully accepted *and* processed it.

Federated architectures more easily enforce good design practices. Federated architectures are made up solely of peers, that is, there is not any overarching device or software process whose health is needed to assure the continued health of subordinates. Chapter 4 includes several strictures, such as the use of messaging and not having to kill a healthy process in order to revive a sick one, that are inherent in a federated scheme. It must be admitted that a centralized architecture can also be designed such that these design rules are also followed. Unfortunately,

experience also suggests that centralized architectures also make it easier to cheat, in very non-evident ways.

Do not forget about software single point failure modes. So-called god processes are just as dangerous as hardware single point failures and possibly less evident.

Make It Work, then Robust. Only Then, Make It Better.

This is another form of declaring victory. Early on, you need to decide just what is the minimum functionality that your product needs. Some people call this "day zero" functionality. Regardless of your terminology, this is the definition of what you *must* implement. The remaining features are more than just frills, but they are not essential.

It is routine to implement requirements in phases, but you will rarely succeed with designing in phases. That is the main reason the other features are not just frills. If you do not architecture for them in the beginning, you are very liable to have to start over, rather than just iterate your way to success.

After you have "day zero" appearing to work, then focus on making it solid as a rock. Again, we remind that well over half of production software relates to handling exception conditions, not to doing what customers or Marketing thinks they have asked for. Both customers and Marketing will push you constantly to add those frills. Do not do it. Their begging quickly turns to regret when the residual bugs surface, particularly if in front of the customer's customers.

> Note that this is simply robustness in its layman's connotation, not the formal design for robustness that rigorously attempts to desensitize designs to noise and variations by parameter selection and the like.

Only then, add the frills and do so slowly, so that you never lose the robustness. Commercial products have the distinct advantage that the developer has the prerogative of defining, mostly after the fact, just what the product will do. It is much easier to fend off your own Marketing department regarding frills. If you are doing contracted development, you may be forced to work with your customer to accept a delay in the frills, but as long as "day zero" is solid, they invariably will agree.

Branching is a Necessary Pain

Unfortunately, fixing bugs and adding features are at cross-purposes. When you are fixing bugs, you want to minimize the number of changes from the mostly robust version that you are trying to improve. By definition, when you are adding features, you are making changes that will invariably add new bugs, particularly into the sections of software code that were not broken before. Branching is a configuration management scheme to uncouple these issues.

Branching takes the software code base and splits off another version that starts with the same code as in the main branch. Let us call the main branch the "bulletproof" branch. Likewise, let us call the split the "features" branch. You now do your bug fixes in both branches, but you only add new frills to the features branch. At any point in time, there are at least three versions of software that a project is dealing with.

> The oldest version is often called the "deployed", i.e., what customers are routinely using in the field. There may also be intermediate versions, commonly called beta. Beta versions can be considered as a tentative deployed version, usually to a select customer subset to achieve usage that is more comprehensive than internal testing. There is nothing like real customer usage to surface exception conditions.
>
> Next, there is a version that is being system tested internally for potential deployment. Historically, this was called alpha software, but that usage has gotten sloppy over the years. For example, some companies even provide select customers what they term alpha software, usually implying that it only contains limited functionality that is known to be buggy. As discussed later, we strongly recommend that you never deploy software that is known to be buggy unless you are willing to disclose all the known issues. On the other hand, there is nothing wrong with deploying software with missing features, as long as it is robust.
>
> Finally, there is the version that the developers are trying to integrate into a system by adding frilly features. This integration testing is usually done within Engineering, before handover to the Test department. This is the version that commonly requires branching. However, since each individual developer has contended that his software is ready for integration, this integration build is usually placed under centralized configuration management control. The latter

group assures that a third party can successfully and repeatedly replicate the build. Note that there are even more versions still under the individual developers' control where they are fixing bugs and adding even newer frilly features.

The pain comes when you merge the branches back together. If the world was perfect, and the developers all truly made the bug fixes in both the bulletproof and features branches in a timely manner, this would not be that hard. Unfortunately, with all the developers involved in a large-scale system, it *is* hard. The staff working on new features does not want the bug fixes messing up their new code, even though it is cleaning up those messy interactions that are needed to merge successfully. Nevertheless, you have to do it periodically, say at least monthly. Be ruthless about merging. The longer you wait, the harder it is to merge. Remember, there are even more frills and fixes coming down the pike. You must get back to a single baseline to move it forward into validation and deployment and to provide a stable basis for the next features branch.

Numbers are Better than Judgment

Everyone has an opinion, and it is often wrong. Company folklore is even more suspect. Later chapters will discuss topics like full in-boxes and continuous improvement processes that beg for data. As a preview, you should ask any group that is deemed non-responsive to other departments to track their inputs, outputs, and backlog, say, for the last six months. We have never found one that was not keeping up, even though they all thought that they were overwhelmed. Similarly, folklore invariably says that most field defects can only be fixed by revising the engineering designs. Again, go gather the facts first. You will instead find that most defects are process failures, which is good because you can change processes much faster than design.

Variation is a fact of life, particularly in the field. As such, you will typically have to characterize field performance statistically. Moreover, you will need to focus on what occurs most frequently or with the highest cost impact, as you will never have the time or money to work on everything. Again, you need data, not folklore, to guide you.

Customers Need Managing Too

It is usually in the customer's self-interest for you to succeed. Remember, they selected you, either by bid or by purchase. Thus, if they are smart, they will work with you as you evolve your system's functionality and to close out a project.

Know your customer! Learn their interests and objectives. What do they primarily need to achieve? What is truly important to them? These answers will be key when you are working with them to declare victory in the end game of the project.

Smart customers will also recognize their impacts on your performance. Yes, they will still whine about your claims for feature creep and for customer delays, but they will not hold a grudge. They know you are not a charity.

Some customers, unfortunately, are not smart. You have our sympathy. These days, at least it is common for them to recognize that and to hire outside consultants to help. Luckily, most consultants are smart. Otherwise, your chances of success are not good, even if your efforts are exemplary. In addition, some consultants seem more interested in maximizing their billings, which can lead to substantial delays. Again, establishing clear customer milestones in your project plans will be key to successful claims.

Some customers are actually mean, but we have likely taught them to be. In effect, they are still holding every grudge that vendors caused them to develop over the years.

Make sure that "beneficial use" is prima facie proof of "substantial completion" in your development contracts. The most specious behavior of mean customers is to enjoy what is called "beneficial use" and still refuse to pay. Beneficial use occurs when the customer is making money or otherwise commercially benefiting from your products. Most contracts also have a term called "substantial completion" which gets the supplier ninety percent of his money, i.e., typically his costs. If they do not want to pay, then they should not use it. (Your management, legal, or finance departments may still want to make them pay even if they do not use it, but that is their call.)

However, ultimatums are never a good idea. Remember, there are two responses to an ultimatum, one of which you will not like. Most commonly, some program manager tries to pressure his customers and/or bosses with something like "if you can not fund at least $xxx, then

you might as well cancel the project". That manager likely forgot that his bosses often consider the expenses to date as sunk costs, or may not realize that other projects have already contributed their fair share of budget cutbacks, or failed to remember that people higher in the hierarchy just resent being threatened, or whatever. Remain flexible at all costs, e.g., any program/project can be stretched out to match any available funding profile. In fact, if the project had been over-running, this stretch-out demanded by customers or bosses is likely to be just the rationalization you needed, since total costs are expected by them to increase because of stretch-outs.

Closing Out

The only thing harder than starting a project is ending one. It is probably just as well that you did not get as many staff initially assigned as you had planned. You only need the architects and departmental leads for the first two to three months anyway. They need to develop those functional specifications and project plans discussed earlier before turning the troops loose to implement them. Otherwise, the troops are twiddling their thumbs or making false starts… that are sometimes hard to stop.

Getting staff off your project is much harder than getting them assigned originally. You would think that this was not a problem as long as the company has other projects in the pipeline, but it is. Again, a product- or output-based project plan is your best tool. It is hard for them to justify their charging if they are doing something other than producing the products that you requested.

Be careful when you are finishing development but not yet into full production. That is the riskiest time for any project as it is the last time to kill it practically. Management has endured most of the pain, but those costs are considered sunk. Now, they are facing substantial additional investment in production and service, by both the company and its customers, so it is their last chance to turn back.

Declaring victory is usually required. At some point, the frills must stop, even for a contracted development. Bugs will still need to be fixed, but even those will need to be prioritized. Commonly, several will require you to treat a symptom rather than fix the underlying problem, either because you cannot replicate it or because its architectural impact is too extensive.

Chapter 4 Often Forgotten Programming 101

Much, if not most, of the functionality in products today is provided by software, some evident, some embedded. As such, you will spend a disproportionate amount of your time finding and correcting software defects in your new developments. A much better approach is to preemptively attack this issue by developing, promulgating, and enforcing documented software design guidelines. Some companies literally have hundreds of pages of such guidance, but one has to wonder if it is more to impress the ISO-9000 auditors than to help the staff. We recommend that you again stick with the basics, two to five pages at most that your staff might periodically reread, and focus on the poor practices that cause the most schedule delays.

Internal software design guidelines should focus on architectural and design issues, rather than style or format. While the latter offer some improvement in reusability and maintainability, one will rarely encounter a time that they cause large program delays. On the other hand, failing to comply with the following groundrules will resurface painfully and frequently in most organizations that are developing software as part of a system project or product. They are listed in order of pain or payoff, depending on your perspective.

Several staff will resent your hammering home these groundrules, but do it anyway…, repeatedly. The responses are along the lines of "Who is this new boss that is nagging us about these simple rules? Everyone knows them." Yes, but everyone also seems to violate them far too frequently. This list has changed very little over the last twenty years.

1) Do not embed text, parameters, flag conditions, etc. in source code. There is no such thing as a constant (except maybe π and e). Always use setup or "ini" or database files as the source of configuration data.

> **Configurability of your application is your only hope in controlling costs**. Design and technology is constantly evolving. Market forces are just too strong. The good news about software is that it is easily changeable. The bad news is that the market continually forces you to change it. You either incur substantial costs doing customizations or make the code configurable so that someone other than developers can adapt it to the customers needs.

Even if it is a purportedly small change, retesting for unanticipated side effects is costly… whether you do it now or later when the side effect shows up in the field because you "saved" costs by limiting retest. It is just too easy to delete, say, a brace in your C++ code by mistake.

You will need at least four levels of access to configuration parameters: developer, production, customer service, and end-user. A simple hierarchical password access mechanism is usually adequate.

Regardless of the access level, you will need a configuration application, not just a file editor. Even my experts (and they really were) have made several disastrous mistakes editing in a live system. This configuration application mainly provides some validity checking of entries and logs who made what change from where and when. These logs are a godsend when digging out of a problem, as too many seem to have selective memories when faced with a downed system. Since everyone can make a mistake, it is also helpful if this application offers the option to revert to using some saved defaults.

Embedding text was the first no-no in the list for good reason. While commonly found as error messages, discussed later in Rule #4, most products today are made for international usage. If you have ever tried to translate embedded text, you will adopt this rule quickly. Text is also difficult to accommodate in fixed-length display fields because the length of text for the same meaning varies greatly. For example, German is usually notably longer than is the corresponding English. Double-byte languages, like Arabic or Chinese can also create issues.

2) All logs, files, databases, etc. will be circular, ping-ponged, or otherwise bounded. Recovering cleanly from a full disk is almost impossible. Unbounded log files invariably run wild at the most inopportune time. Experience in a variety of industries suggests this is likely to occur about once per quarter.

3) All processes must be able to be independently started and shut down without any assumption regarding the state of any other process. For example, never have to kill a good process to recover a sick one. Even though we suggested the use of "ini" files in rule #1, we are not a big fan because many applications are written to refer to the

"ini" file only upon startup. Such is still much better than embedding the values in source code, but killing healthy processes can make things difficult, particularly in real-time systems. As you would expect, this rule increases geometrically in practical importance as the number of simultaneous independent processes increases.

4) Each and every possible error or exception condition should have a unique error code ID. The code should be passed rather than the text to be logged or displayed. Text should always be pulled at the last minute from a table or resource dynamic link library (dll). The text should also be unique (no generic "system error" messages) and should generally display the code, a description, and the suggested action by the user: close and reopen app, reboot, call service, whatever.

> **This rule is mainly for the benefit of the developers, not end users**. Debugging a system is hard work. Why tie one of your own hands behind your back? You have already gone to a lot of work and have myriad "if/then/else" tests coded. Why have those all return the same error message, particularly if they are often also silent about key parameter states? While production, customer service, and even end-users will also benefit, the original developer most needs this to get through initial integration.
>
> **As a side note, avoid strict equality tests, even on integers**. This one shows up about once a year when some counter jumps, initially inexplicably, and then a mess occurs because that loop is now unbounded.

5) It is the sender/creator's responsibility (not the receiver's) to assure that data is valid, such as being of proper type and within usable range. (While it is understood that the receiver will probably also do so with debug statements to get through initial integration, these should be able to be turned off easily from an ini file once the senders do their job.)

> **a.) Data should never be able to kill a process**. Every process should keep going regardless. Post an error message, but never halt. Use a default, a bound, whatever, but keep going. While it may seem efficient to halt upon an error, doing so turns any attempt at integration into a time-wasting serial, rather than parallel, process.

6) Never use time to imply sequence. If you need to insure or know sequence, use an explicit sequence number. Remember time can and will go backward, usually in the most awkward circumstance, unless you happen to be the master clock.

> **a.) As a corollary, all machines should only run on GMT/UMT**. Only displays and reports that humans see should use local time, but that is just a matter of display. You will only have to recover from a single daylight savings time switch to understand this rule.

7) Processes should not communicate through global or shared memory. Use messaging. In the old days of limited memory and CPU speed, this was a more common problem, but it makes integration extremely difficult, as it leaves no record of who did what to any parameter.

8) When cutting back requirements, do not implement just one if N is required. At least do two.

9) All data should be passed as pairs of the variable/field name and the value. Do not make any assumptions regarding field order, location, type, or size. The latter should be in setup files anyway. (In truth, this is the only rule that you will sometimes concede, but its benefits remain, so it stays on the list.) It is understandable how this practice arose in the old days of slow computers and limited memory, but you will only have to go through one substantial database restructuring on a live system to understand these benefits. Such restructuring is almost impossible without shutting down the system, and many systems like broadcast cannot allow you to shut them down.

10) Do not use TBD or be silent on a functional trait. If you do, you are saying that you are indifferent to the value and, thus, someone else can decide it arbitrarily. Such is rarely found to be the case. To-be-determined (TBD) entries are often a good clue that you are dealing with agreements to disagree that can lead to interminable integration delays. At best, they indicate indecision regarding the design.

> **What matters is making a timely choice among items on any short list of alternatives and getting on with it.** The choice almost does not matter. By definition, all items on a short list meet your basic needs. If there are several good

approaches, just pick one. The differences are more like religious preferences, not technical. For example, in the old days, there was much debate about whether to choose WordPerfect or MS Word (or even WordStar in its day). It did not matter as they all easily produced very nice correspondence. Not unexpectedly, formal user preference studies invariably show that people prefer the one with which they were most familiar.

Chapter 5 User Interface Design 101

Graphical User Interfaces (GUIs) are also now an integral part of most products. Their design involves a mixture of artistic and stylistic preference combined with technical traits that ease the product's usage by beginners while still allowing proficient users access to all the bells and whistles your product provides.

GUI design should be viewed as a religious preference, not technical. Digital Equipment Corporation (DEC) staff first acquainted the author with this perspective of religious bias in engineering. There are still quite legitimate advantages and disadvantages with many of the design choices you have to make, but do not carry on like there truly is a right and a wrong way. They are mostly just *different* ways.

> For the younger reader, DEC totally dominated the mini-computer market with their VAX's in the seventies and eighties. This perspective arose in a DEC presentation in the eighties comparing the benefits of the three dominate networking protocols at that time: CSMA/CD (best known as Ethernet or IEEE 802.3), GM's MAP (token bus or IEEE 802.4), and IBM's Token Ring (IEEE 802.5). Even though DEC was solely in the Ethernet camp, their point was that each had its strengths and weaknesses and all could be used to create quite useful networks, ergo it really comes down to a religious preference. The author was biased in favor of MAP, but Ethernet won out ~~solely~~ for economic reasons (see, to this day, the bias remains). By the late eighties, one could buy Ethernet adapters for 1/10 the cost of the others.
>
> As a further digression, the problem with Ethernet is that you cannot *guarantee* delivery of any message within any finite specified time. At the somewhat slow networking speeds of the day, such uncertainty greatly complicated real-time control of machinery. With time, sheer overall speed increases reduced this problem from a practical perspective, and the adapters' recurring cost advantage made worthwhile the one-time investment into more elaborate machine exception handling.

However, it is important that you express your beliefs. Internal GUI standards are still quite helpful, almost irrespective of their content. Engineers are lousy artists but love to play with graphics. They routinely waste a lot of time playing rather poorly with the cosmetics. You are doing everyone a favor by establishing a project or company standard

that takes many of these choices off the table. Such usually also has the benefit of providing an overall corporate family "look and feel" that can be evangelized commercially by your Marketing staff.

Clickable Mockups, Often in Lieu of Specs

A picture is truly worth a thousand words. We now recommend foregoing written specifications to define the needed functionality of any product whose primary user interface is graphical. Instead, use a tool like Adobe/Macromedia's Dreamweaver® to develop a static prototypical representation of the functionality. It is static in that the representative data in dropdowns, tables, and the like do not change, unlike the varying data in the real product. However, these "clickable mockups" are dynamic in that they support full user interaction with the screens, links, dropdowns, error conditions, etc. Said another way, these are not cartoons or pictures or PowerPoint slides, but they are a fully working "website". It is just that their data never changes.

Prototyping is relatively fast and is best done by somewhat artistic staff, usually *not* engineers or programmers. One-hundred-screen prototypes can be drafted, reviewed, and refined in two weeks or so, at least internal to your company. All of the cosmetic choices are settled in this process. Nevertheless, Engineering should be heavily involved in these GUI design reviews to assure that the mockup is efficient to implement. There are invariably a few alternatives to achieve the same functional objective that are more efficient than are those in the draft.

Clickable mockups substantially shorten schedules by allowing several subsequent processes to occur in parallel. When done, the mockups are given simultaneously to engineering, test, and marketing/customers. Engineering's direction is simply "Make these truly dynamic. Don't waste any time thinking about the cosmetics or content." One no longer has to wait for Engineering to almost finish product development before one can see how the product will behave. Testers can use the mockups to train their automated test software, which is a non-trivial, time-consuming process. Marketing and/or customers can use the mockups to show others how the product will behave. They may also have to obtain further approvals, but tweaking the mockups is much easier than waiting for and then changing the full product software.

Mockups set a baseline against feature creep. If you are building a custom product, you now have a solid basis for subsequent claims for changes… and there will be changes. Remember, we are talking here

about religious preferences and cosmetics. Failing to solidify such baselines is a major cause of project overruns.

Admittedly Biased Design Practices

Figure 5.1 illustrates a GUI design that implements many of the design rules that follow. Recall, the main point is that you should codify your *own* beliefs, but the listed practices can serve as a starting point for your consideration. As but one example, despite the current abuse by internet advertisers, these rules strongly recommend the use of pop-up or daughter windows so that a user does not lose their context when performing a subordinate action.

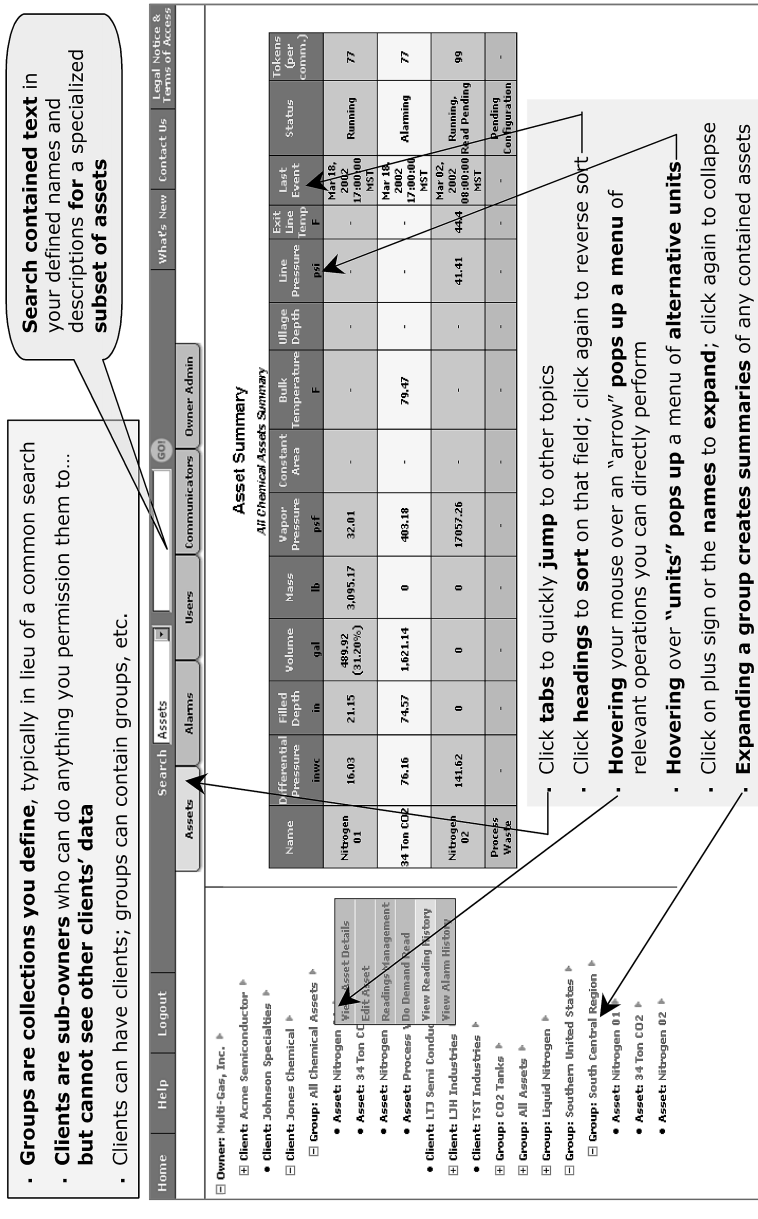

Figure 5.1 GUI Illustration

Arrangement:

1. **Maximize horizontal real estate for use with tables and graphs,** such as placing hot-link menu tabs at the top of the page rather than on the left margin.
    a. Most, if not all, screens should retain the top-level menu choices as hot links.
2. **Avoid action buttons only located at the bottom of a screen**. Such would require scrolling through an entire list that may be quite long. Duplicates are fine, to avoid having to scroll back to the top.
3. **Do not move the user to even a daughter/pop-up screen just to make other navigation choices**. Provide all the relevant navigation choices on the initial screen.
4. **When returning more than a single result, err on the side of presenting more information rather than less**.
5. **List/Table views will be common.**
    a. Just do not require routine horizontal scrolling.
    b. Minimalism invariably forces the user to more database queries.
6. **Make most confirming or administrative actions occur within a secondary daughter/pop-up window**.
    a. Avoids the user losing his or her context which will often be a table or list view
    b. Avoids having to re-access the database to repaint the table, unless the secondary action invalidates some currently displayed field.
    c. That this behavior may restrict users to modern versions of browsers is OK.
7. **Provide a "printer-friendly", non-graphics mode where relevant,** to facilitate users with slow modems or handhelds.

Actions:

8. **Do not make a user confirm any non-destructive action.** For example, avoid repainting the same new user data for confirmation even though it is subsequently easily editable.
9. **However, do provide a visual cue of success.**
10. **Do make a user confirm any truly destructive action**, such as if you really were doing a "delete", rather than a "hide".

11. **Avoid destructive actions, almost at all costs**. Instead, allow at least someone in the hierarchy to edit anything and provide hide/unhide modes. Normally, destructive actions are limited to privileged users like customer support or development staff, but even they make mistakes.
12. **Avoid mandatory fields unless they really are.** These typically should be unique and a primary database key.

Behavior:

13. **Each record in a list view should provide hot-links to obvious individual views:** each device, each end-user, each sensor, each whatever.
    a. These individual views should invariably be presented in a daughter window/screen so the context is not lost.
    b. Each individual view would then contain appropriate action buttons to modify, graph, hide, add new, whatever.
    c. Paging (first, previous, next, last) should be avoided, almost at all cost. It just will not scale up.
14. **List/Table views containing data should contain a mechanism to select one or more parameters for graphing**. However, see the following extensive style discussions regarding graphs.
15. **List/Table views containing data should also include a mechanism to change: parameters** (e.g., rainfall or temperature); **units** (e.g., inches or feet); period (e.g., current month, year to date, last 12 months, etc., but primarily including a "custom" choice of start and end date); **function displayed** (e.g., current/last, cumulative, extrapolated, mean, median, mode, max, min, whatever), **and resolution displayed** (e.g., number of decimals).
    a. One implementation could be to click on the heading but then spawn a daughter window offering to sort or to change the displayed values since sorting on data is not that common.
    b. It would be highly desirable to retain these display choices on a per end user basis. Database retention is probably needed for true persistence, but using cookies is an acceptable expedient since the user would only have to repeat the selection process when they change machines.
16. **Every List/Table view should have a button offering to download that view as an Excel spreadsheet**.
    a. This provides the ultimate cop-out when users ask for even fancier features than one currently has.
    b. However, it does imply that one provides list views rather than just individual views.

17. **Administrative entry screens should usually have a user-settable choice of clearing and/or closing after a submit**.
    a. "Clear" needs to be a choice when the default behavior is to retain the submitted fields because many (such as contact name, address, city, state, zip, and country) are likely not going to change for the next entry. Otherwise, it is a pain when one does need to change everything.
    b. "Close" needs to be a choice for when the user is done, that is, they neither want to "clear" nor simply modify a few fields of the prior submit.
    c. One should auto-populate city and state based on zip code, or at least validate them.

18. **Avoid auto-generating passwords for users**. They will not remember them and will write them down which is an even greater security risk.
    a. If needed, let them suggest, and have your software enforce good password practices, such as length minimums, no dictionary words, at least one non-alpha character, whatever.
    b. If socially allowable, prompt users periodically to change their passwords by checking their password's age. Changing is about the most robust security protection.

Search & Sort:

19. **Provide a customer-definable descriptive field for each device or configurable entity**. These will invariably be their primary sort and search fields. It would be nice if we could enforce their uniqueness, but…
    a. It's hard to do, even for a single user
    b. The only benefit to the user is that such avoids having to pick from a subsequent list/table.

20. **Most query screens should offer at least a couple of query fields that support simple standard "*" and "?" wildcards**. One such field will invariably be the user-definable descriptive field discussed above.
    a. If blanks are left in all the query fields, a table should be provided that lists all the devices accessible by that user.
    b. Warn if a list is overly long and/or offer to split the list into pages of a user selectable length.

21. **List/Table views should have sortable fields/columns** invoked typically by just clicking once on the field heading, with a subsequent click on the same heading reversing the sort order.

a. It is highly desirable to allow the users to select a multiple column sort, probably using a daughter screen to select which column to sort on first, which is second, etc.

Cosmetics

22. **Confirm that each combination of colors, fonts, and fills will still distinguish intended differences when printed on a grayscale printer**.
23. **Always combine a color change with some distinguishable difference in font or fill**. Remember, about 8% of males (including two of the author's CEOs), as well as up to 1% of females, are colorblind.
24. **Do not use underline as a visual cue except for its standard usage of denoting a hot-link**.
25. **Hide or gray-out any link that is not accessible for the current class of user or that is not yet functioning**.
26. **Provide a visual cue that an action button was successfully pressed**. Moving to or presenting another page is sufficient, but a simple screen repaint of the current page is not.
    a. Change the button's color and fill pattern for a noticeable duration, put up some temporary "successfully submitted" text, whatever.
    b. If one has to create a subordinate screen to achieve this, try to make it automatically disappear after some fixed duration so that an "OK" is not required. However, include a "close this window" link as well for those who would rather click than wait.
    c. Regardless, make the subordinate confirming window a secondary or daughter screen so that the context of the action button is not lost.
27. **Time should be handled internally only as GMT but should invariably be displayed in local time**.
    a. Date formats should offer a choice regarding display, such as month/day/year, year/month/day, etc. There are several local conventions internationally.
    b. Provide a user choice regarding the resolution and format of date and time. Four-year display fields as well as seconds and fractions thereof are not usually meaningful and take up valuable screen space.
28. **Use of commas and periods as numerical and decimal separators again should be a local user-specified convention**.

29. **Avoid embedding text into graphics since they are a bear to translate.**
    a. Instead, try to develop a unique graphic for each class of action.
    b. Use international symbology rather than text. You can put the text explanation into a help screen.
30. **On-screen hints are helpful but should also be placed into "help" with more clarifying details**.

Semantics

31. **Do not use the term "delete" unless you really mean it**. Instead, use "hide", "de-activate", "close", or whatever is appropriate.
32. **Do not use the term "ID" when you really mean a name**. "ID" usually implies some non-real-world-meaningful alphanumeric code.

Graphs

33. **Units must be displayed on any graph for each plotted parameter.**
    a. If relevant, offer at least a second y-axis.
    b. It is OK to require the user to make sure their selected parameters will display reasonably on a single y-axis.
        i. At least warn them if the various ranges are grossly inconsistent
        ii. More simply, one could offer a logarithmic y-axis for those users who can understand them.
    c. If practical, offer cumulative/stacked as well as the independent parameter plots. However, recall this requires a check that the requested parameters for stacking all have the same units.
34. **History line graphs should be plotted against a single time axis**. This often implies that one will normally be using x-y plots since the data will often not be available at consistent time intervals.
    a. The user can select the period, typically by selecting the period displayed in the source table that is being graphed. One should then adjust the scale of the time plot to reasonably display on a landscape printable page.
    b. If the user wants multiple pages or other fancier stuff, let them use the "download an Excel-compatible version of this table" button.

35. **Bar graphs can be used to plot sequential histories**, for example, where the x-axis represents independent events taken at inconsistent time intervals.

Speed:

36. **Responsiveness is paramount to a satisfactory user experience.** A handful of seconds should be the goal for the maximum wait for any single database query screen to load.
    a. Limit the spatial and, particularly, the color depth resolution of all graphics, such as no more than monitor resolution of, say, 96 pixels per inch, and preferably only 256 colors.
    b. As noted earlier, offer a “printer-friendly” text-only version for users with slow modems.
    c. As noted earlier, warn the user when one anticipates the database query to take more than 2-3 seconds and/or the list to be more than, say 50 entries, corresponding to a single printed page. Offer the option of a refining query and/or a paged presentation. You are not likely showing ads; so do not limit the lists to an artificially low number of entries. Eventually, the number of entries per page should be user-settable.

Chapter 6 Presentations 101

Become a religious zealot with respect to presentation style. While these recommendations may seem to be just preferences, they were painfully acquired, and they work. Most result from extensive briefings of generals and admirals in the Pentagon. That environment is unlike anything you will likely encounter outside of aerospace, but the resulting habits apply everywhere.

Allow yourself no less than two minutes a slide, preferably three. Divide the amount of time for your presentation by this factor and determine the number of points you will be able to make. The result will be 20 to 30 slides or claims per hour.

> You will always encounter people who say that they have simple slides and can go through many more than that in an hour. If they actually can, it is usually because they are showing some sequence that they are impractically asking their audience to keep in their mind. Alternatively, the slides are not truly significant to the overall presentation.
>
> If you have a sequence of graphics that you need the audience to remember or relate together, montage them all onto a single slide so that they can listen to you make your point rather than trying to remember whatever was up three slides earlier. In the days when presentations used film transparencies, one would build up a montage using overlays. Doing so still lets you make your point about each constituent element, but when you are done, the audience easily envisages your claimed relationships. Montages are even easier today with electronic slides, such as by using Microsoft PowerPoint® animation features.

Write an action title for each of your slides. This is a short sentence in the active voice describing the claim you want to make. Some people are offended by action titles. If that is you or your audience, you can use a bland title and make your sentence the first bullet on your slide, with everything else subordinate.

Action titles answer the question, "Why are you showing me this slide?" Unfortunately**,** that will likely become the most common question you ask when reviewing draft presentations.

> Horse charts are the most offensive. (That is a chart consisting of a picture of a horse with the title of "Horse".) The

most common engineering horse chart is invariably entitled "schematic". Another slang term for these is "chamber of commerce" slides. That is, they are slides with which you can change your story with differing audiences. Nevertheless, the real message that horse charts tell your audience is that they are not worth the effort for you to prepare material specific to their interests and needs.

For the record, schematics can be a very effective graphic to support all kinds of claims regarding a design. Just figure out your main claim as a title; add some bullets or callouts to refer to the relevant points on the schematic; and you probably have a good slide.

Find a graphic of some type that will help you explain your claim. For some reason, audiences tend to believe you more when you have a picture, photograph, spreadsheet, equations, block diagram, schematic, graph, etc., and not just words. Anyone can write words. (Of course, anyone can also plot a graph, but it just seems less likely that one would go to that bother if it were not true.)

Graphics tend to focus an audience's attention to the slide where your claim is burning itself into their brain through their eyes.

Create three to five "bullets" that explain or otherwise defend your claim. Bullets are abbreviated sentences, not cryptic speaker's crib notes. They need to defend your claim without you saying a word.

Quote, "But, if I put my key messages on the slide, they won't listen to me!" They are not listening to you anyway. If you ever take a teaching course, you will find that people retain greater than 80% of what they see but less than 20% of what they hear. You want your claims and messages burning into their eyes as long as possible.

More importantly, much of the time you will be making the presentation to someone whom you want to carry your story forward for you. If all you have provided them are blurbs to remind yourself, they are unlikely to recall your points at all.

You should not have to say a word to defend your claim if it is a good slide.

For the types of customers in the Pentagon, cryptic slides will not even get you in the door. These people are very busy. It

is very common to be asked, “Would you like to reschedule in a month, or settle for ten minutes today?” In those cases, the style was, “Put your slide up for a few seconds, wait for a question, or proceed when grunted at.” You were not allowed to proffer a word (the infamous “speak when spoken to”). Further, at least one Air Force three-star general would famously ask that you put up your slides in reverse order. He knew everyone built up to his or her main claim at the end. If you had followed this chapter’s advice, you were usually welcomed back. If you did not, they brusquely terminated the briefing.

For the record, the author is aware that this portion of our advice is contrary to that expressed on page 127 of The Tongue and Quill, Air Force Handbook 33-337, 1 August 2004, which generally provides excellent guidance on all forms of communication. Obviously, the author’s Air Force and civilian experiences have differed. Nevertheless, we agree on much more than we disagree. As but one example, AFH 33-337 offers the useful guideline of a “7 x 7 rule”, i.e., no more than seven lines or bullets on a chart with each containing no more than seven words.

When presenting, do not read the slide back to the audience. Remember, your bullets and graphics already defended your claim. Your verbalizations should be tailoring or elaborating your message based on feedback from your audience.

Figure 6.1 is an example chart demonstrating the recommendations of this chapter, while belaboring our points regarding horse charts.

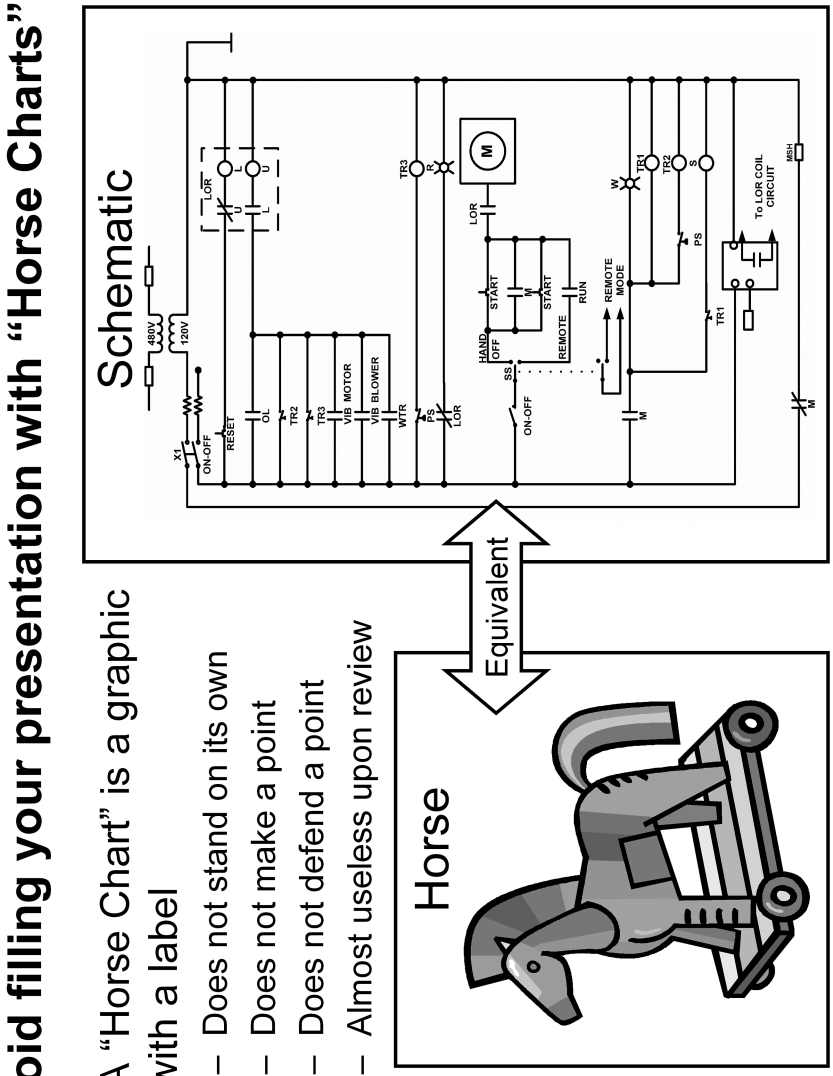

Figure 6.1 Horse Charts

Chapter 7 Find & Flush the Full In-Boxes

You will be amazed about how much good will, increased responsiveness, and cost savings can result by finding and flushing all the full in-boxes in your organization. We are not talking about all the spam in your email in-box, but the old style physical boxes on your desk where your undone work seems to be piling up. They are everywhere. You should be suspicious of any task that has the term "processing" in it. You will find them in Engineering processing bug reports, releasing drawings, reviewing drawings, assessing failures, etc. You will find them in Production processing vendor returns, receiving parts, updating assembly records, etc. You will find them in Customer Service supporting Material Review of defects, waiting to retest returned parts, doing over-reads at customer request, etc. You will find them administratively processing change requests, approving corrective action reports, etc. They are everywhere.

Groups with full in-boxes are invariably keeping up. They do not think that they are, but it is easy enough to prove. Just have them track their daily, weekly, or monthly input, output, and backlog over, say the last six months. I have yet to find the group whose backlog was growing. They are just stuck with some large backlog that makes their department disliked by all as non-responsive.

It seems to be a cultural thing. I do not know if it is satisfying to feel overburdened, if they think it implies they are busier than others are, if they get a sense of power by having others always beholden to their eventual action, or something else. The only explanation I have ever been offered is that they seem to worry that if their in-box were empty then somehow such would be wasteful. My response has always been that I would personally find them something useful to do and that they are at no risk for putting themselves or their staff out of a job. That is not a gamble because the data has already shown that that many staff were needed to keep up with their input. What *is* wasteful is the time delays and cost of money that these full in-boxes represent. It is enormous.

Luckily, it is a mostly one-time expense to empty in-boxes. Often you can do so with overtime, paid if necessary as it is worth it in improved productivity. Failing that, do not hesitate to hire temporary staff to flush them. Remember, this one-time expense has continuing recurring cost and responsiveness benefits.

Figure 7.1 shows a couple of examples from the Operations domain. "Waiting to Test" represents inventory that has been returned from the field and is awaiting retest to see if the field failure can be replicated or,

instead, is part of the typical 30 to 40 percent of returns that “retest OK” (RTOK). Obviously, one should work on reducing this percentage, but experience in several industries suggests that only much improved failure mode capture or other self-test holds out much prospect. Regardless, failing to retest before returning material to suppliers will just get your company a well-deserved Chicken Little reputation. “Material Review” represents inventory that has a known failure but that has not yet been returned to the supplier or to manufacturing for rework or refunds. This example starts with over a million dollars of inventory sitting idle, tying up cash, and slowing feedback from suppliers regarding failure modes and corrective actions.

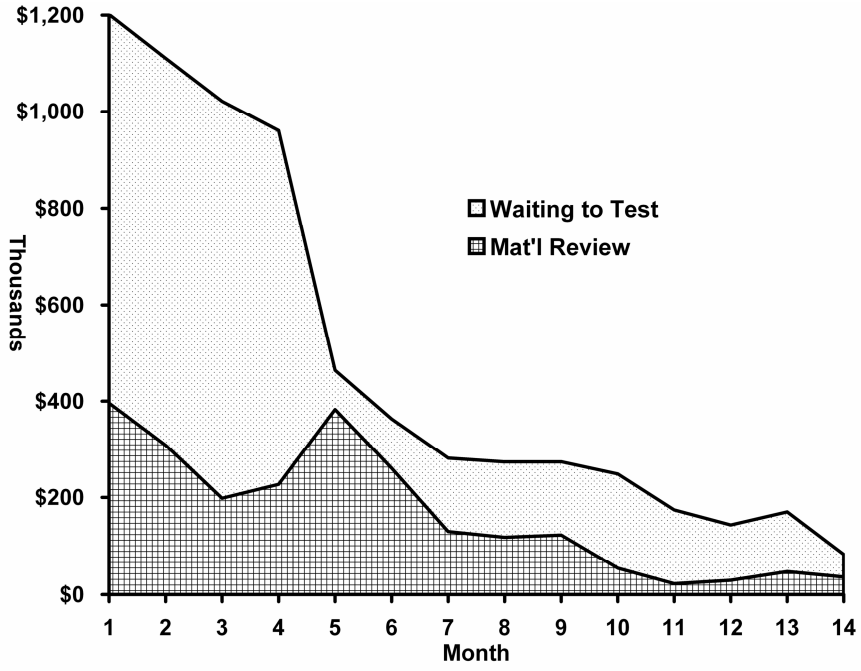

Figure 7.1 Full In-boxes

There is no black magic involved, just clear evidence of management interest and commitment of one–time resources.

Chapter 8 Continuous Improvement 101

Managing the quality of the systems and products you develop will involve much of your time. There have been myriad quality strategies over the years, but since the eighties, most everyone has adopted the continuous improvement principles most effectively espoused by Juran, Deming, et al that were adopted with a vengeance by most of the Japanese manufacturers, notably Toyota.

A key premise of continuous improvement is that defects are mostly management's fault; we failed to provide the right training, or tools, or instructions, or guidelines, or whatever. Continuous Improvement programs work**,** unlike many of the previous quality program attempts such as Zero Defects in the sixties and "Do It Right the First Time" in the seventies. The main difference is that those less effective programs focused on the person doing the work; they presumed that defects were invariably the worker's fault.

Another key premise is that the process is continuing. I recall being asked by Marketing, "To what percentage will our defects be reduced by the end of this fiscal year?" My response was, "I don't know because I don't have any control over what are the dominant defects and how much their elimination will buy us." On the other side of the coin, I was asked by a Production manager, "When do we get to stop?" The answer is, "Never!"

A third premise stresses focus on whatever has the most payoffs. Prior programs tended to claim that one was going to investigate every defect down to root cause, something no one ever has the time or money to do. An early step in any continuous improvement program is to quantify the frequency of defects. This is not as easy as it might sound, because what is needed is data in terms of actionable defects.

> For example, it is not sufficient to know how many printers of a particular manufacturer's model number that you had to replace last month. Let us say there were 12. If your total fielded population is 15, you have a big problem; if it is 1500, probably not. Still, you do not know enough yet, except to whine ineffectually at your printer supplier. What you really need to know is the specific failure mode (what some call the actionable defect), such as did the power light even come on; did the paper jam, was there a line defect in the printing, was there a large but local region of smudging, whatever. Now, if 11 of the 12 failed printers had a line defect, you might still have a big problem, even if you have 1500 fielded.

A fourth premise is that the originator is responsible for assuring the quality of his or her own work. No one is going to check it for them; you are no longer going to do incoming inspection, except to kick and count. However, there is a corollary that they better do their own inspection, including being given the tools and time to do so. The solution is not to inspect for them, it is to work reasonably with them to improve, but to drop them if he or she cannot or will not do their job.

No one ever *inspected* quality into a product. Not a new saying, but well worth repeating, over and over. By the way, no one ever *tested* quality into a product either.

Categorizing Defects

End-user or devices?

When collecting failure data, your first decision is likely to be whether to collect information regarding end-user failures or to limit yourself to device failures. You will also face service returns that do not exhibit any problems, typically 30-40 percent of your returns. There are many acronyms for this phenomenon: CND (could not duplicate), RTOK (retest OK), NTF (no trouble found), etc. If possible, capture the end-user failure information. You will rarely find customers lying. The notable exception is that they sometimes fib about changing configurations, hence the earlier advice regarding logging those events. At least track their difficulties, even if you do not plan to chase corrective actions initially. At the minimum, these issues can provide clues where you need to improve your manuals or help files. More likely, the most frequent are real problems you just have not yet duplicated in your lab environments.

Serial-number-specific or model-related?

You should initially treat hardware defects as if they are all model-number related, not random serial-number-specific defects. Software defects are rarely, if ever, serial-number-specific. That is, software is inherently model-number related. However, we are not advocating that you claim that you are going to make an extensive investigation of every failed device to determine the root cause. Remember that a root cause means just that. For example, determining a PC board failed because of a cold solder joint has not determined its root cause. You have to figure out why the solder joint was cold. Instead, you will investigate the most frequently occurring, provide a corrective action, and continue to work

down the list as other issues bubble to the top of the list of most-frequently-occurring or most-overall-cost-impact.

Hardware, software, and enhancements

It is not uncommon in very large companies to have separate management systems for hardware defects, software defects, and product enhancements. However, in many companies, one is usually happy to get even one change or defect management system working well, so our advice is to adapt it to handle all “defects”, whether hardware, software, or enhancements.

Severity and Urgency

The most common mistake in setting up defect classification schemes is mixing the concepts of severity and urgency. These are totally independent and need independent assessment. The following classifications in Tables 8.1 and 8.2 have proved useful over the years:

Table 8.1 Defect Severity Classes

Severity Code	Description	Example
1 - Safety	Problem affects the safety of the equipment or users.	Unprotected access to dangerous electrical power; inadvertent turn on of discrete outputs or moving parts; edges that cut.
2 - Inoperable	Problem renders equipment inoperable with no workaround.	Entire (sub)system (not just a particular function) hangs, requires reboot, will not function at all; commonly due to software or to missing parts because of drawing misinformation.

Severity Code	Description	Example
3 - Corrupting	Mostly, incorrect database entries with no means to fix.	Creates database entries that are unable to be corrected by an end-user through their normal graphical user interface (GUI). If the error can be corrected through some user interface, then classify as a 4, incorrect.
4 - Incorrect	Incorrect behavior: equipment works but one or more functions are not right without any workaround.	Function hangs or produces incorrect behavior with no workaround, i.e., inconsistent with our specifications, brochures, user manual instructions, "read me" files; incorrect parts called out or installed
4A - Absent	Customer requirement has not been implemented.	Functionality that is contractually required, but is planned to be missing from the current version of software or revision of hardware
5 - Workaround	Incorrect behavior: equipment works but one or more functions are not right. However, a workaround exists	Function hangs or produces incorrect behavior, e.g., a "delete" button on a screen does not work, but the keyboard delete key will execute the desired function.
6 - Cosmetic	Problem is the result of some defect in the appearance of the equipment.	Misspelled (but not misleading) text, inconsistent fonts, inconsistent surface finishes, etc. Ugly, but not misinterpretable.

Severity Code	Description	Example
7 – Internal Enhancement	A requested improvement. However, it is also deemed within contract scope.	Not a defect per se, but functional alternatives and additions that would be more convenient, less confusing, etc.
7A –Contractual Enhancement	A requested improvement that is deemed out of scope (OOS) to the contract.	Not a defect per se, but functional alternatives and additions that would be more convenient, less confusing, etc.
8 – CND (Could Not Duplicate, RTOK, NTF)	Defect cannot be replicated despite reasonable efforts, so work put on hold until more episodes provide better clues.	Retained in the system to minimize duplicate AR's and to remind all to continue to look for recurrence.

Severity is only a function of the character of the defect. As such, Engineering has the final say on that classification. Be particularly careful about any defects in the top three categories. There can be a tendency to label 4's as 2's, usually in hopes of getting them fixed quicker. One should never intentionally field anything with severity 1, 2, or 3 defects, and should try hard to eliminate any 4's, but sometimes that is not feasible. One can remain comfortable with that practice in conjunction with a policy of disclosure to customers of these 'known issues' as will be discussed later.

Declaring victory is a manager's best friend, particularly with software. The 4A subclass was introduced to distinguish between things that were intended to work in a particular version versus things that were not scheduled yet, for a variety of reasons. You should always assess any specification for whichever features are mandatory and which are just nice to have. One common criterion is that the new version must at least do what the existing product does, or more likely used-to-do or was-supposed-to-do. Your project plan should focus initially on this minimum functionality. If you are in a purely commercial environment where you can define the product, then typically the next version just has whatever is working robustly on such and such a day. If you have a development contract, then things are not as easy, but even then, you

and your customer can invariably agree on the minimum behavior for what some call “day zero” functionality.

Enhancements (#7) are not really defects, but you can manage them the same. You will find you need to retain almost the same types of information. Moreover, many of your enhancements probably came to you originally as a purported defect. If you are in an organization that does contracted development, then sub-class 7A has been found useful to distinguish enhancements that require the outside customer’s authorization and funding. While common in aerospace, most small to medium size companies do not have separate formal mechanisms for managing configuration changes. Just adapt your defect tracking systems. It has most of what you need.

> **Simple breakeven calculations are recommended to prioritize enhancements.** There is extensive business literature regarding return on investment (ROI) calculations. Besides requiring a lot more effort, the author’s main problem with ROI calculations is that their assumptions are too easy to manipulate to get almost any answer that you want. Instead, breakeven calculations only require that you estimate the development cost and the unit cost savings or additional unit sales that the enhancement will provide.
>
> Most breakeven assessments will simply be based on selecting the enhancements that have the lowest number of additional unit sales required. Unfortunately, you will commonly find that Sales and Marketing departments will not commit to any additional unit sales as the result of all those enhancements that they have been so vocally advocating. Instead, they will invariably argue that they will lose sales if they are unable to offer them competitively. Likewise, while still somewhat subjective, it is a bit more straightforward to assess enhancements that improve reliability or serviceability as these can be prioritized by simple breakeven calculations in terms of savings in Service costs.

Be brutal in your classification of a “bug”. “Better” is not a bug. First, it has to be a problem that appears likely to affect all instances of a model number, not just a defect related to a specific serial number. The latter are indeed still defects that may turn out to be more pervasive than initially evident, but you will invariably have so many examples of the model-number-related issues to address that it is doubtful you will have the resources to address apparently random failures. Second, it has to be strictly non-compliant with the product’s documented requirements. In many cases, you wrote those requirements. That is one reason you

commonly find “read me” files with software releases. The seller is simply redefining what the product is required to do. Any use of terms like easier, nicer, faster, whatever-er is a good clue that this is not a bug.

CND (#8) is also not strictly a severity, but you will need a means of filtering those from status reports. You could also use the “watch” urgency classification to be discussed in a bit. Some organizations cancel these CNDs, but that invariably just leads to myriad re-entries and re-cancellations. In addition, deleting these defects paints an overly optimistic picture of reality. One should rarely doubt customers claims, and then usually to your regret. Keep them around so that one can track their overall frequency and avoid lots of paper churning. This separate classification lets everyone know that no one is really working on them until better evidence becomes available.

Table 8.2 Defect Urgency Codes

Urgency Code	Description
1 - Immediate	Further development and/or testing, or customer operations cannot occur until the defect has been repaired. The defect is usually due to safety or system-wide shutdown; the system cannot be used until the repair has been affected. Assigned staff should immediately stop whatever else they are doing and work on this problem.
2 - High	The defect needs to be resolved in the next planned build. If your current task has lower urgency, put it on hold, and do this next
3 - Medium	The defect should be resolved in the normal course of development activities but will be required prior to project or customer signoff
4 - Low	The defect should be repaired as time permits, but typically before customer retention moneys can be invoiced
5 - Watch	CND defects will be monitored for recurrence, but not canceled until sufficient additional incidents provide clues allowing replication

Severity says absolutely nothing about the speed with which a defect should be fixed; that is defined by urgency. Urgency is set by program/project or product managers, not engineering. Of course, it is very likely that they will elect to address the most severe defects first, but it is quite possible for a cosmetic defect to have a relatively high

urgency, particularly when a customer's customer is involved. Said another way, severity is a technical call while urgency is a programmatic call.

Your urgency default should be low. Too many program managers cry wolf by setting urgencies far too high. They will invariably just be ignored. One useful approach is to require program managers to authorize fixes with associated budget. If they are not willing to pay for it, by definition the urgency is low. The downside of this scheme is that you can end up with extremely mad customers of cheapskate program managers that you will eventually have to assuage by fixing the defects anyway.

Violently reject any attempts to save money by not fixing defects. It will not. That is why there is no "ignore" urgency classification. Customers talk, particularly in this day of web blogs and "unauthorized" user forums. Most contracts have some form of payment retention, so the golden rule will burn you. Nevertheless, you need to prioritize and schedule fixes, and you can defer enhancements forever until someone is willing to pay for them.

To clarify further the distinction among severities, the following Table 8.3 is an example "known issues" table. You should be a strong believer in disclosing known issues to customers. You will hear many counter arguments, such as you are fueling your competition, scaring your customers, etc. However, it is also likely that nothing personally makes you madder than to spend hours with a problem and then have your supplier's Customer Service say, "Yea, we already know about that." Actually, many customer service departments will not admit it even if they knew, but that is even more infuriating.

Table 8.3 Known Issues

Severity	Known Issue	Workaround
4	If a reading is out of range, i.e., <4ma, >20ma, or has a bad data flag of 22, message contents are discarded. The website should present an exception message such as out of range high or low.	
5	When opening the user page to edit, the privileges default to ADMIN no matter what privilege they had before.	Confirm the User privileges are as intended before saving.
5	When you create a sensor type that can be used with a tank, such as depth or temperature, the web site generates a set of tank-related derived readings, whether appropriate or not.	Ignore the extra readings.
5	When the reading history table is pivoted to horizontal, the unit conversions do not work.	Pivot the table back to vertical to change the units being viewed.
6	Fonts on Owner Summary page are not consistent.	
7	Authorized agent page during Owner's first time registration does not have a field for pager E-mail.	Open Edit Agent field to enter pager E-mail address.
7	Negative values for volume and mass are displayed on the asset summary page.	Check the tank and sensor parameters to ensure they are defined correctly. A filled depth reading greater than max height could also result in negative values.
7	When a session times out during the start-first time process and the User clicks the save button, they will get a blank screen.	Complete start-first time before the session times out (approx. 20 minutes). The session time out is designed for security purposes.
7	Cannot delete an erroneous name, e.g., for assets, users, groups, etc.	This is by design. Subsequent versions will allow hide/unhide. Until then, when you need to add a "new" whatever, instead "modify" this erroneous entry instead.

Engineering Metrics

Figures 8.1 and 8.2 show typical quality metrics for Engineering. Bugs are considered critical if their severities are 1, 2, 3, or 4, but not 4A, and severities 7 and 8 are understandably not even in these counts. This particular product suffered from that syndrome of saving money by ignoring bugs discussed earlier. Regardless of what Marketing or Program/Product Management says, you will never get staff serious about squelching bugs in new releases if they know you will let them slide when they are old enough. It does not matter if the features are infrequently used. If it's wrong, it's wrong.

However, **be careful that you are measuring what is truly important** because your staff *will* modify their behavior to make themselves look good in your metrics. One classic example of what not to measure was "lines of code per time period per developer". While conceived as a measure of individual productivity, all it commonly led to was a lot of cosmetic code content with minimal value. In fact, efficient programmers were disadvantaged. Trying to develop individual metrics is usually a bad idea, not because there is not a difference between programmers, but because there are other factors that invariably have greater influence. Another ill-fated attempt was to measure defects per time period per programmer. Invariably, your best programmers will measure as the worst… simply because you always assign them the toughest tasks. Now, if there was an easy measure for "how long does it take a programmer to fix a bug after it is found and they are allowed to work on it", your good programmers will invariably shine on that one. If you need to grade your programmers, just sit through several of their design reviews. It will become quite evident who considered appropriate alternatives, who was efficient in implementation, who was methodical and thorough regarding exception handling, who follows your company design practices, etc.

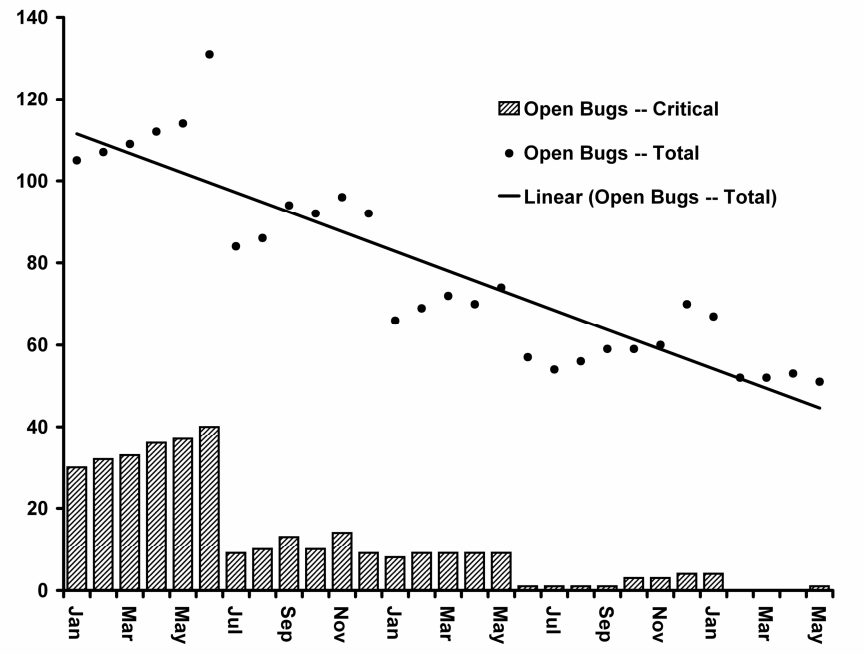

Figure 8.1 Bug Quantity

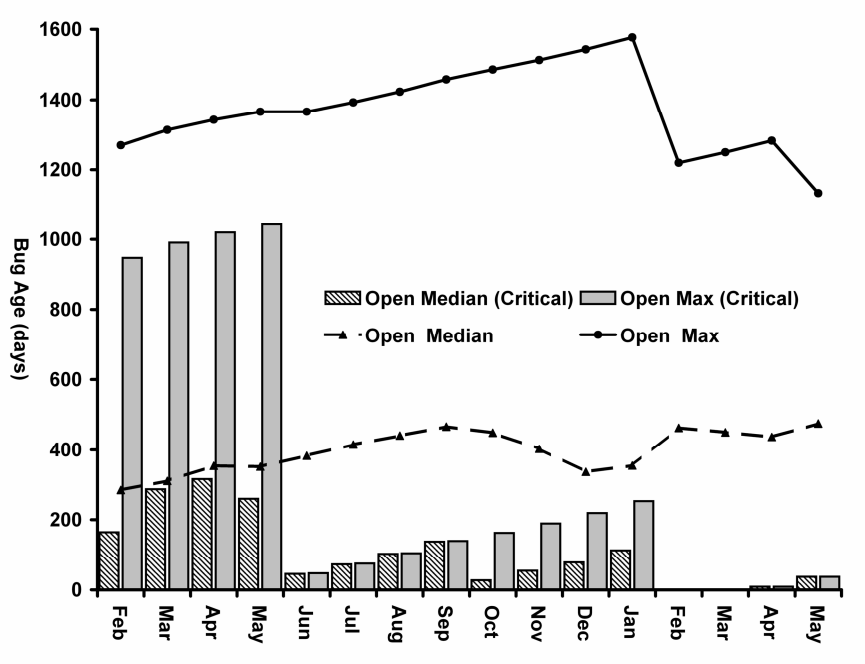

Figure 8.2 Bug Aging

Medians, rather an averages or maximums, are the best indicator of the age of defects. Averages are dominated by maximums, and you will often have some very old bugs, although hopefully not as bad as this example. Again, the focus should be on severity, not age alone. Regardless, given comparable urgency, one should work on the old one's first. As noted before, you should also ignore anyone who claims that critical bugs have low urgency.

In these examples in Figures 8.1 and 8.2 of a mature hardware system product, over one third of the bugs were both critical and over three years old. This mature product was getting updates about twice a year. Once focus was applied, in the next release the median age of critical bugs dropped from years to a few months. A couple more releases cleared out the rest along with addressing some of those less critical as well. One side effect of this focus on the critical bugs is that the overall median age increased a bit, but that is less of a concern as it can simply be fixed with more resources if desired. One should also note that these illustrate the typical behavior of each release causing a relatively large drop followed by a slow growth in quantity until the next release. Despite your best efforts, the slow growth usually is the result of new bugs introduced along with the new features and fixes in the release.

Production & Service Metrics

Pareto is your friend. Quality professionals seem to love to use the term "Pareto analysis". You have done it all your life. It simply means to rank order your items and attack the most frequently occurring first.

The 80/20 (90/10?) rule is a Pareto corollary. You *will* get 80 to 90 percent of a result from 20 to10 percent of the effort if you focus on the primary causes… and such is usually quite sufficient.

You will invariably have to develop some standard error codes. When you first try to develop some quantified failure history, particularly from the field, you will find that your service staff has very creative ways of describing the same failure mode in radically different terminology. This assumes you have gotten through the initial obstacle and gotten them to record something other than the parts that they removed and replaced (R&R). You can have senior staff review these field reports and manually categorize them initially, but that is rather inefficient and somewhat judgmental. Regardless, if you have to, do so and get the Pareto process started.

You will need to normalize the raw occurrence data to determine the dominant failure rates on a product-by-product basis. You can

use monthly production quantity to normalize Work-In-Progress (WIP) defects, monthly install quantity to normalize install defects, and total fielded population to normalize post-install rates. Figures 8.3-8.6 illustrate some typical production and service quality metrics, again for sophisticated hardware system products. Note that one can average more than one defect per system, so it unfortunately is not uncommon to have more than 100% defects per system.

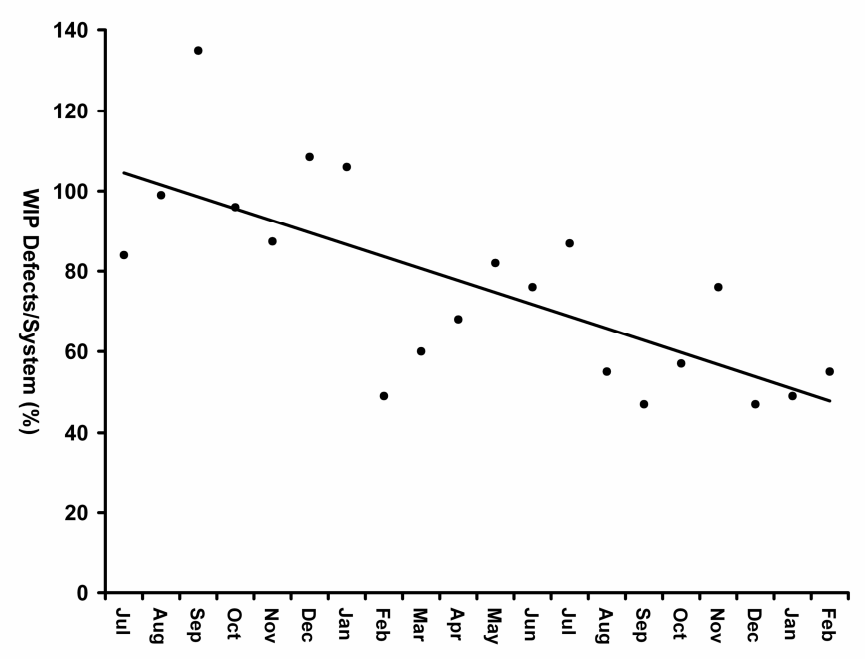

Figure 8.3 Work In Progress (WIP) Defects

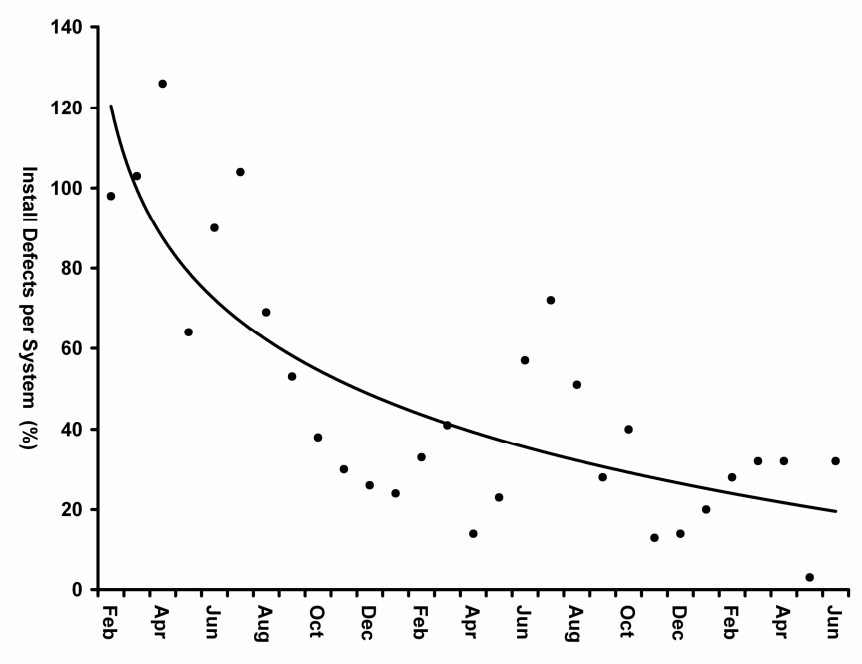

Figure 8.4 Install Defects

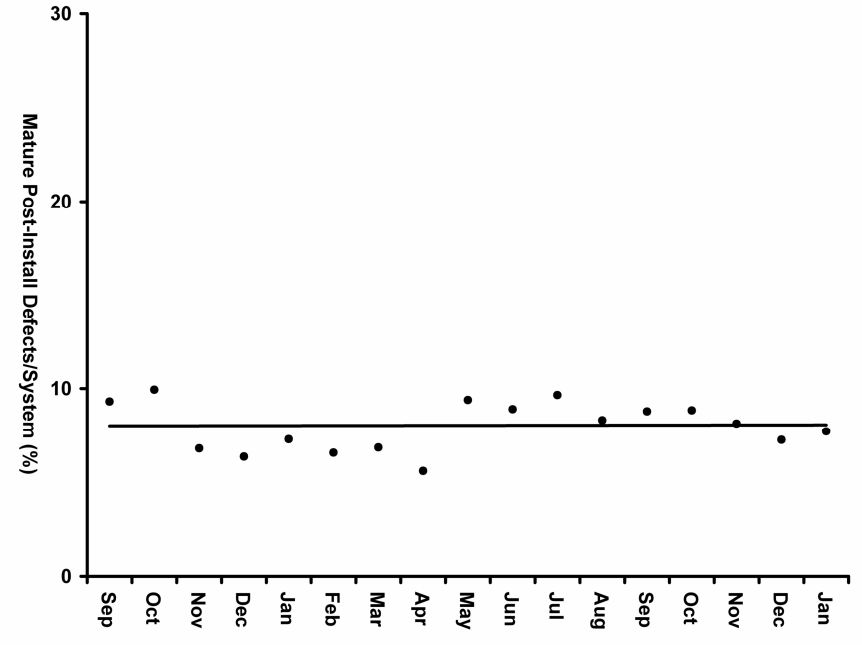

Figure 8.5 Mature Product Post Install Defects

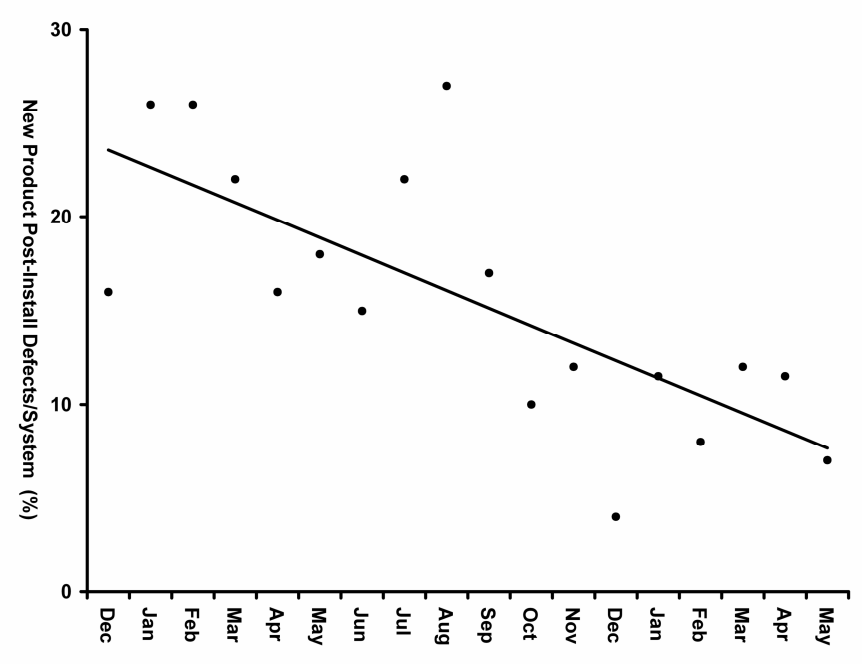

Figure 8.6 New Product Post Install Defects

It is difficult to affect the post-install defect rate of mature products, as shown in Figure 8.5. They "sorta are what they are". R&D investments to reduce the dominant failure modes are usually better spent on the next version of the product. About all you can do is to try to make the new product's key parts backward compatible. It is usually not easy. Sometimes the dominant defects are truthfully cosmetic or due to overly sensitive error reporting, but it is often hard to convince customers of that. So, use software to mask these types of defects, as they legitimately have no effect on customer function. For example, in many applications, a sensor defect of a few pixels or even a single line will have no practical effect on the calculated result, so it is perfectly legitimate to also mask those defects from a user display.

Most field defects are the result of failures in practice or process, rather than design. That is lucky for us, because design takes much longer to fix. So, what are some practical steps in developing corrective actions for the items currently on the top of your ranked list of defects?

Pareto Step 1: Eliminate the opportunity for error by eliminating the task. Examples include cloning so-called gold disks instead of manually installing operating systems, application software, and configuration files; quitting double-checking printers and monitors if you do not change them at all; and building products into finished goods that are "vanilla", that is, not customer or site specific.

Pareto Step 2: Move the task to where it is not time-critical. Errors often result when staff are rushed, such as at the end of any fiscal quarter at most commercial companies. For the uninitiated, it is not uncommon for ¼ of a fiscal quarter's shipments to occur in the last week of the quarter. In addition, for some reason, the bulk of service calls occur in the last couple of hours in a day. Examples include creating country-specific destination kits and common symptom-specific service kits that can be prepared in advance of need.

Pareto Step 3: Define a new tool or aid. Examples include product-specific accessory boxes so missing items are clearly visible and the use of torque wrenches in lieu of "finger-tight".

Pareto Step 4: Quantifiably determine what makes some workers better than others. Invariably, they are better because they have personally developed some different practice, skill, or knack that you can then transfer to the others by training.

Pareto Step 5: Clarify or create better work instructions. For example, production and service staff can have their own configuration and test screens built into products.

Pareto Step 6: Retrain the staff. I hate to even list this. In many places, it is the most commonly claimed corrective action, but it is mostly an admission of management failure to determine root cause.

Revisiting the example metrics, Figures 8.3 and 8.4 demonstrate that this continuous improvement process works for WIP and install defects, even on mature products, as they correspond to the same product shown in Figure 8.5. As you would expect, Figure 8.6 shows it even has benefits post-install on newer products.

Chapter 9 Performance Ranking 101

While it is expected in most companies to "pay for performance", the issue becomes what does one use as an equitable measurement standard? I first encountered the scheme of staff ranking at General Dynamics. In those days, we successfully merged the rankings of a two thousand-person Engineering group. I am particularly comfortable with both its equity and effectiveness. The key is that each person is assessed with respect to *what is expected of someone of that specialty in his/her labor grade*.

As a slight aside, **ranking first became critical in academia during the Vietnam era.** Until then, the median grade point average (GPA) for undergraduates at most colleges was in the 2.7-2.8 range with the top ten percent having a GPA above, say 3.2. Vietnam-era grades became substantially inflated, probably to keep students out of the draft, such that post-Vietnam the median GPAs were in the 3.2 range with the top 10% above a 4.0. So, how did you decide whom to accept into graduate programs when you had candidates that were both pre- and post-Vietnam? You ignored GPAs and looked at their class rank. Rank does not change with grade inflation.

The analogy to grades in personnel matters is ratings: exceptional, outstanding, typical, whatever your particular Human Resources department has called them. These have a tendency to inflate as well. It is worthwhile to note that we now have at least two generations of staff that have *all* been told their entire life that they were above average. You can see that is going to create a problem, even with engineers who intellectually know that half of any group must be below average by definition. I once had an Engineering manager tell me that something was wrong with any system that required you to tell half your staff that they were below average.

When the author went to school, a "C" was a good grade. Most of your staff now consider that an "F". Nevertheless, you are also trapped with the reality that you have to pay half of your staff below the average. Moreover, accept as a fact that everyone knows the average pay increase each year in your company. As such, I have stuck with my story of "'C' is a good grade" and focus instead on assuring equity by using ranking. The alternative is to tell them with ratings that they are above average while paying them inconsistently.

Ranking is one of the primary mechanisms to assure program and product managers have comparable influence with functional managers regarding the rating and merit increases of all staff. The process starts with each functional department manager ranking his staff from top to bottom. The key step is an integrated ranking meeting where both engineering and program managers will merge these departmental lists. You can facilitate this process by distributing index cards and data to your managers that contains the name, department, title, and labor grade of each of your staff. The departmental managers can then annotate their staff ranking onto the cards so that everyone doing the ranking knows how each originally assessed a given skill set. Merging the departments will usually take about half a day. You might have a neutral HR person serve as facilitator for this process. The main objective is to assure the free and open feedback among these peers.

You will encounter both hard and easy graders, so you will collectively be moving other manager's staff around to assure equity. About the only problem will be a very few staff where there will invariably be dramatic differences in viewpoint between the functional and project managers. These can go both directions. What you need to assure is that all are forthright with each other and subsequently to include some of these expressed specifics in that individual's review, irrespective of where they are finally ranked by the group.

When the ranking process is complete, some equity crosschecks need to be performed. Ideally, one looks for each labor grade, each project/product, and each supervisor group to have someone near the top, someone near the bottom, and an average ranking near 50%. The results will not be perfect, but any residual inequities should be real, such as a particular project that has been shortchanged. These may be a bit of a revelation, and you should strongly reconsider the related assignments.

Based on the rankings, you then need to decide where to draw the line between the performance ratings. This is not unlike the job of a teacher deciding where to draw the line between the A's, B's, C's, etc. It depends on the number of rating classifications within your company. In a company with theoretically five ratings, you would expect the top group to end near the 20^{th} percentile, the next near the 50th, with the bottom somewhere in the 80's. Anyone that belongs in the fifth category should have been put on a corrective action program long ago.

It has also been found useful to identify "key" and "high-potential" employees, perhaps about 10-20% of your staff. Key staff typically possesses unique experience that is unable to be reasonably hired, learned in a year, taught in schools, etc. Classically, this code is used

by HR and senior management to make sure key staff are somewhat protected from extreme acts. Hi-Pot's are defined as having *at least two* promotions left in their career. These are your future, so their reviews should be carefully assessed to make sure they are being groomed, sent for training, and the like.

Now, you can estimate an equitable merit increase by using a simple straight-line relationship with ranking. Someone ranked at the 50th percentile would nominally deserve a raise equal to the pool average. The person at the top should get double the average, and the person at the bottom gets zero. However, it is not quite that simple because one also needs to consider "penetration". That is, how does what the person currently makes compare to his peers in the same department in the same labor grade?

One can then take the estimate based purely on performance and either increase or decrease it, say, by forty percent of the pool average if one were 0% or 100% penetrated, respectively. For example, using a pool average of 5%, the person at the top of the ranking would get 10% if they were currently making the average salary for their labor grade in their department. If they were 100% penetrated, they would get 8%, while if they were grossly underpaid compared to their peers and only making the range minimum, they would get 12%. If they were ranked as the middle performer, they would get 7%, 5%, or 3% if they were penetrated 0%, 50%, and 100%, respectively. Next, the estimated merit recommendations are clipped at the bottom typically so the smallest raise allowed is 1 to 2% for salaried staff or, say, $0.25 for hourly staff. Figure 9.1 shows a typical result with the desired effect of clearly paying for performance.

Finally, these mathematical estimate guidelines are just that, estimates. Management judgment will still need to be applied to lead to meaningful rounded raises, to somewhat offset penetration issues that are justified, such as remote staff in high cost areas, and the like. However, one should not deviate very much without reassessing either the person's labor grade or ranking. It is sometime rather painful to pay for performance. Nevertheless, it is absolutely necessary if you are clearly to reward those who are making the most contribution to your company's success.

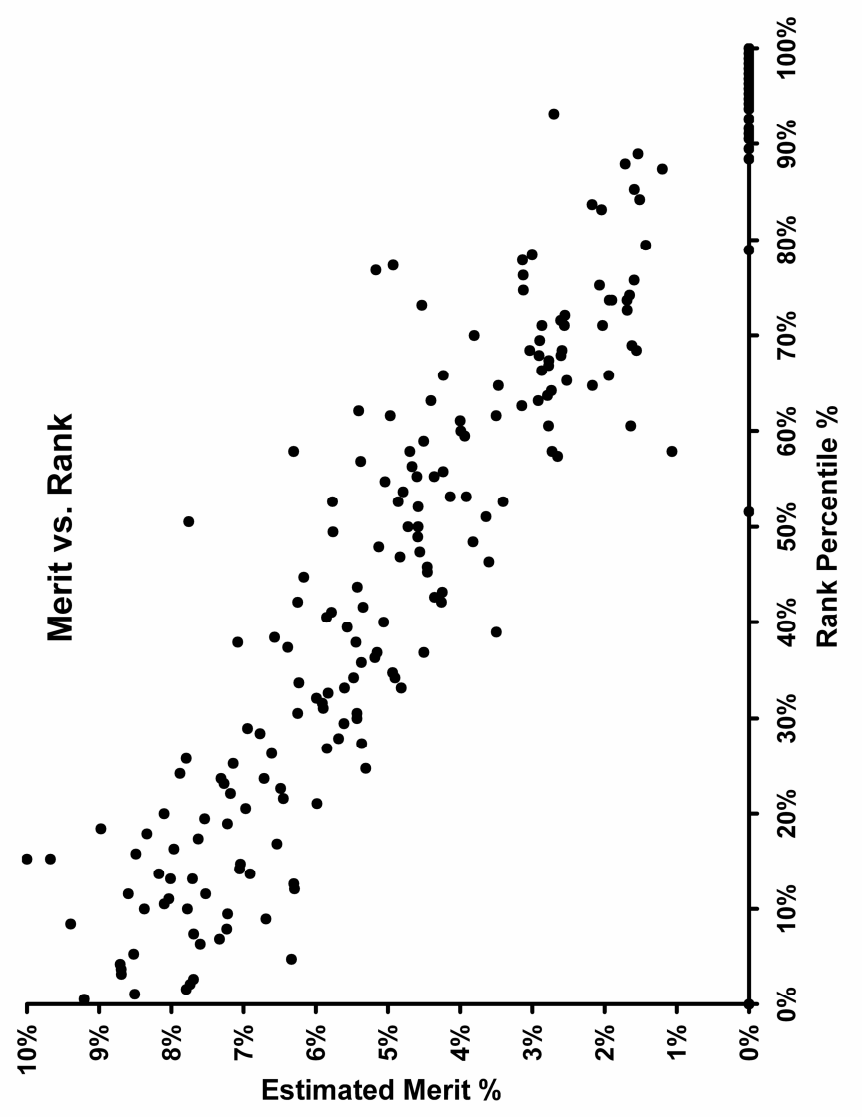

Figure 9.1 Merit Pay versus Rank

Chapter 10 Incentive Criteria 101

As you progress upward in responsibility and labor grade within an organization, you will often encounter some incentive or bonus plans. Initially, you will not have much say in its details, but eventually you may. Most companies have some sort of quantified incentive criteria so that it is not just viewed as a beauty pageant or teacher's pet. I recommend three classes of objective measures: individual financial success, overall division success, and mid-term improvements. Each class needs a target value, preferably with scaled awards ranging from zero to, say 150% of target.

Individual Financial Success: Most staff has a clear, direct budgetary (cost and schedule) responsibility for a specific program(s)/project(s)/contract(s), product, or functional department. Some will supervise those with direct responsibility. Supervisors' objectives can simply be a weighted average of the incentives earned by subordinates. Meaningful quantified objectives for those directly responsible for execution are thus the key to this incentive strategy. Objectives are being rolled up, rather than flowed down. This is a bit unusual, but improves ownership in the results.

An annualized Cost Performance Index (CPI), excluding rate variances, is proposed as the most meaningful indicator of individual cost control. Recall, CPI is simply the ratio of the Budgeted Cost of Work Performed to the Actual Cost of Work Performed, or CPI = BCWP/ACWP. Use of an annual measure lets each staff member start the year with a clean slate. Rate variances are excluded since most staff can only directly control person-hours and other direct costs (ODC). Other members of management are held accountable for rates and indirect costs. One should also note that this strategy has the benefit of penalizing the holding of excessive funds in management reserve or the working of unplanned activities since such would limit everyone's performance. You cannot get credit for the work unless it is planned and budgeted.

Figure 10.1 is an example incentive formula. An obvious question is "Why consider just meeting budget worthy of a more than target payout?" The answer is that you will never see a program with substantial development and initial shakedown ever come in under budget, even with the best of teams and management. Obviously, such execution performance on basic contracts mandates that you make sure that your enhancement and service activities must provide offsetting additional profits. Another obvious question is "What about schedule performance?" Such invariably reflects rather quickly into cost. In

addition, if you recall from the earlier discussions in Chapter 2, a poor Schedule Performance Indicator (SPI) can sometimes be intentional.

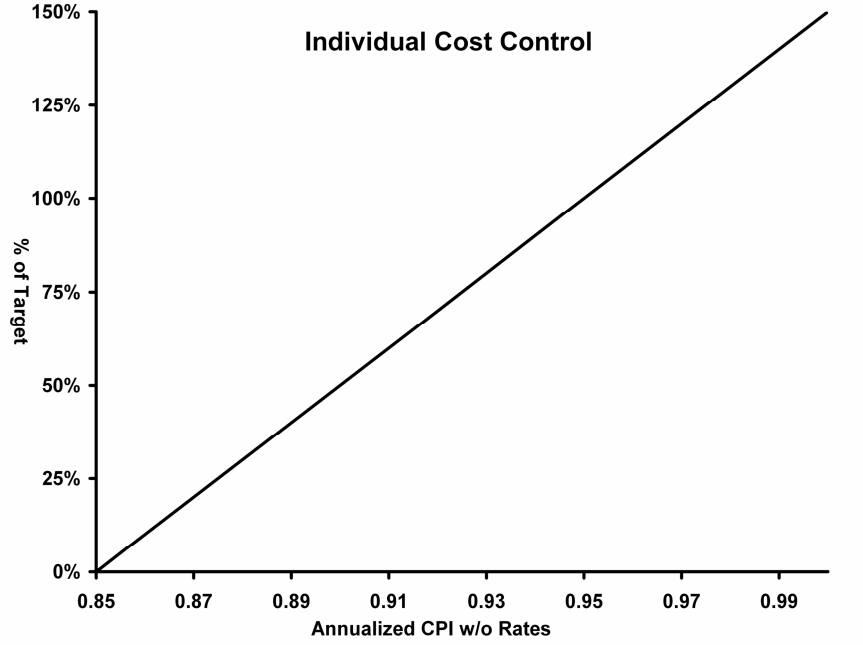

Figure 10.1 Individual Performance Incentive

Overall Division Success: Unfortunately, if each staff focused solely on their individual cost performance, it could have very detrimental effects on the company overall, whether characterized as sub-optimizing, fiefdoms, or other undesirable descriptors. You must equally make sure that your staff's efforts provide the support and consideration of the needs of their internal customers and peers. For example, if some staff provided earlier test software that did not depend on end user features but only checked hardware functionality, your production staff could save substantial carrying costs and improve the division's profitability, although the effort obviously represents costs to an engineering and program manager.

The second objective simply measures your Division's profitability versus the fiscal year business plan that is committed to Corporate. Figure 10.2 shows an example formula. Note that the payout is faster for profits above the 100% target than the decreases for profits below. However, one must also note that it is not uncommon that if you fail to reach even, in the example, 70% of plan, then all bonuses, not just this measure, can become inapplicable.

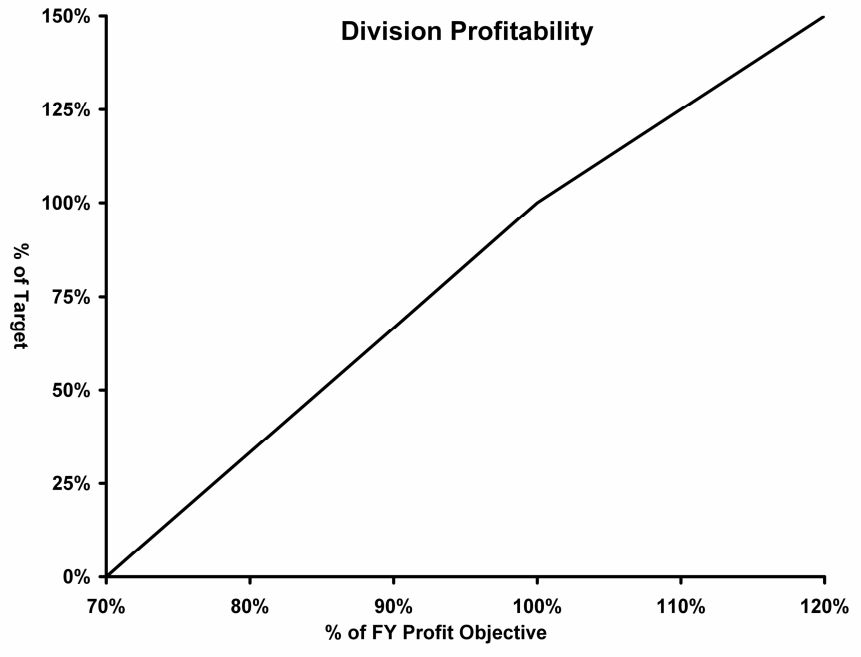

Figure 10.2 Group Performance Incentive

Mid-Term Improvements: Focusing solely on either of these near-term financial measures could also lead to myopic behavior by failing to invest in efforts providing longer-term efficiencies. Thus, engineering management objectives needs a third class indicative of support for peers and improved practices. In effect, you need to motivate them to invest in activity this year that will mainly have benefits in the years thereafter. By the nature of your staff's roles, these are of necessity specific to their individual areas of contribution. As such, one might ask them to define them for themselves. The ground-rules can be relatively simple: the results must be achievable in the year and must be worded in a manner assessable objectively by a third party. Obviously, you should expect them to focus on items that have high payoff and represent non-trivial or non-routine efforts. You should advise them to select several to improve their chances for maximizing their incentives, since these typically will be of the form of yes or no regarding achievement.

Some examples could include:

Support for other functions:

a. Reduce spare part numbers on yy product by xx percent
b. Reduce manufacturing costs on yy product by xx percent
c. Reduce part count on yy product by xx percent
d. Create built-in test hardware points with xx percent coverage or test software independent of end user functionality on xx product
e. Reduce average response time in processing whatever paperwork by xx
f. Redesign out the top xx customer support failure modes on product yy
g. Redesign as needed to improve MCBF by xx percent

Support for improved practices:

h. Create, document, and implement formal hardware de-rating criteria
i. Create, document, and implement software design practices
j. Create, document, and implement GUI design guidelines
k. Create, document, and implement appropriate quality metrics for engineering
l. Develop parametric estimating tools and/or factors so that proposals can be planned and man-loaded based on high-level technical parameters rather than excruciating and questionable bottoms-up details.

My recommendation is that you give equal weight to each of these three classes of measure: individual, group, and mid-term. However, that will depend on your company's culture. Most U.S. companies focus on the first class, most Japanese companies focus on the second, and when I was working as a functional manager, I needed some hook to get staff to work on the third. Do not worry; there is not a wrong answer. These are all a bonus after all. Just be careful that you really intend to emphasize whatever you finally choose.

Chapter 11 Matrix Organization 101

A matrix organizational structure's key feature is shared responsibilities in order to provide management the checks and balances needed to assure simultaneously meeting both the short and mid-term objectives needed for profitable growth. Matrix organizations are very common in aerospace, but much less so in commercial entities. In effect, all staff has two bosses: a program/product manager and a functional manager. Examples of the latter are managers of electrical engineering, software development, mechanical design, etc. It is called a matrix because this duality of bosses is usually shown as a matrix:

Table 11.1 Boss Duality in a Matrix

	Functional Mgr A	Functional Mgr B	Functional Mgr C	Etc.
Program Mgr 1				
Program Mgr 2				
Program Mgr 3		Skill B working on Program 3		
Etc.				
Product Mgr 1			Skill C working on Product 1	
Product Mgr 2	Skill A working on Product 2			
Product Mgr 3				
Etc.				Etc.

In practice, most programs and products employ several staff of a given skill set, so that it is only the "lead" for that skill which has two bosses. Most staff members just take direction from their functional lead person.

Companies in the business of delivering complex systems composed of somewhat common products found that **dedicated program or dedicated functional organizational structures led to behaviors that were contrary to the company's well being.** So, where does the conflict come from? Program managers are expected to deliver systems to their customers at the least cost and time to maximize the company's short-term profitability. Functional managers are to assure the

timeliness and quality of their staff's contributions and the continuing professional development of their staff.

Left to their own devices, program managers can be tempted to parochially retain key staff far beyond their needs, precluding their professional growth. They will rarely spend one penny extra on their design in order to make sure it is easily adapted for subsequent usage by others. They can be swayed by staff to adopt the development environment de jour, even if that means they will not easily be able to make use of other staff that is unfamiliar with their unique choice. On the other hand, functional managers have been known to want to employ every new technology coming down the pike and can be more enamored with sophistication and complexity to demonstrate their prowess, rather than cost effectiveness.

A matrix structure addresses these conflicting overall perspectives by turning them into a continuing series of minor skirmishes. Let us consider that management has to continually answer four questions: what, when, who, and how, while dealing with two resources: $'s and people. In a matrix, program and product managers are responsible for "what" and "when" and control the dollars. Functional managers are responsible for "who" and "how" and control the staff. In effect, neither can do anything useful without the consent of the other. The ensuing agreement is documented in a jointly negotiated program plan, such as in Microsoft Project® or in Primavera®, as discussed extensively in Chapter 2.

Keeping these internal contracts current is absolutely critical to keeping each other informed and enabling each to do their respective jobs. It is equally important that both parties continually renegotiate in good faith and not agree to disagree. The latter leads to meaningless plans that are, at best, poor financial scorecards of both their failures. Instead, these plans should serve as the mutually pro-active means to communicate actions and status with themselves and with senior management. These contracts, embodied in plans, are the key to achieving the balance of objectives that a matrix can provide. Such planning is the key to being managers, rather than reacting as fire drill monitors.

Program managers, in most companies, would be delivering systems by tailoring and/or configuring products that were developed by product managers spending R&D funds. Product managers typically get those funds by negotiating a specification and budget with senior management. Senior management in effect forces this subcontracting to occur by usually insisting that program managers make use of internal products. Just as program managers, product

managers negotiate for staff and tasks from functional managers, except they are spending company R&D dollars rather than customer funds.

Note that the product manager's role also fits on the program rather than functional side of the matrix. That is, they control the what, when, and dollars, with the objective of simultaneously addressing the product needs of all the program managers while minimizing overall development costs and risks. In effect, their primary role is to find and enhance the common core among all the program requirements and to architecture their products such that the various unique traits can be accommodated with minimal added effort. Particularly with respect to software, one would have to be an extreme masochist even to contemplate having separate developments for each project. A "product" approach is also central to reasonable factory production efficiencies and benefits all programs by acquiring meaningful field feedback that can improve ongoing production for subsequent customers. Remember, these products are usually non-trivial subsystems unto themselves that need good definition and focused management reconciling the demands of multiple customers, internal and external.

Collocation of a project team is beneficial, irrespective of your organizational structure. Several studies have shown that interaction and communication frequency is inversely proportional to the physical distance between staff members' offices, i.e., they talk more often if they are closer. In addition, collocation provides more pragmatic, day-to-day control to project management, even in environments where formal control is more functional. It is much easier to achieve esprit de corps, ownership, and a sense of urgency in a project or matrix structure, but collocation aids regardless.

Even in matrix organizations, over time you will see the pendulum of power vacillate from one side of the matrix to the other. Again, there is no right answer, but a matrix has its merits.

Chapter 12 Tailor Your Behavior to the Software, not Vice Versa

While generally a good practice overall, I will argue vociferously that such is the only path to success when adopting enterprise-wide systems, such as MRP, or ERP, or Configuration Control, etc. I have been around several unsuccessful attempts to adopt systems from SAP, but I am sure the problem is not unique to that vendor. Rather, it is from their approach. These providers generally tout that their system is so adaptable that it can be configured so that you do not have to change your existing behaviors. What they fail to emphasize is that you *have* to configure every little nit and lice of your behaviors. For the companies that I was exposed to, this generally meant dedicating at least one senior member of every department for something on the order of a year. This is a huge expense and does not even count the fees for the external implementation consultants with which this staff is interfacing.

Enterprise Resource Planning (ERP) systems are a good thing. If Finance, Engineering, Production, Service, and Sales are not working to a common database and tracking system, myriad home grown or specialized packages are either routinely inconsistent or duplicative at best. Management spends a fair bit of their energies reconciling differences between systems, while remaining unable to establish effective feedback mechanisms.

Find the package that is least painful and adapt your behavior to it. Notice that I did not say "the best". I am not sure what that even means. With my automatic controls background, I am very harsh with the common abuse of the term "optimum", which is the fancy semantics for "best". The one thing you learn quickly in automatic control courses is that there are as many optimums as there are optimization criteria. Therefore, my response is always, "Best in what sense?" Most cannot answer. Those that try quickly realize that there are many, often conflicting, criteria that they are trying to meet.

You should invariably be quite happy to find *any* solution. By the way, this statement applies in general. You will rarely have the resources to bother with "best".

Any of the software packages you consider will add substantial new functions that will benefit daily. Even if they did not provide missing functionality, which they do, just eliminating multiple entries of almost the same data and the associated attempts at reconciliation is worth the pain of modifying your behavior.

Configuring is needed, but avoid customizations, almost at any cost in behavior change. It is not even the cost of doing the customizing that is the problem, although that can be large as noted earlier. The real problem comes in a few months when your vendor releases his next upgrade with many new features that you were not even smart enough to know you were missing. However, the upgrade invariably stomps on your older custom features, so you again have the added expense of more customizations, or, worse, you forego the upgrade. The latter is a bad choice as the vendor understandably will eventually not even support older versions, and you have lost all the benefits of buying from a third party. They will keep providing substantial productivity-enhancing features due to their wide customer base. Internal systems just cannot keep up with this features race. Pick your favorite horse and ride them.

I've Never Found the Software that I'd Rather Write than Buy.

Buy almost any tools and middleware that you can. You will be assured by your staff that they can write it from scratch quicker than learning and using the tool, but do not ever believe them. By the way, they will tell you the same thing when they are asked to modify, fix, or enhance the software that another employee wrote. Again, ignore them. That existing code has stood the rigors of substantial internal tests and end use. It may mean a bit more hours of coding, but you will save overall.

Remember when you are down to a short list of choices, what matters is that you choose and get on with it. As such, what follows is a list of tools that I have found useful, but just consider them examples. They all have competitors that you should evaluate in the light of your own religious preferences.

Make sure you buy software maintenance. That does not preclude use of open source tools like Apache®, Tomcat®, MySQL®, and the like as most of the usage leaders have companies like Covalent that provide on-call support. However, it is important that you keep these tools current. Most understandably respond to a bug fix request with a demand that you update to their current version, as they have often fixed it in the interim.

The main reason to buy, rather than write, is that you would never keep up in the features race, even if you could match them originally. Typically, these new features, along with bug fixes, come with software maintenance contracts. Nowadays, I do not think anyone would be

stupid enough to write his or her own database program, but I had to fend that off several times.

Recommended Tools

GUI Prototyping: Adobe/Macromedia's Dreamweaver® has already been mentioned several times.

List/Table Middleware: Actuate's Formula One® is an excellent servlet that literally provides an Excel clone embedded into your Java applications, replete with graphing, sorting, math, etc.

Specialized Routines: Make it a habit to do an internet search for applets, servlets, and that ilk whenever your staff needs to do something outside your company's special knowledge. I have commonly used them for communication protocols, time setting, automated backup, file transfer, etc. They may be small in scope but, as always, they include tons of unique exception conditions handling that your staff does not need to learn the hard way.

Simplified English Checker: Boeing provides a tool that enforces this aerospace standard, but it should be considered by anyone producing manuals or help screens, particularly if you anticipate translation. "Simplified English" has a greatly restricted vocabulary and enforces readability. (My text would never pass.) A notable feature is that there is one and only one word available for a given action. For example, mixed usage of stop, halt, quit, end, etc. can be very confusing when you want to translate, or even in English for users who are searching for hidden meanings in your semantics.

3-D Parametric Computer-Aided-Design (CAD): I prefer SolidWorks® (because I am cheap), but PTC's Pro-Engineer® remains quite good. Alibre® is an even more affordable parametric solids package. These tools may require you to adapt your configuration management and documentation practices to accommodate the fact that it is now really the solid model that requires control, not any particular view. Regardless, the parametric flexibility for changes is worth it.

Database: Oracle® seems to be the defacto leader, which I have used many times successfully. Microsoft keeps trying by enhancing Sybase's SQL Server®. As an aside, do not pay any attention to software list prices, provided that you build systems where both vendors remain viable options.

Report Writer: Business Objects' Crystal Reports® seems to be the market leader, although there are several others that are worthwhile. I would particularly suggest that you focus on enabling end users to dynamically define reports on the fly with the cosmetics of their choosing. You will never keep up with their demands. Your job should be to assure that the database has the underlying information that they need.

> **You will also need a set of internal standardized reports to uncouple problems in the reports from problems in generating the source data.** Otherwise, your poor report staff will be deemed guilty until they prove their innocence when trying to debug and integrate new system features. It is an undue burden on them and wastes schedule that should be spent solving the real problems.

Automated GUI Testers: Emperix's e-Test Suite® has been effective. Its greatest payoff results from confidence that the testing is thorough and consistent, not just its speed. Without such a tool, people tend to test until they are tired. In particular, they get lax in testing all the exception conditions because they rarely encounter a problem, at least recently. The key feature you want to look for in these tools is that they actually interpret the underlying HTML code, rather than depend on matching localized bitmaps. The first generation of testers mostly did the latter and thus was sensitive to the slightest change in screen layout. As such, it was almost impossible to keep your test scripts current with an evolving or tailorable product.

Lint and Leak Detectors: IBM Rational's Purify® is a super code quality checker. You should run it, or a competitor, on any code before you even spend much time testing. It goes without saying that coders need to fix any errors and memory leaks, but I wish I had a nickel for every developer who said, "It's just a warning". You will be amazed how much more robust your application becomes when you make all those warnings go away.

Profilers: IBM Rational's Quantify® is a good profiler as is Quest's JProbe®. IBM is bundling Purify®, Quantify®, and a code coverage checker as Purify Plus®.

> Regardless, I have long ago given up trying to find anyone who can predict database performance in advance for a new application. That is, they can extrapolate an existing app fairly well, but they cannot estimate a new one. And, I have spent a fortune trying. My conclusion is that you mainly build the application with the flexibility and configurability you need, and

then profile it. These are tools that let you know where you are spending the most amount of time, the second most amount of time, etc. You then restructure the top 5 to 10 culprits to speed things up. If that is not fast enough, then throw hardware at it. For the record, I have never had an instance where the developer correctly guessed where his application was spending the most time. They usually were in the top five, but never the first.

As a corollary, so-called load testing is imperative as early as possible since you will not be able to estimate performance. Be particularly careful not to be too simplistic in your data, e.g., do not just duplicate the same data record thousands of times. Be sure you vary all the data fields so that the processing is duly loaded.

Closing Thoughts

Recapping some of my favorite advice…

That's a solution, not a requirement.

Ambiguity in a specification is always to the buyer's advantage

If there is only one feature of aerospace system practice that you can adopt, it should be *design reviews*.

Test to break it, not demonstrate it.

One manages "starts", not "finishes". You react to finishes.

Beware of the student syndrome.

Noah's Principle: Predicting rain doesn't count, building arks does.

There is no such thing as a constant (except maybe π and e).

GUI design should be viewed as a religious preference, not technical. However, it is important that you express your beliefs.

Why are you showing me that slide?

Groups with full in-boxes are invariably keeping up. Flush them (the boxes, not the groups).

Declaring victory (or the contract's changes clause) is a manager's best friend.

Violently reject any attempts to "save money" by not fixing defects

Be brutal in your classification of a "bug". "Better" is not a bug.

We now have at least two generations of staff that have *all* been told their entire lives that they were above average.

Tailor your behavior to the software, not vice versa.

I've never found the software that I'd rather write than buy.

When you are down to a short list, what matters is that you choose and get on with it.

Additional Reading

Augustine's Laws, Norman R. Augustine, 6th Edition, 1997, American Institute of Aeronautics and Astronautics, Inc, New York, NY. This book is always my first reading recommendation for any new program or product manager, whether in aerospace or not. Commercial bureaucracies are very similar, and politics is politics, whether national or corporate. Mostly a well-edited compilation of short articles originally published in the AIAA's monthly member magazine, the content is both insightful and hilarious. As but one more notable observation, Figure 31 on page 153 of the 6th edition shows the excellent correlation (over one hundred data points from various programs) of the fit between the estimated time-to-go and the actual time-to-go. This scheduling "fantasy factor" was found to equal 1.33, which is, not unexpectedly, rather close to the median overrun… further proof that time is money.

First, Break All the Rules: What the World's Greatest Managers Do Differently, Marcus Buckingham & Curt Coffman, 1999, Simon & Schuster, New York, NY. My favorite book on managing personnel, based on over 80,000 interviews by Gallup, the authors explain why one should not try to "fix" people but, instead, focus on their strengths and match those to your needs.

The Engineering Design of Systems: Models and Methods, Dennis M. Buede, 2000, John Wiley & Sons, Inc., New York, NY. This text focuses on applying the rigor of formal modeling tools and processes to systems development. A descendant of Structured Analysis and Design Techniques (SADT), IDEF0 (Integrated Definition of Function Modeling) is explained and advocated for the formal capture, evaluation, and analysis of requirements. Such modeling techniques should be particularly useful where, as in aerospace and defense, a very technically astute team (typically representing the buyer) is responsible for the independent verification and validation of systems whose elements and subsystems are developed by others. As a side note, there is an emerging development of an open standard systems engineering modeling language, called SysML (see www.sysml.org), which is a formal subset of the more widely known Unified Modeling Language 2 (UML2) that enjoys increasing usage in the software engineering community.

Product Design and Development, Karl T. Ulrich & Steven D. Eppinger, 3rd Edition, 2003, McGraw-Hill, New York, NY. This book focuses on the design process for commercial products, so it is a good starting point if your focus is on the engineering of systems, rather than systems engineering. Each chapter contains an illustrative case study, each from

a different industry having a variety of complexity, although predominately mechanical products. Most of the content concerns the processes involved in evolving to the most appropriate design approach, rather than the mechanics of design implementation. Notably, Chapter 13 is a good introduction to formal design for robustness.

Project Management: a Systems Approach to Planning, Scheduling, and Controlling, Harold Kerzner, 9th Edition, 2006, John Wiley & Sons, inc., New York, NY. This text extensively elaborates on the organizational and personnel issues discussed in the present Chapter 11 as well as generically addressing the planning aspects of Chapter 2. Its final chapter on applying Goldratt's Critical Chain process to project scheduling is particularly noteworthy of consideration. If you have not yet made the move to formal project management and organization, this text will get you started on the issues and solutions involved.

Systems Engineering and Analysis, Benjamin S. Blanchard & Wolter J. Fabrycky, 4th Edition, 2006, Pearson Prentice-Hall, Upper Saddle River, NJ. If your interest is in systems engineering, rather than in the engineering of systems, this is the text with which you should start. A systems engineer can be viewed as the surrogate for the external and internal customers of engineering. They first work with external customers to assure appropriate and effective requirements and then develop and assess alternative solutions. This text provides a good overview of the analytic tools used to perform formal analytic trade-offs among these alternatives. Systems engineers also assure due consideration is given to a full life-cycle consideration by engineering, in effect, by representing downstream internal customers such as production and service. As such, the text provides both insight and the associated analytic tools related to what are commonly called the "-illities": reliability, maintainability, usability, serviceability, producibility, and affordability.

The Art of Systems Architecting, Mark Mair & Eberhardt Rechtin, 2nd Edition, 2000, CRC Press, Boca Raton, FL. Recall our comment in Chapter 1 regarding the "black magic" of systems decomposition. This book provides helpful insights into that process by advocating that system developments require the full spectrum of skills of both architects and engineers, directly analogous to the roles played by both for centuries in the civil structure arena. The authors stress that a system architect's role involves art as much as science, but they provide the reader with explicit heuristics or guidelines that are broadly applicable. Almost 200 are included in their Appendix A after substantial explanation and elaboration in the main body. They also do an excellent job of explaining the rationale behind the differences between classical functional decomposition and modern layered object-oriented software,

particularly as it affects system modeling and design. If your focus is mainly early in the systems development process by representing and articulating the client's interests, this is an excellent place to start.

The Quality Improvement Process, James F. Riley & Joseph M. Juran, 1999 (excerpted from Juran's Quality Handbook), McGraw-Hill, New York, NY. You will find the section entitled "The Remedial Journey" particularly helpful in providing tools and guidance in assessing root causes related to apparent worker errors. Besides these mechanics, this book mainly guides you in the processes and practices needed to establish an effective quality improvement program.

Index

5

50/50 rule, 35

7

7 x 7 rule, 75

9

90/10 rule, 27, 28, 90

A

acceptance, 11
action title, 73
actual cost of the work performed (ACWP), 33
allocations, 48
ambiguity, 7
Augustine's Laws, 33, 115
authorization, 5
automated test, 16, 110

B

Barbie® doll, 18
baseline, 25
beneficial use, 55
bids, stillborn, 47
black box, 7
black magic, 9, 116
boss duality, 103
bottoms-up estimates, 43
bounded logs, 58
bounds, vs. tolerances, 10
branching, 53
break it, 17
breakeven calculations, 84
brute force redundancy, 51
budgeted cost of the work performed (BCWP), 33
budgeted cost of the work scheduled (BCWS), 33
bug (not better), 84
bulletproof branch, 53
bullets, 74

C

changes clause, 19
Chicken Little, 78
clickable mockups, 64
closing out, 56
CND (could not duplicate), 18, 80
collocation, 105
company practices, 15
completed staff work, 13
configurability, 57
configuration application, 58
continuous improvement, 79, 117
Cost Performance Indicator (CPI), 33
cost variance (CV), 38
cost versus price, 48
crime of management, 23
Critical Chain, Goldratt's, 116
Critical Design Review (CDR), 14
critical path, 28
critical task, 25
Crystal Reports®, 110
customer-definable descriptive field, 69
customers, 55

D

date constraint, 28
day zero, 52
deadline, 25
declaring victory, 52, 56, 83
Defense Advanced Research Projects Agency (DARPA), 50
design requirements documents, 7
design reviews, 12
destructive actions, 68
Dreamweaver®, 64, 109

E

earned value, 32
emulators, 16
enhancements, 84
Enterprise Resource Planning (ERP), 107

estimating factors, 30
e-Test Suite®, 110
Ethernet, 63
exception conditions, 16, 18, 21, 52, 59, 109, 110

F

fail early, 11
Failure Mode and Effects Analysis, 15
fault tolerance, 50
Fault Tree Analysis, 15
FDA, 14
feature creep, 7, 20, 43, 44, 55, 64
features branch, 53
federated architectures, 51
finger pointing, 17
finish-to-finish, 25
finish-to-start, 25
First, Break All the Rules, 115
folklore, 15, 54
Formula One®, 109
full in-boxes, 77
Functional Configuration Audit (FCA), 14
functional decomposition, 8, 116
functional requirements, 7

G

Gallup, 115
gestation periods, 43
GMT/UMT, 60, 70
god processes, 52
granularity, varying, 25
Graphical User Interface (GUI) design, 63

H

hammock, 27
hardware maintenance, 18
high-potential staff, 97
horse charts, 73
house of cards, 48

I

IDEF0 (Integrated Definition of Function Modeling), 115
-illities, 116
incentive criteria, 99
incremental pricing, 49
incumbents, 21, 48
interfaces, 9
ISO-9000, 2, 57

J

JProbe®, 110
Juran, 79, 117

K

key staff, 96
known issues, 86

L

labor rates
  average, 42
  budgeted, 42
leaving money on the table, 47
levels of effort (LOE), 28
load sharing, 51
look and feel, 64

M

management reserve, 44
mandatory fields, 68
matrix organizational structure, 103
medians, 90
mid-term improvements, 101
milestone, 25

N

Noah's Principle, 32, 43
NTF (no trouble found), 18, 80

O

omniscient, 17
open source, 108
Oracle®, 109
other direct costs (ODC), 28
overheads, 48
overrun, 19, 23, 33, 43, 44, 47, 65, 115

P

Parametric 3-D Computer-Aided-Design (CAD, 109
Pareto analysis, 90
penetration, 97
performance variance, 42
post-install, 93
Preliminary Design Review (PDR), 13
presentations, 73
Primavera®, 23
Product Design and Development, 115
product specifications, 10
Production Configuration Audit (PCA), 14
Project Management: a Systems Approach, 116
Project®, Microsoft, 23
proposal plans, 29
Purify®, 110

Q

qualification, 9
quality metrics, 88
Quantify®, 110

R

R&R (remove and replace), 18
rate variance, 42
religious preference, 63
resource
  bound, 26
  leveling, 26
  link, 26
reuse, 10
risk
  analysis, 15
  mitigation, 15, 45
rolling wave, 26
RTOK (retest OK), 18, 80

S

Schedule Performance Indicator (SPI), 33
schedule variance (SV), 38
severity, 81
should cost, 32
similarity, 15
Simplified English, 109
simultaneous tasks, 26
software design guidelines, 57
software maintenance, 18
solution, not a requirement, 7
specification
  functional, 7
  product, 10
  top-level, 13
split, 25
SQL Server®, 109
staff ranking, 95
start-to-start, 25
Statistical Total at Completion (STAC), 38
store-and-forward, 51
student syndrome, 27, 43
subordinate prevails, 19
substantial completion, 55
SysML (systems engineering modeling language), 115
System Design Review (SDR), 13
Systems Engineering and Analysis, 116

T

TBD, 60
technical analyses, 15
test, 16
The Art of Systems Architecting, 116
The Engineering Design of Systems, 115
The Quality Improvement Process, 117
The Remedial Journey, 117
tolerances, vs. bounds, 10
traceability matrices, 14
tracking Gantt chart, 23

U

ultimatums, 55
Unified Modeling Language 2 (UML2), 115
unique error code ID, 59
unk-unks, 45
urgency, 85
used hardware, 19

V

validation, 9
verification, 11
vertical waterfalls, 27

W

white box, 7
will cost, 32
WIP (work-in-progress), 91

About the Author

James T. Karam, Jr. obtained his B.S. in Mechanical Engineering from the University of Arkansas in 1964 and his M.S. in Aerospace Engineering from the Air Force Institute of Technology (AFIT) at Wright-Patterson AFB, OH in 1966. He started his professional career as an R&D officer in the U.S. Air Force, serving for 14 years. His first assignment was as a Development Engineer at the Air Force Plant Representatives Office at Lockheed Missiles and Space Company developing military satellite systems. After obtaining his PhD from Purdue University in 1972 specializing in automatic control, he returned to AFIT, becoming an Associate Professor. He then became a Program Manager at the Defense Advanced Research Projects Agency (DARPA) where he conceived and executed three major advanced cruise missile technology thrusts.

In 1978, Jim joined General Dynamics Convair Division, soon becoming their Director of Systems Engineering; then Program Director for a version of the Tomahawk Cruise Missile; and then the Director of the All-Up-Round Systems Engineering and Integration Agent, providing technical direction for the design baselines of all 44 contractors of all Tomahawk variants.

In 1984, Jim joined Philips Medical Systems, Inc. as their Director of Engineering developing advanced image processing products for Digital Subtraction Angiography and Computed Radiography. Returning to aerospace in 1987, Jim became VP, San Diego Operations for the Advanced Systems Division (ASD) of United Technologies Corporation (UTC). Among other products, ASD developed the payloads for Northrop's Tri-Service Standoff Attack Missile (TSSAM).

When peace broke out in the nineties, Jim rejoined the commercial world as VP, Systems Engineering of Sony Corporation of America's Business & Professional Products Group where his team developed DirecTV's Broadcast Control Subsystem and non-linear, disk-based video servers. He then became VP of Operations for Lunar Corporation where he had cradle-to-grave responsibility for all product-related activities of this bone densitometry market leader.

Jim then gambled as VP, Operations with Cybersensor, Inc, a dot-com startup providing remote monitoring of high-value assets such as pipeline compressors. Rejoining the real world, Jim ended his employed career as the Sr. VP, Engineering and Program Management for Cubic Transportations Systems, refreshing the entire product line of the

dominant supplier of automated fare collection systems for transit agencies worldwide.

While academically an ME, most would guess that Jim was an EE. Jim just stayed a student and let his staff teach him. Lately, he has most enjoyed returning the favor as a mentor to those of his staff who are becoming the new leaders. This book is just another step in that direction.

Since 2005, Jim has semi-retired to Weeki Wachee, FL with his lovely wife Alicia. He volunteers with the Service Corps of Retired Executives (SCORE) helping entrepreneurs to start up and run small businesses. More details, both professional and personal, may be found at his consulting website, www.karam.com.