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

Table of Tables

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.

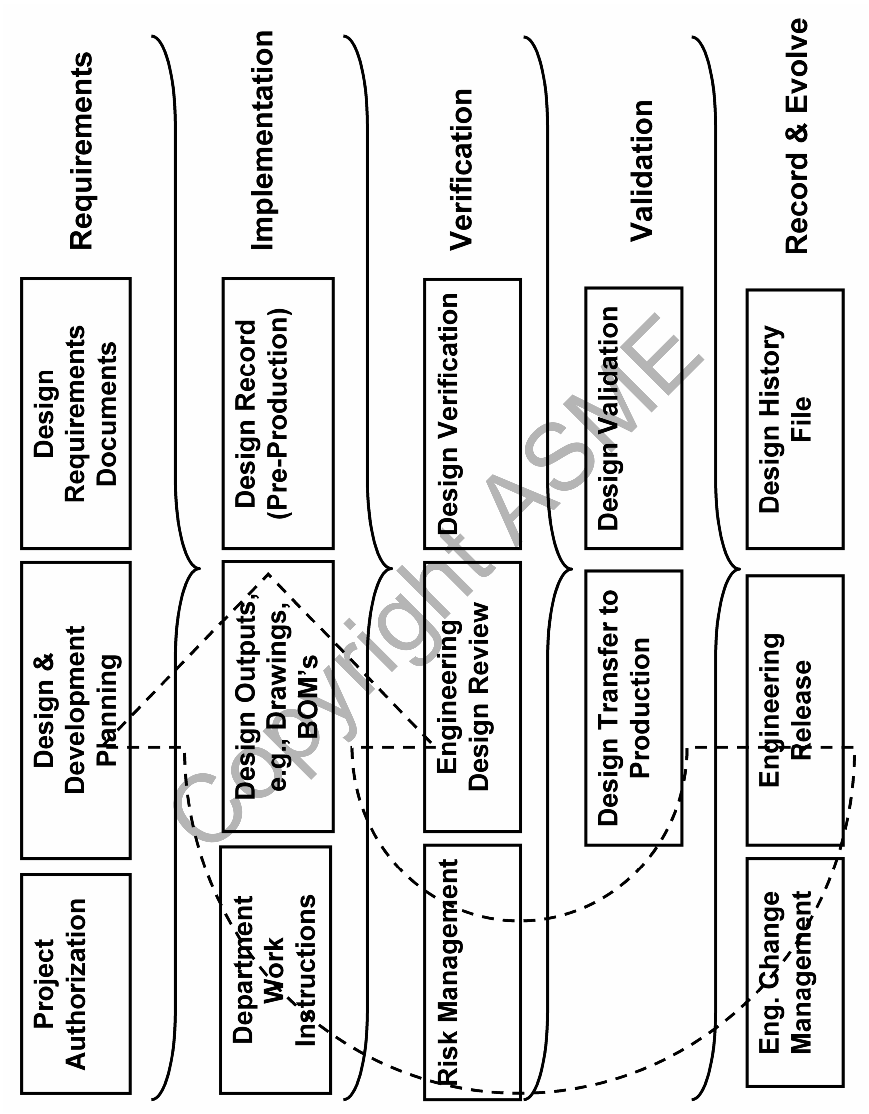
Closing remarks refresh our key advice. Candidates for additional reading conclude the text.

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.



Requirements
Project Authorization
Design & Development Planning
Design Requirements Documents
Implementation
Department Work Instructions
Design Outputs, e.g., Drawings, BOM's
Design Record (Pre-Production)
Verification
Risk Management
Engineering Design Review
Design Verification
Validation
Design Transfer to Production
Design Validation
Record & Evolve
Eng. Change Management
Engineering Release
Design History File

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.

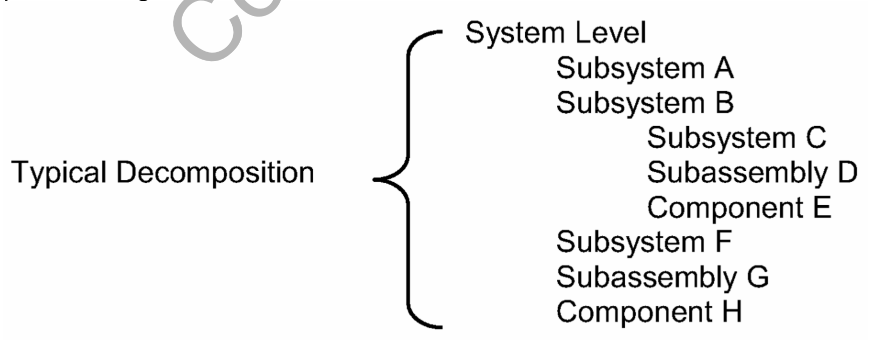


Figure 1.2 Decomposition Hierarchy

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**.