



Don't Just Kick the Tires

Reuel Alder
Publisher



When I was the manager of a software development team, I worked for some difficult customers. Our processes were developing but in fairly good shape; however, our customers were difficult to service because they lacked discipline in developing or managing their requirements. They did not know what they needed or the priority of each requirement. Add to this that our software product served several user communities—all of whom similarly lacked a requirements management process—and you can imagine our distress. This forced us to assemble all user requests and assign our own priority. We would then work from the top down in hopes of completing the work before the dead-

line. This chaotic requirements “process” inevitably resulted in schedule and cost overruns.

I would like to say that this is an anomaly in defense software development, but it is not. Requirements generation for software-intensive systems is a widespread problem area. Requirements are at the beginning of the lifecycle product development and are therefore the most expensive to change or fix.

I wonder if this problem has a cultural basis. The American public has been trained by a sales-intensive environment to not analyze needs before making a major purchase. In a sense, we kick the tires, examine the pretty red paint, listen to a lot of evasive monthly payment jargon, then buy the car.

In making a sale, salespeople are trained to get customers emotionally

committed to a product before filling in the rational thinking. This technique works because few buyers enter the store knowing what they want. They depend on the salesperson to essentially tell them what they want. They want to be persuaded to make a purchase. If only they did their homework ahead of time, they could make a purchase more suited to their needs, not to mention their budget.

This cultural bias appears to feed the indecision and impulsive requirements development in today's defense acquisition community. With the need to conserve resources and stay battle ready, the Department of Defense needs the discipline of a requirements management process for buyers as well as for developers. Mature software development processes are highly effective, but they cannot compensate for a lack of requirements management by the user. ♦



Letters to the Editor

Experienced Developers as Mentors for Management?

Alan Reagan's comments (Letter to the Editor, September 1998) prompt a reply. First, for the past nine years, I have been a government contractor, involved in both the nuts and the bolts of software development and in software process improvement consulting. Second, I don't think of myself as “the enemy,” but believe that the roots of the problem lie elsewhere.

Software developers generally are engineers interested in building a working system that fills the customer's needs. Their goals (regardless of their CMM [Capability Maturity Model] or their ISO [International Organization for Standardization] status) are related to the so-called *ilities*: quality, reliability, usability, etc. Program managers, though, are often more concerned with budget and schedule issues: milestone achievement, acquisition reviews, etc. This does not imply that one set of priorities is wrong and the other is right or even that one has inherently more merit. It merely means that their priorities are different.

My point: While project managers may understand the factors and dynamics affecting their priorities, they often have little background in the technical side of software development. Rather than trying to make project managers into software engineers, we could rely on an experienced developer to act as guide (mentor?) through the swamp of software development.

And finally, if, as Reagan asserts, many of the independent validation and verification (IV&V) team members he has dealt with have fewer than five years experience, perhaps the request

for proposal for IV&V support should be rewritten to require proven development experience.

Joe Saur
SOCOM, Keane FedSysDiv
Tampa, Fla.

Use It or Loose It

No one disputes the problems associated with software. Over the years, many solutions have been developed but applied sparsely. We now have the Software Engineering Institute, the Software Technology Support Center, and other organizations that provide outstanding packages. We also have organizations such as the Institute of Electrical and Electronics Engineers Computer Society, the Association for Computing Machinery, and the Institute for Certification of Computing Professionals that provide leadership in moving software engineering into a profession. Some of these efforts are directed toward the Department of Defense (DoD) and related industries and others toward process control or imbedded systems. I hope the developments will not be lost on the non-DoD business organizations, because software engineering is software engineering regardless of where it is applied. In fact, we may need these processes more here than in the areas for which they were developed. I also hope that the integration efforts, e.g., hardware, software, and communication, will continue and become the norm. We need total systems and not the finger pointing we seem to have now.

Ed Mechler
Project Controls, Inc.
Penn Hills, Pa.



Experiences in the Adoption of Requirements Engineering Technologies

Jim Van Buren and David A. Cook
Software Technology Support Center

It has been known since as early as the 1950s that addressing requirements issues improves the chance of systems development success. In fact, whole software development standards (such as MIL-STD-2167, MIL-STD-2167A, MIL-STD-498, and IEEE/EIA 12207) were designed to enforce this behavior for software-intensive systems. Relatively recently (sidebar) a new field of study, requirements engineering, has begun to systematically and scientifically address barriers to the successful use of requirements in systems development. Since 1992, the Software Technology Support Center (STSC) has been helping organizations adopt new technologies. This article defines requirements engineering (RE) from the viewpoint of technology adoption, discusses which RE technologies are most critical to mission success and why and which are most difficult to adopt, and outlines successful adoption approaches.

Requirements engineering as a field addresses requirements issues in a holistic manner. Understanding the interrelationships between the various requirements activities and how they support each other is as important as understanding the technical details of any one of the individual activities. Contrast this to the 1970s and 1980s, when the software engineering community focused essentially only on requirements analysis, or the early 1990s, when requirements management was the fad. This holistic approach is the great requirements insight of the 1990s. The understanding of requirements activities from this view helps engineers build and follow lifecycle models that account for their project's business goals, the attributes of their requirements, and the strengths and weaknesses of their requirements technologies.

Challenging Old Assumptions

One must challenge the assumption that a requirements specification is equivalent to a development contract. For some projects, blind adherence to this assumption makes project success much more difficult. Once this assumption is challenged on a project basis, this drives what requirements are needed and for what they are needed. Once it is recognized that the project's goals (such as time to market or long-term maintainability) and attributes (such as requirements volatility) should drive its lifecycle

development and requirements process, it is fairly straightforward to build or tailor, with appropriate emphasis on the requirements specification, a requirements engineering process. This is overwhelmingly the number one requirements engineering technology adoption lesson learned (and the second great requirements insight of the 1990s).

Today's requirements research [1] is focused on issues that come to light when the "requirements are equivalent to development contract" idea is discarded. Research concerns include

- How are requirements prioritized? Requirements prioritization becomes critical when fixed develop-

ment dates or fast development (time-to-market) considerations drive the development rather than the need to meet all requirements.

- How does a project cope with incomplete requirements?
- How can requirements engineering support the commercial development paradigm (where feature sets, product sizing, and market window are the focus rather than functional requirements)?
- What is the interdependence of requirements and design (for example, a strong interdependence is necessary when building commercial-off-the-shelf-based systems)?

Requirements Engineering Background

Papers as early as 1956 have discussed the importance of requirements definition in software development, but it was not until 1976, at the International Conference on Software Engineering, that requirements engineering was recognized as a subdiscipline of software engineering. In fact, at the 1968-69 NATO software engineering workshops (where the term "software engineering" was first coined), software engineering was explicitly decomposed into only design, code, and test activities [2].

The first time we noticed the term "requirements engineering" was in conjunction with the 1993 International Symposium on Requirements Engineering (RE '93), the first conference devoted entirely to requirements topics. Before 1993, it seemed that requirements research was stovepiped into areas such as "requirements management" or "requirements analysis." Since 1993, the term "requirements engineering" and its accompanying thesis of holistically addressing all the requirements activities has become widespread. In addition to the RE series of conferences [3], the IEEE's International Conference on Requirements Engineering [4] meets every other year. The IEEE has also published a seminal collection of RE papers [5].

Bad Requirements Process Leads to Development Failure

In June 1994, the Federal Aviation Administration (FAA) canceled its 10-year effort to modernize the nation's air control system. About \$1.3 billion was written off [6]. In 10 years, the requirements elicitation phase had never come to closure. A requirements specification with a height that could be measured in yards was produced, but it was fundamentally incomplete. This is the most expensive development failure due to a requirements failure of which we are aware. One can argue that there were many other problems with the program, but it was during the requirements process that the program failed.

Tom DeMarco produced a brilliant analysis of what went wrong [7]. He knew he was on the right track when he could not find a keyboard mentioned in the specification. This led to the observation that the customer, the FAA, was unable to specify if the system was to be centralized (Washington office's desire) or decentralized (controllers' and regional operating centers' desire). DeMarco has since lectured extensively that internal customer conflicts like this must be resolved before a specification can be completed and that conflict resolution is an overlooked arrow in a requirements engineer's quiver.

Requirements elicitation can help identify internal customer conflict. But the customer—not the requirements engineer—must resolve conflicts or the system being built is doomed to fail.

Elements of Requirements Engineering

Even if one breaks the link between requirements specification and development contract, this does not alter the need to perform requirements activities. They are merely performed with a different flavoring of objectives. Individual requirements technologies are still best viewed from the perspective of the requirements objectives they address. The caveat is that they must support the chosen overall requirements engineering process.

We divide requirements engineering into the categories of elicitation, analysis, management, validation and verification, and documentation. This taxonomy helps one understand both the requirements problems and the requirements technology adoption issues that face our clients. Like the biological taxonomy, ours is intended to be a living entity, subject to slow change. As we have learned more about RE and as RE matures as a field, our taxonomy has, in fact, changed.

Requirements Elicitation

This field addresses issues that revolve around getting customers to state exactly what their requirements are. Pro-

grams large and small still fail to reach closure on this step, in spite of adequate effort. Other programs reach closure but do not capture all the requirements. This is perhaps the area of requirements engineering with the highest incidence of malpractice. Software engineers all agree that requirements elicitation is important, yet they uniformly spend too little time performing it.

Requirements elicitation is the only requirements engineering field without a definitive technical solution, yet good informal solutions exist. The lack of technical solutions is expected because the elicitation problem is human in nature. The issue is that customers often cannot state what the requirements are because they either do not know what they want, are not ready to fully define what they want, or are unable, due to outside influences, to decide what they want. The FAA sidebar above outlines the classic example of this last behavior.

The biggest elicitation failings (missed requirements and inability to state requirements) manifest themselves as omissions or inconsistencies, which may not become apparent until requirements analysis or systems acceptance testing or even systems use (see Ariane

Flight 501 sidebar). Customers must understand that incomplete, inconsistent, or ambiguous requirements, at best, cost a lot of money. At worst, they guarantee failure of the entire system. Spending additional time “fleshing out” requirements always results in an overall cost saving.

Elicitation Mechanisms

During requirements elicitation, one must derive the system requirements from domain experts—people familiar with their domain but not necessarily with building software systems. The system developers must therefore be conversant in the terms and limitations of the domain, since the domain experts are probably not conversant in the terms and limitations of software engineering. To help overcome this potential communication barrier, elicitation mechanisms are needed to add formality to what could otherwise be a “seat of the pants” methodology.

Informal elicitation mechanisms (such as prototyping, Joint Application Development, Quality Function Deployment, Planguage [8], or good old structured brainstorming) address motivated and able customers who do not know how to express their needs. We conjecture that the root cause of communication barriers is that the term “requirements” is used differently by different parties. The “requirement” that is the output of the elicitation process has a specific meaning to a software or systems engineer. It is a real need of the customer, it is testable, and it may be prioritized. The requirement can also be validated by the customer.

To an uninformed customer, a requirement is often simply only a statement of need. Elicitation mechanisms help overcome this communications barrier by helping the customer understand and state needs in an objective manner. There are interesting side effects of these mechanisms. Customers develop an ownership in the outcome of the development effort and better understand the problem that is being solved. The customer's needs and desires (nice-to-haves) are explicitly separated. Developers establish a working relationship

with the customer and have an understanding of what the problem is and where trade-offs can be made.

Despite the hype, formal methods are not a complete solution. They are not effective in involving the customer; however, they are effective in gaining greater understanding of constrained parts of the problem domain. They can assist an elicitation approach based on informal techniques but cannot stand on their own.

The adoption of elicitation technologies first requires the recognition that elicitation can be a problem and recognition that the term “requirement” has different meanings to different people. Once this occurs, the straightforward plan is to obtain training for the organization’s elicitors in a variety of elicitation techniques and interpersonal skills. Practicing elicitation on real projects involves the use of a variety of elicitation and validation techniques that together increase the probability that the customer has properly stated its real requirements and that the development organization understands them. Organizations need to recognize that elicitation is a skill learned through practice. Practitioners generally are proficient after training but not expert until after several projects.

Requirements Analysis

Requirements analysis serves two primary purposes:

- It is used to make qualitative judgments, i.e., consistency, feasibility, about the systems requirements.
- It is a technical step in most systems development lifecycles in which an extremely high-level design of the system is completed. This high-level design consists of decomposing the system into components and specifying the component interfaces. The critical output for most software development requirements analysis activities is the interface specification for the decomposed components.

There are a number of well-understood technical approaches to analysis, i.e., OMT, Schlaer Mellor, Structured Analysis, and UML. Most have good commercial tool support. Organizations

do not have difficulty in finding experts, in developing (through training, mentoring, and experience) experts, or in finding tools to support their analysis efforts. Instead, technology problems arise from both over- and underanalysis. For example, well-funded programs tend to overanalyze. They analyze everything that can be analyzed without first determining what should be analyzed. Often, detailed design occurs during this analysis phase. Programs with cost constraints suffer the opposite fate. They tend to underanalyze, probably as a cost savings measure.

Plenty of methods and tools support analysis, and there are both tool-related and training-related technology adoption issues. Tool-related problems occur when there are inconsistencies between a tool’s implied development process and the organization’s standard development process. Tool vendors have come a long way this decade in addressing this issue, but it has not gone away. Tools become “shelfware” if they impose their process over the organization’s process, even if the organization’s process is undefined and ad hoc.

To adopt requirements analysis successfully, pilot the analysis methods manually, then identify what steps need to be automated, then make a tool selection, then tailor the tools use, then use the tool. Over time, the organization’s process can gradually evolve.

We have observed that standard training plans are often inadequate to address new analysis methods. The detailed training of how to apply a method or a tool is necessary but not sufficient. Education may also be needed if the new method is radically different—as object-oriented differs from structured—from established methods. In addition, mentoring on the first pilot project is necessary for all but a few individuals. When adopting new analysis methods, always plan for education, training, and mentoring.

Requirements Management

Requirements management addresses aspects of controlling requirements entities. Requirements change during and after development. The accepted require-

ments volatility metric is 1 percent of requirements per month [10]. If it is much less, one should ask oneself if the system will be desirable to its intended audience. If it is much more than 2 percent a month, development chaos is all but assured.

Requirements management is the requirements issue that most impacts military software projects. In a 1993 report, Capers Jones found that 70 percent of all military software projects are at programmatic risk because of requirements volatility [10]. The root causes (discussed below) of this volatility have not disappeared, so we believe his finding is still true.

With one exception, requirements volatility is uncontrollable. It will occur as a byproduct of building a useful product, and one’s development processes needs to account for it. Software systems do not exist in isolation. As a new system is built, the system will affect its environment, which will in turn change its environmental requirements. This is inevitable and indicates one is building the right system.

Most development efforts, and all development efforts for which requirements management is important, take time, sometimes on the order of years. Over these periods, underlying technologies, user expectations, and even laws change, to name just a few of the many possible external interfaces. If one is not getting requirements change requests on large projects, one needs to ask why. The one source of volatility that can be controlled is the quality of the initial requirements specification. Good elicitation techniques can limit rework as a cause of volatility.

Requirements Management Technologies

Fortunately, the technologies needed to address the requirements volatility issue are relatively simple. The organization needs a defined interface mechanism with its customer by which requirements are changed, a mechanism (usually a configuration management system or an RM tool) capable of defining the current requirements baseline, and a development approach, i.e., incremen-

tal lifecycle, that supports the anticipated requirements volatility. The primary adoption issues that must be addressed are

- Building senior management recognition that this defined mechanism is necessary.
- Having the discipline to always follow this change mechanism.

The old adage that “the customer is always right” is not an absolute. When customers ask for a requirements change, they must be told the impacts of that change, usually in terms of other prior commitments, and then be allowed to make the final decision. The software development organization must never unilaterally add a requirement.

Tools and Adoption Technologies
Effective management usually implies both exerting control over and knowing the status of requirements. There is a whole class of RM tools (such as Requisite Pro, DOORS, RTM, and Caliber-RM) that automate the tracking of requirements across lifecycle phases. They also support many of the requirements baselining and requirements documentation issues. They are particularly useful for programs that wish to follow a requirements-centric development approach. Technology adoption issues associated with these tools are the standard issues for tools. Understand what the tool will be used for, find a tool that meets those needs, then build

a plan for adopting the tool, being careful that the tool not be used for purposes beyond identified needs.

The Software Capability Maturity Model (SW-CMM) (and to some extent the Systems Engineering and the Software Acquisition CMMs) includes requirements management as a Level 2 key process area (KPA). Within this context, the requirements management KPA only applies to managing requirements change. Other requirements management tracking and status activities could apply as part of the Level 3 SW-CMM KPA, Software Product Engineering, depending on the organization’s development approach. When implementing a CMM-based technology change pro-

Ariane Flight 501

Bad Requirements Lead to Systems Failure

On June 4, 1996, the maiden flight of Europe’s Ariane 5 rocket ended in catastrophic failure with a complete destruction of the rocket and its payload [9]. The cause was a software error, perhaps the most expensive software error on record. The root cause of this error was a breakdown in the requirements process—not in the software design or coding processes—that was not caught by the developmental verification and validation process. Within the requirements process, there were problems with elicitation, analysis, and verification and validation.

At liftoff, plus 30 seconds, an operand exception was generated in the Inertial Reference System (SRI) computer during a conversion of a 64-bit floating entity into a 16-bit signed integer. This caused the SRI to crash and output a diagnostic bit pattern. The redundant backup SRI had also crashed 72 milliseconds earlier for the same reason. Ariane 5’s on-board computer interpreted the SRI’s diagnostic bit pattern as valid commands and ordered full nozzle deflections of both the solid boosters and the main engines. The rocket was then destined to break up.

The official Inquiry Board found that the primary causes of the crash were “... specification and design errors in the software ...” and “... reviews and tests ... did not include adequate analysis and testing.” The software requirements were incomplete and neither the requirements analysis activities or the requirements verification and validation process discovered this omission. They also found fault in exception handling requirements, which basically were to log the error and terminate. This arose from a faulty belief that random hardware failures were the only reason for an exception and that systematic software errors would never occur (systems analysis failure).

The software and software requirements were essentially reused from Ariane 4. An explicit decision had been made to not include the Ariane 5 normal liftoff trajectory as part of the software requirements (requirement elicitation failure). When computing the alignment horizontal bias for the Ariane 5 trajectory, the operand exception will always occur at about liftoff, plus 30 seconds. The operand exception will never occur for the Ariane 4 trajectory in the first 43 seconds of flight. Had the trajectory been included as a requirement, the official Inquiry Board believed that the developer’s analysis and testing process would have observed this exception.

Exception handling had been turned off because of a processor performance requirement (maximum 80 percent processor utilization). The Ariane 4 analysis indicated that horizontal bias would remain within the range of a 16-bit signed integer with the Ariane 4 trajectory. This justification analysis was not easily available to the Ariane 5 development team (requirements process failure).

The reuse of the Ariane 4 software requirements was also flawed. Ariane 4’s requirement to continue computing alignment (which includes the horizontal bias) for 50 seconds after entering flight mode (liftoff is seven seconds into flight mode) to support a late hold is not needed for Ariane 5. The original requirement may even be a bit flawed, because the alignment calculation is physically meaningless after liftoff (systems analysis failure).

The Inquiry Board also took issue with the verification and validation processes. Its primary finding was that these processes did not identify the defect and were thus a “contributory factor in the failure.” The explanation given for not testing or analyzing the Ariane 5 trajectory was that it was not a part of the requirements specification (requirements verification and validation failure).

gram, remember that managing the customer interface comes first. A second step might be tracking the status of requirements across the development lifecycle and using that information to manage the development.

Requirements Validation and Verification

The requirements verification and validation (V&V) portion of RE addresses how quality is built into the RE process. Validation (“Are we building the right system?”) addresses the issue of building the system the customer wants. This quality step should identify missing and extra requirements. Validation activities always occur as part of system acceptance testing and also typically, but not always, as part of the elicitation process. There are several orders of magnitude cost difference in requirements misunderstandings that are identified as part of elicitation, before development resources have been expended, vs. those that are found during system acceptance testing. This points out the critical need for the elicitation process to include validation.

Customer and domain expert input are necessary for validation. In fact, attempting to validate a system without customer input is equivalent, in the words of one of our customers, to designing a “self-licking ice cream cone.” The necessity of user input is another reason formal methods are insufficient for building systems—the customer usually cannot understand and does not want to learn how to validate using formal mechanisms. Mechanisms that a customer can easily understand (and hence easily validate) are almost always based on clear language and easily understandable pictures—input the customer can already comprehend. If the customer has to learn a new notation or method to validate a system, a new quality issue is introduced to the validation process: lack of a clear understanding of the method. The focus needs to be on the solution, not on the method.

Verification (“Did we build the system right?”) addresses the issue of meeting all the requirements. Typically, the verification method and sometimes the

verification level is included in the requirements specification. Verification methods include demonstration (observable functional requirements), analysis (collected and processed data), simulation (use of a special tool or environment to simulate the real world), and inspection (examination of source code and documentation). Verification levels depend on the intended development environment. They specify the development lifecycle stage at which the verification will be performed, i.e., unit test, integration test, system installation, or flight test.

From a technology adoption perspective, requirements V&V is a question of designing a development lifecycle that meets the needs of the product. Emphasis needs to be placed on validation in the *elicitation* phase. It should be considered software engineering malpractice if requirements V&V is not also included during design and coding phases. Validation and verification must be performed after the system has been built.

Requirements Documentation

There are a number of potential standards for structuring requirements specifications. American National Standards Institute/Institute of Electrical and Electronics Engineers-STD 830-1993 specifically addresses requirements specifications. The lifecycle standards Electronic Industries Association (EIA)/IEEE 12207 and the withdrawn standards MIL-STD-2167A and MIL-STD-498 specify another similar format. The basic contents of all these are the same: They all include an overall description, external interfaces, functional requirements, performance requirements, design constraints, and quality attributes.

Another school of thought posits that there should be a bare minimum of requirements documentation. A concept of operation document or a users manual are all that are needed for a requirements statement. This makes sense for applications where time to market is more important than long-term maintainability.

Adopting requirements documentation technology is fairly straightforward. One should choose a standard that fits the lifecycle requirements of the system, tailor that standard to fit the system’s specific requirements, then apply it. If one intends to use an RM or a requirements analysis tool to automate a portion of the document generation, one should pilot the documentation process. Experience has always shown this to be much more difficult than originally envisioned.

Technology Adoption

A technology adoption process will increase the probability of a successful technology change. At the STSC, our technology adoption process is based on the IDEAL Model [11] (Figure 1). We also use two other important adoption principles:

- Small improvement steps.
- Address needs at all levels of the organization.

We discuss the IDEAL Model here not necessarily because it is the best technology adoption model (although we believe it is), but to demonstrate the importance of picking an adoption model, basing one’s adoption process on that model, and improving one’s model and process over time. The following is a synopsis of the five steps in the IDEAL Model:

- **Initiate** – Obtain and maintain sponsorship.
- **Diagnose** – Assess current practice.
- **Establish** – Produce plan to address shortcomings.
- **Act** – Pilot and use the technology(ies).
- **Learn** – Collect lessons learned, next steps (such as rollout), cycle back to diagnose.

Our RE field experience indicates that organizations planning RE technology purchases or process changes generally do not follow a process or a model for technology adoption. If they are not planning a purchase, they usually do not even realize that what they are doing is technology adoption. This differs from organizations interested in process improvement, which tend to

produce detailed technology adoption plans based on models like IDEAL.

When a tool or a method is purchased, the vendor is consulted regarding its specific adoption recommendations. With one exception, an SW-CMM Level 5 organization, we have not seen any formal mechanisms that use lessons learned from prior technology adoptions. Our advice is that vendors' recommendations become the functional requirements for the adoption plan and that the plan be driven by the organization's past adoption experiences.

Small Steps

To build adoption plans, two adoption principles must be adhered to. First is the principle of small steps. Many small process improvement steps have a greater chance of success than one giant process improvement leap.

To build a requirements-centric development process, one cannot jump right to the final state. Instead, the first step might be to get all one's requirements changes under control. The second step would be to pilot an RM tool that reports the development status of every requirement and produces requirements documents. The final step would be to use requirements status

Technology	Organizational Level		
	Individual	Project	Organization
Elicitation	<ul style="list-style-type: none"> • Training • Mentoring 	<ul style="list-style-type: none"> • Technique Selection • Tailoring of V&V strategy 	<ul style="list-style-type: none"> • Persistent Training Program
Analysis	<ul style="list-style-type: none"> • Education • Training • Mentoring 	<ul style="list-style-type: none"> • Technique Selection • Tailoring of V&V strategy 	<ul style="list-style-type: none"> • Persistent Training Program
Management (control)	<ul style="list-style-type: none"> • Discipline to follow the process 	<ul style="list-style-type: none"> • Tailored Process • Tool Adoption (if necessary) 	<ul style="list-style-type: none"> • Policy • Policy enforcement by organization's executives
Management (status)	<ul style="list-style-type: none"> • Training 	<ul style="list-style-type: none"> • Tool Piloting 	<ul style="list-style-type: none"> • Organizational Standards
Validation and Verification	<ul style="list-style-type: none"> • Training (reviews, inspections, test tools) 	<ul style="list-style-type: none"> • Tailored Approach 	<ul style="list-style-type: none"> • Enforcement
Documentation	<ul style="list-style-type: none"> • Training 	<ul style="list-style-type: none"> • Tailored Approach 	<ul style="list-style-type: none"> • Organizational Standards

Table 1. Requirements engineering technology adoption issues.

information to manage one's development efforts.

Address All Organizational Levels

The second adoption principle is that plans must address all levels of the organization: the individual, the project, and the organization in its entirety. For example, an adoption plan to meet the objectives of the RM KPA of the SW-CMM would involve all three levels in different ways.

Senior management, representing the organization, would need background RM training (indoctrination) on why controlling requirements is an important issue. They will have to issue and enforce an organizational policy. More important, they may have to stand up to the organization's customers and tell them that, unlike the old days, the customers cannot change or add requirements in an uncontrolled manner.

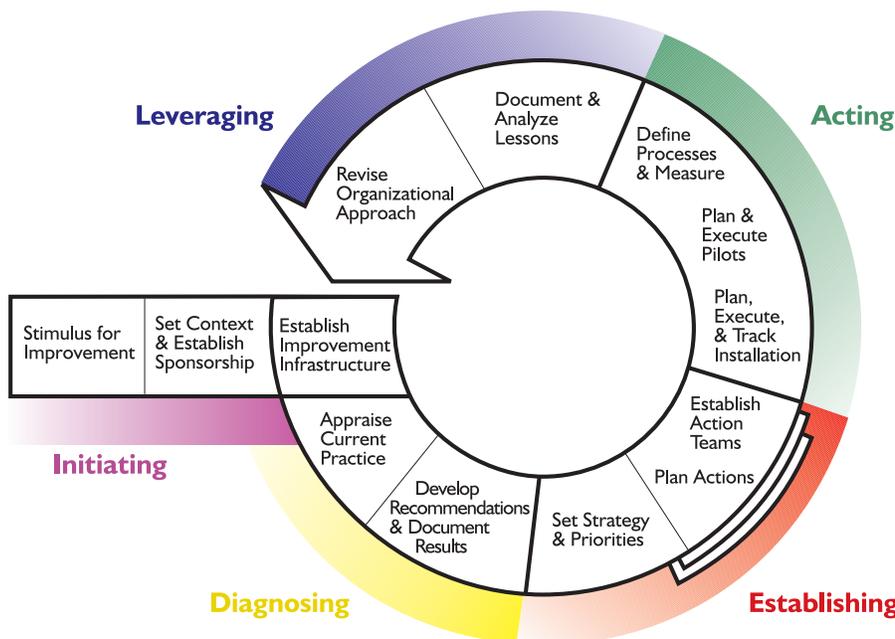
On the project level, a system, most likely tools and processes, will be needed to track requirements baselines. Ultimately, individuals must have the discipline to never allow requirements to creep into the system outside of standard channels.

In the prior example, the adoption emphasis needs to be placed at the organizational level. If that step succeeds, the others will generally follow. But the level of emphasis differs depending on the type of requirements technology. For example, emphasis should be placed on the individual adoption issues when elicitation technologies are being adopted. Elicitation is essentially an individual skill, bordering on art form.

Adoption Effort

Table 1 examines the technology adoption issues for each of the requirements technologies from the perspective of various organizational levels. Although

Figure 1. The IDEAL Model.



RE Technology	Effort Required to Adopt	Effort Required to Verify That Adoption Has Worked
Elicitation	Moderate for proficient application. Hard for expert application.	Hard.
Analysis	Hard.	Moderate.
Management (Control)	Easy (Getting commitment is sometimes hard.)	Easy.
Management (Tracking Status)	Moderate.	Easy.
Documentation	Easy. Moderate, if the documentation is to be automatically generated.	Easy.
Verification and Validation	Easy.	Moderate.

Table 2. *Relative difficulties of the various requirements technologies.*

all issues need to be addressed, those that are bold italicized are the issues critical to adoption success for each of the technology areas.

Finally, there is the question of how hard technologies are to adopt. Some technologies require a lot of effort to master. Analysis technologies are an example of this. The elicitation technologies require a medium amount of effort to become proficient but a lot to master. The other requirements technologies all require relatively less amount of effort to master.

Another view of the difficulty of technology adoption is how difficult it is to verify that the technology has been adopted. Elicitation is extremely hard, analysis is moderate, and the others are easy. Table 2, summarizes the relative difficulties of the various requirements technologies. Note that the table only captures relative differences between the various RE technologies and only addresses adoption issues; it does not address the relative difficulty of practicing each of the requirements activities.

We have observed that organizations usually succeed when adopting "easy" technologies, even without outside assistance. They usually fail when adopting "hard" technologies, unless supported by external consultants.

Summary

Requirements engineering is the systems development activity with the

highest return on investment payoff. The cost savings that result from finding errors during verification and validation of requirements can be as high as 200-to-1 [12].

However, the requirements task is inevitably always harder than it first appears. If one were to presuppose that the customers were motivated and able to specify accurate and complete requirements, that the requirements would never change, and that there were no cost or schedule constraints placed on a development, there would not be any requirements issues. Unfortunately, none of these presuppositions are true. RE is the technical field of study that attempts to address and balance these issues.

The requirements phase is the interface between a customer's needs and the technical development process. The skills needed to perform requirements activities are a marriage of the people skills necessary to interface with the customer and the technical skills needed to understand the development process. At their heart, requirements skills are human based. Tools and technologies can only support requirements activities. When evaluating and adopting new RE technologies, focus on those technologies and adoption issues that support the human requirements engineer. ♦

About the Authors



Jim Van Buren is on the technical staff of Charles Stark Draper Laboratory, which he joined in 1983, under contract to the STSC. He has supported the STSC and the STSC's customers since 1989 in requirements, design, object-oriented technologies, and other technologies relating to the development of software. He is an SEI-authorized Personal Software ProcessSM (PSP) instructor. He currently serves as Draper's technical program manager at the STSC.

Software Technology Support Center
7278 Fourth Street
Hill AFB, UT 84056
Voice: 801-777-7085
Fax: 801-777-8069
E-mail: vanburej@software.hill.af.mil



David Cook is a principal member of the technical staff, Charles Stark Draper Laboratory, currently working under contract to the STSC. He has over 25 years experience in software development and has lectured and published articles on software engineering, requirements engineering, Ada, and simulation. He has been an associate professor of computer science at the U.S. Air Force Academy, deputy department head of the software engineering department at the Air Force Institute of Technology, and chairman of the Ada Software Engineering Education and Training Team. He has a doctorate in computer science from Texas A&M University and is an SEI-authorized PSP instructor.

Software Technology Support Center
7278 Fourth Street
Hill AFB, UT 84056
Voice: 801-775-3055
Fax: 801-777-8069
E-mail: cookd@software.hill.af.mil

References

1. Siddiqi, Jawed and M. Chandra Shekaran, "Requirements Engineering: The Emerging Wisdom," *IEEE Software*, March 1996, pp. 15-19.
2. Boehm, Barry, *Proceedings of the 2nd International Conference on Requirements Engineering*, 1996, p. 255.

3. International Symposium on Requirements Engineering (RE) '92, RE '94, and RE '96.
4. International Conference on Requirements Engineering (ICRE), 1994, 1996, and 1998.
5. Thayer, R.H. and M. Dorfman, eds., *System and Software Requirements Engineering*, 2nd ed., IEEE Computer Society Press, Los Alamitos, Calif., 1996.
6. Barlas, Stephen, "FAA Shifts Focus to Sealed-Back DSR," *IEEE Software*, March 1996, p. 110.
7. DeMarco, Tom, International Conference on Requirements Engineering, Tutorial, March 1998.
8. Gilb, Tom, "Requirements-Driven Management: A Planning Language," *CROSSTALK*, Software Technology Support Center, Hill Air Force Base, Utah, June 1997, p. 18. Language description is available at <http://www.stsc.hill.af.mil/SWTesting/gilb.html>.
9. "Ariane 5, Flight 501 Failure," Report by the Inquiry Board, July 19, 1996, <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>.
10. Jones, Capers, *Assessment and Control of Software Risks*, Prentice-Hall, Englewood Cliffs, N.J., 1994.
11. Gremba, Jennifer and Chuck Myers, "The IDEAL Model: A Practical Guide for Improvement," *Bridge*, Software Engineering Institute, Issue 3, 1997. Also available at <http://www.sei.cmu.edu/ideal/ideal.bridge.html>.
12. Davis, A., *Software Requirements, Objects, Functions, and States*, Prentice-Hall, Englewood Cliffs, N.J., 1993.

Coming Events

Call for Papers: The International Conference on Practical Software Quality Techniques '99

Dates and Locations: June 7-10, 1999, San Antonio, Texas; Oct. 4-7, 1999, St. Paul, Minn.

Sponsor: The San Antonio Software Process Improvement Network

Featuring: Watts Humphrey and James Bach

Topics of Interest: Inspections, Reviews, and Walk-throughs, Testing, Software Process Assessment and Improvement, Quality Management Issues, Measurements and Benchmarking, ISO 9000 Certification, Configuration Management and Version Control, Change Tracking, Requirements Management, Year 2000 Process Quality Issues, and automated tools that deal with any of these areas. Abstracts that deal with other topics will also be considered. Presentations are one hour and 15 minutes.

Abstract due date: Jan. 15, 1999

Send all submissions (MS Word or RTF format) via E-mail to: Dr. Magdy S. Hanna

E-mail: mhanna@softdim.com

Internet: <http://www.softdim.com>.

NDSS '99 Symposium

Dates: Feb. 3-5, 1999

Location: San Diego, Calif.

Topics: This sixth annual Network and Distributed System Security Symposium brings together researchers, implementers, and users of network and distributed system security technologies to discuss today's important security issues and challenges. The symposium fosters the exchange of technical information and encourages the Internet community to deploy available security technologies and develop new solutions to unsolved problems.

Contact: Carla Rosenfeld

E-mail: carla@isoc.org

Internet: <http://www.isoc.org/ndss99>

WICSA1: First Working IFIP Conference on Software Architecture

Dates: Feb. 22-24, 1999

Location: San Antonio, Texas

Sponsor: International Federation for Information Processing (IFIP).

Topic: WICSA1 will provide a focused and dedicated forum for the international software architecture community to unify and coordinate their effort in advancing the state of practice and research. An important goal of this working conference is to facilitate information exchange between practicing software architects and software architecture researchers. This conference will serve as a kickoff event for a new IFIP Technical Committee 2 working group on software architecture and will shape the focus and tasks of the working group for the initial period.

Contact: Paul Clements

E-Mail: pclement@sei.cmu.edu

Internet: <http://www.bell-labs.com/usr/dep/prof/wicsa1>

Third Symposium on Operating Systems Design and Implementation (OSDI '99)

Dates: Feb. 22-25, 1999

Location: New Orleans, La.

Topic: Continuing in the tradition of the OSDI symposium, the third OSDI will continue to focus on practical issues related to modern operating systems. OSDI brings together professionals from academic and industrial backgrounds and has become the perfect forum for issues concerning the design and implementation of operating systems for modern computing platforms such as workstations, parallel architectures, mobile computers, and high-speed networks.

Internet: <http://www.usenix.org/events/osdi99>

An Examination of the Effects of Requirements Changes on Software Releases

George Stark, *IBM Global Services*
Al Skillicorn, *The MITRE Corporation*
1st Lt. Ryan Ameele, *U.S. Air Force*

Requirements are the foundation of the software release process. They provide the basis to develop budgets, schedules, and design and testing specifications. Changing requirements during a software release process impacts the cost, schedule, and quality of the product that results. We have collected data on 40 software releases in our environment to understand the source, magnitude, and effects of changing requirements on software maintenance releases. The benefits received include better management of releases and improved customer communications.

Several authors have noted that maintenance of software systems intended for a long operational life pose special management problems [1-3]. The Software Engineering Institute believes that organizational processes are a major factor in the predictability and quality of software [4]. J. Arthur and K. Stevens explain that descriptiveness, completeness, and readability of software documentation are key factors affecting system maintainability [5]. Additionally, M. Hariza, et al., B. Curtis, and C. Yuen all conclude that programmer experience is at least as important as code attributes in determining the complexity associated with software maintenance [3,6,7]. Research by the Standish Group and W. Wayt Gibbs indicates that a low software success rate results from poor requirements and poor risk management [8, 9]. Therefore, software maintenance planning and management should be formalized and quantified.

Requirements are the foundation of the software release process. They provide the basis to develop budgets, schedules, and design and testing specifications. In the maintenance environment, requirements are gathered through change requests from a variety of people including decision makers, system operators, developers, and external interface teams. These people have different backgrounds and different levels of understanding of computers and system operations. This diversity often leads to misinterpretation of the intent of the change description, which can change the scope of the requirement.

Furthermore, throughout the release process, requirements often change.

During release planning, requirements analysis, design, and test reviews, new priorities are established, and changes to the release content are requested in the form of change requests being added or deleted from the release. This requirements volatility makes it difficult to develop dependable release schedules and budgets. B. Curtis, H. Krasner, and N. Iscoe conclude that accurate problem domain knowledge is critical to the success of a project, and requirements volatility causes major difficulties during development [10]. Although these conclusions confirm most people's intuitions concerning requirements volatility, they are not precise enough to help managers take effective action on their projects. M. Lubars, C. Potts, and C. Richter went further by interviewing 23 project teams and recommending organizational solutions rather than technological solutions to the requirements analysis issue [11]. In no case did they find a coherent relationship between requirements analysis and project planning.

This article therefore has two major goals: first, to present an organization's data regarding the source, timing, and impact of requirements volatility on the project planning process; and second, to describe opportunities for management action in the project planning process.

Organizational Data on Requirements Volatility and Project Planning

The Organization and the Data Collected

In 1994, the Missile Warning and Space Surveillance Sensors (MWSSS) Program

Management Office was assigned responsibility for the maintenance of seven products executing in 10 locations worldwide. Combined, the products contained 8 million source lines of code written in 22 languages. Some of the systems were more than 30 years old, and the newest system became operational in 1992. They all operated in hard real-time environments and had a small set of users. To support the management of these products, we instituted the measurement program defined in [12].

In this project environment, a requirement was defined as an approved change request. The customer and supplier agreed to a set of requirements and a project plan to deliver a new version of a product. A requirements change was either an added change request, a deleted change request, or a change in scope to an agreed-on change request in the version content. Because requirements management was a primary factor in our success, we collected data on

- Type of requirement.
- Planned and actual effort days for each requirement.
- Planned and actual number of calendar days for a version.
- Requirements changes made to the version after plan approval—type of change, requesting group, and impact.

Who, How Often, and What Kind of Requirements Changes

To better understand our environment and how to improve it, we needed to answer the following questions.

- Who requests requirements changes?

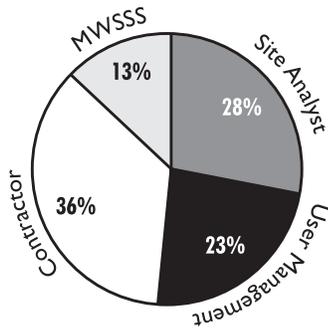


Figure 1. Requirements changes by source.

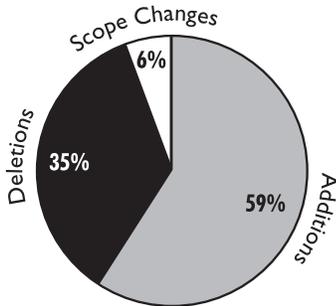


Figure 2. Requirements changes by type.

- How often do our releases experience requirements changes?
- What kind of changes are most common?
- How much effort is associated with individual requirements?

Four groups contributed to the release process: contractor development team, acquisition management team, user management, and site analysts. Each of these groups contributed to the requirements changes associated with a release. Figure 1 shows the percent of changes made by each group.

Requirements volatility comes in three types: additions to the delivery

content, deletions from the delivery content, and changes in scope to an agreed-on requirement. A total of 108 requirements changes were made during 40 software releases since 1994. Figure 2 shows the distribution of these changes by type. Additions to the release content were the most common form of change, followed by deletions, with scope change being relatively rare.

Figure 3 shows the requirements volatility for each of the 40 deliveries. Fourteen of the 40 deliveries (35 percent) had no requirements change. Of the 14 deliveries, six were made on or ahead of schedule, four were within 15 percent of the original scheduled date, and four were more than 15 percent late. Twenty-six of the 40 (65 percent) had requirements change, with eight of them having greater than 50 percent change. Of the 26 releases that experienced requirements change, 16 had requirements added, 15 had deletions, and four had scope changes. Seven releases had a combination of adds, deletes, or changes.

To understand how much effort was associated with individual changes, we developed the software change taxonomy shown in Table 1. It includes 10 types of changes and root causes for each change type.

We categorized the changes delivered in eight releases using this taxonomy, which consisted of 104 modification changes (43 percent) and 139 fix changes (57 percent). Figure 4 is a Pareto diagram of this change data. The left vertical axis shows the number of changes attributed to each class, and the right vertical axis represents the cumulative

percentage of defects and is a convenient scale from which to read the line graph. The line graph connects the cumulative percents (and counts) at each category.

Table 1. Software change taxonomy.

Change Type	Root Cause
Computational	Incorrect operand in equation.
	Incorrect use of parentheses.
	Incorrect or inaccurate equation.
	Rounding or truncation error.
Logic	Incorrect operand in logical expression.
	Logic out of sequence.
	Wrong variable being checked.
Input	Missing logic or condition test.
	Loop iterated incorrect number of times.
	Incorrect format.
Data Handling	Input read from incorrect location.
	End-of-file missing or encountered prematurely.
	Data file not available.
	Data referenced out-of-bounds.
Output	Data initialization.
	Variable used as flag, or index not set properly.
	Data not properly defined or dimensioned.
	Subscribing error.
Interface	Data written to different location.
	Incorrect format.
	Incomplete or missing output.
Operations	Output garbled or misleading.
	Software and hardware interface.
	Software and user interface.
Performance	Software and database interface.
	Software and software interface.
	COTS or GOTS software change.
Specification	Configuration control.
	Time limit exceeded.
	Storage limit exceeded.
Improvement	Code or design inefficient.
	Network efficiency.
	System-to-system interface specification incorrect or inadequate.
	Functional specification incorrect or inadequate.
	User manual or training inadequate.
	Improve existing function.
	Improve interface.

Figure 3. Requirements volatility for 40 deliveries.

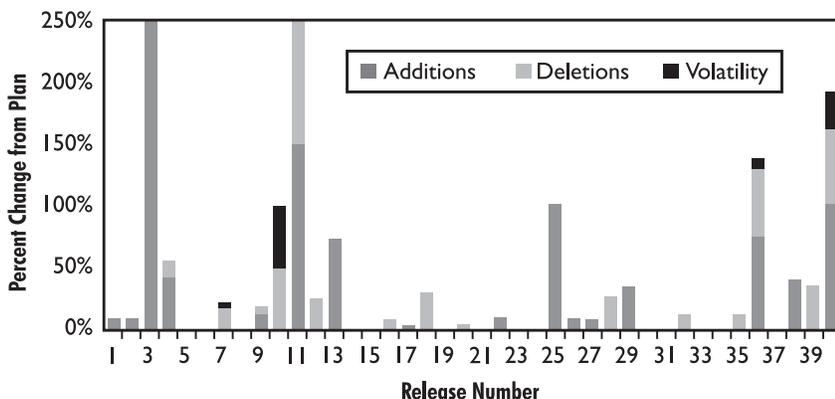


Figure 4 indicates that logic changes to the software are most common (45 changes or 19 percent of the total). (Although not shown in Figure 4, the majority root cause is missing logic or condition tests for error handling.) Using this information, we have our design and code reviews to specifically look for these logic problems. Only two of the 243 changes involved data input problems.

Figure 5 is a Pareto diagram of the effort required to make each change. It shows that although changes based on specification changes only ranked fourth in number of changes with 26, they accounted for 20 percent of the total effort at 591 staff-days. Logic changes fall to sixth when viewed in this manner.

The information from this analysis helped maintenance engineers make better requirements cost estimates. By reviewing change requests and accurately assigning them to the change taxonomy, they could estimate the staff-days required to design, code, and test changes. For example, the average staff-days of effort required for changes to interface requirements are 24 staff-days with a standard deviation of 50 staff-days, whereas the average for functional specification changes is 23 staff-days with a standard deviation of 29 staff-days. Next, the actual was tracked against the estimate, and the taxonomy and cost information was updated as each release was completed.

Although the current information is highly variable for each root cause, the effort data is expected to converge around a reasonable mean as more data is collected. This will increase our confidence in the estimates. Sudden changes could indicate a need to re-examine our processes or a need to change the staff that implements the requirement. Even with the current variability, using historical data is the best method to estimate individual change effort.

A Microview of Requirements Changes on One Release

The Configuration Control Board approved a release plan to deliver 17 requirements in nine months at a cost of approximately \$490,000. Figure 6 shows the requirements changes over time for this release. These changes were processed both formally (through the Configuration Control Board) and informally (agreement between users and developers). The figure also shows that a total of 20 changes were made to the release content in the 14 months since project plan approval. The two spikes for February and October occurred after design reviews where major scope changes occurred with some of the requirements. Nine of the changes occurred in the last five months of the effort, and only six of the delivered requirements were a part of the original approved plan. This greatly impacted the implementation effort.

Requirements Changes by Type

Figure 7 shows a significantly different distribution of changes by type than the overall distribution of Figure 3. In Figure 7, scope changes account for 26 percent of the changes to the release compared with 8 percent for all releases. The increase in

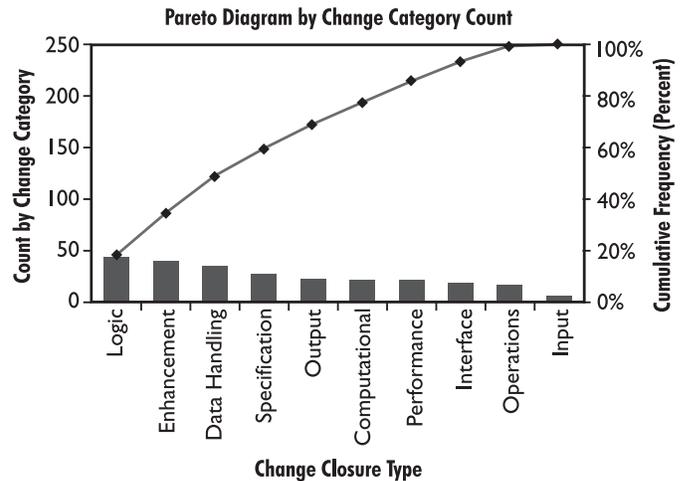


Figure 4. Software maintenance changes by type.

the amount of scope changes was a major factor in the delivery schedule, which illustrates two important points: general distribution should only be used as a planning guide, and releases should be managed as stand-alone projects.

Requirements Changes by Source

Requirements changes could be initiated by the customer (analysts or management personnel) or the development team, i.e., the contractor or the MWSSS Program Management Office. Figure 8 shows the distribution of changes by source for this release. This chart shows that the changes were distributed as 55 percent driven by the development team and 45 percent by the customer. The analyst personnel and the devel-

Figure 5. Staff-days of effort by category.

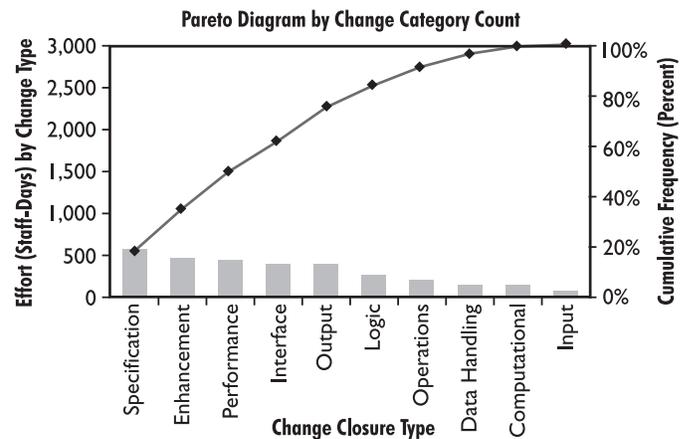
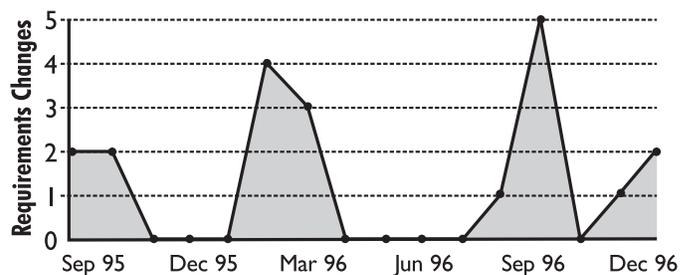


Figure 6. Requirements changes by month for one release.



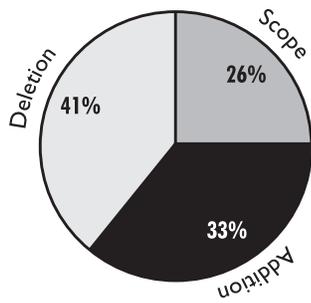


Figure 7. Requirements changes by type.

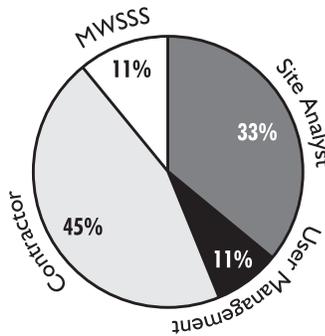


Figure 8. Requirements changes by source.

opment contractor accounted for 80 percent of the changes (16 out of 20).

Schedule, Cost, and Quality Impact of Changes

Figure 9 shows the predicted version operational date over time for the project. The first slip (three and one-half months) was reported at the design review held three months after project start. A second slip (three weeks) was announced eight months into the project. Finally, another completion date, this one four and one-half months later, was announced one year after project start. These announced schedule slips correspond to the major jumps in the requirements changes graph (Figure 6). Two defects that required rework and more testing were reported during operational testing of this release. The release was delivered a month later, making the total schedule 10 months (more than double the original project plan) and the cost \$100,000 (22 percent over budget). Of course, requirement volatility was not the only reason for the schedule and cost overrun, but it was the major factor.

Observations and Recommendations

Requirements must be more clearly explained and understood by the development team, and change agreements must be more formally managed by the management team responsible for the software releases. Accordingly, we changed our process to include a rigorous requirements review meeting with the customer prior to presenting the release plan for Configuration Control Board approval. We also have biweekly meetings with the MWSSS management where the project requirements status and other project issues are briefed.

A Macromodel to Forecast the Effects of Requirements Changes on Releases

To help release teams better manage the requirements volatility and get a handle on the impact of changes to their project, we began to develop models based on the historical release data. Table 2 shows the percent of planned schedule achieved (100 means the plan was met, greater than 100 means late, less than 100 means delivered early), the square root of the percent of requirements volatility (the sum of all changes), and the productivity risk associated with the 20 releases. The square root transformation was used to spread out the numbers close to zero and condense the numbers greater than one. Risk is defined as changes closed per effort days available.

Figures 10 and 11 are scatter plots of the percent of planned schedule vs. the other two descriptive variables in Table 2. Individually, these plots have little correlation, but used together, these variables can provide insight to project managers to help them understand the schedule impact of requirements changes.

A linear regression analysis was performed on the data to develop a model to predict schedule impact based on requirements volatility and risk with the following results:

$$Y = 0.97 + 0.41 * X^{1/2} + 0.23 * Z \quad (1)$$

where

- Y = Percent Schedule Change
- X = Requirements Volatility
- Z = Risk

The proportion of variance explained by this model (R^2) is 0.72, and the standard error of the estimate is 0.17. Notice that the schedule change goes up regardless of whether the requirements changes were an addition or deletion because the input to the model is percent of requirements changes. This is a topic of debate in the organization: Some argue that removing requirements involves effort to change the design and test procedures, whereas others argue that a reduction in requirements means less work for the team and earlier completion of the project.

Figure 12 shows the results of applying the model to all 40 releases executed by our organization. From this figure, it can be seen that the model performs much better in the 115 percent to 130 percent of planned schedule range and yields more optimistic results as predictions get larger, i.e., greater than 150 percent of plan. This may indicate the

Table 2. Schedule, requirements volatility, and risk data for 20 software maintenance versions.

Version Content	Percent of Planned Schedule	SQRT (Percent of Requirements Change)	Risk (Change Requests per Staff-Day)
1	108	33	0.14
2	104	32	0.15
3	168	158	0.50
4	132	76	0.18
5	115	0	0.08
6	115	48	0.27
7	118	45	0.16
8	139	100	0.01
9	219	158	0.19
10	129	50	0.07
11	100	87	0.07
12	111	0	0.01
13	102	18	0.12
14	123	55	0.07
15	92	0	0.20
16	178	245	0.01
17	104	0	0.02
18	110	23	0.08
19	100	31	0.12
20	94	0	0.03

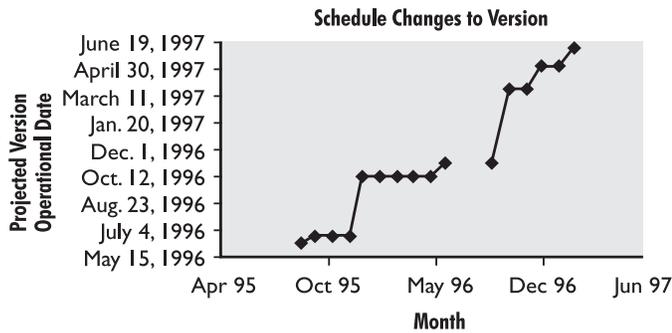


Figure 9. Predicted version operational date by month.

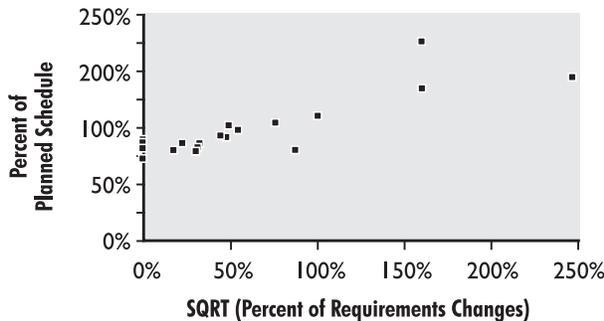


Figure 10. Percent of planned schedule vs. square root (requirements volatility) for 20 versions.

need for another explanatory variable as major changes occur to releases.

Macromodel Use and Benefits

We have used this equation to explain the expected impact of changes to the delivery plan as they arise. For example, one version contained 15 planned requirements scheduled for delivery in 91 calendar days—the customer wanted to drop two of the requirements and change the scope of a third at preliminary design. Managers estimated the risk to version delivery to change from 0.14 (15 changes in 108 staff-days) to 0.1 (13 changes in 130 staff-days). Using the model, managers forecasted the overall schedule impact to be $[0.97 + 0.41*(0.2)^{1/2} + 0.23*(0.1)] = 1.18$ or an 18 percent schedule slip. An 18 percent slip is equivalent to 16 days added to the 91-day schedule. These 16 days would have cost the customer an additional \$60,000.

During discussion about the model and the prediction, the customer decided that this schedule slip was not acceptable to the overall mission of the version; therefore, they decided not to pursue the changes but to incorporate the scope change in the next release. The metrics-based model facilitated objective communication with the customer concerning version release plans and status.

The model forecasted a \$50,000 cost impact and a 12-day schedule slip from a second customer request to change the release content. The additional cost was not acceptable to the customer, so they decided to incorporate the changes in the next release. Thus, the overall cost avoidance because of quantitative schedule impact analysis was \$110,000.

Conclusion

Requirements management involves establishing and maintaining an agreement between the customer and the supplier on the specific number and technical content of the performance and functionality that will be included in a software release. This agreement forms the basis to estimate, plan, perform, and track the project's activities. We believe other organizations can benefit from our experience.

Acknowledgments

We thank Dieter Rombach for his suggestions and for providing references for this article. We also thank the many referees for their excellent reviews. ♦

About the Authors

George Stark is a programming consultant with the IBM Corporation in Austin, Texas. Previously, he was a principal scientist with The MITRE Corporation, where he supported the software efforts of the MWSSS Program Management Office. His technical interests include software metrics and reliability for management decision making. He has been involved in software reliability measurement for 15 years and was the vice chairman of the American Institute of Aeronautics and Astronautics blue-ribbon panel on software reliability. He has been the manager of software testing and reliability for a local loop fiber-optic telephone system. He received the Johnson Space Center Quality Partnership Award and the MITRE General Manager's Award for contributions to software measurement. He has a bachelor's degree in statistics from Colorado State University and a master's degree in mathematics from the University of Houston.

Figure 11. Percent of planned schedule vs. delivery risk.

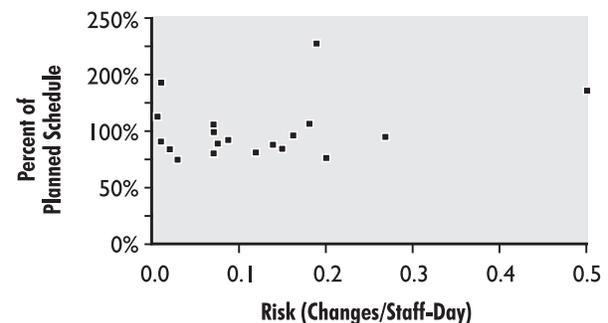
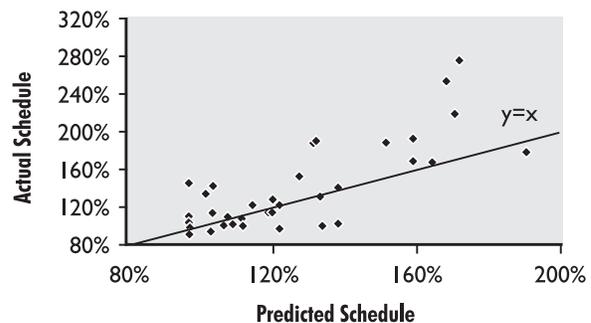


Figure 12. Actual vs. predicted schedule using linear model for all 40 releases.



Do You Acquire Software but Need More Expertise?

Because of all the cutbacks, you are not alone. Without understanding the delivered software and documentation, you cannot assure the taxpayer of a good purchase. At the Software Technology Support Center (STSC), we have helped organizations at numerous Air Force, Army, and Navy locations make

more technically informed buys. Available on a just-in-time basis, we will help your organization strengthen its position. Whether your acquisition involves embedded or information management systems, call us for an exploratory discussion of how the right expertise can provide peace of mind.



The STSC Provides These Services and More

- Technical Documentation Inspection Services
- Independent Documentation Audit
- J-STD-016-1995 Training



Paul Hewitt
Voice: 801-775-5742 DSN 775-5742
E-mail: hewittp@software.hill.af.mil



Reed Sorensen
Voice: 801-775-5738 DSN 775-5738
E-mail: sorensen@software.hill.af.mil

IBM Global Services
11400 Burnet Road, MD 3901
Austin, TX 78759
Voice: 512-823-8515
Fax: 512-823-3385
E-Mail: gstark@us.ibm.com

Al Skillicorn is a member of the technical staff of The MITRE Corporation. He supports the software maintenance of the early warning radar systems. Among his other responsibilities are the Year 2000 problem and future software architectures. He has a bachelor's degree in engineering from the U.S. Military Academy at West Point. Previous work included communications modeling and analysis for the Regency Net Communication System and for the Theatre Nuclear Forces Communications System in Europe.

The MITRE Corporation
1150 Academy Park Loop #212
Colorado Springs, CO 80910
Voice: 719-556-2565
E-mail: skilliad@cisf.af.mil

1st Lt. Ryan Ameele is the software process manager for the MWSSS Program Management Office. Previously, he was the Cargo System Software Development Team leader for the Air Mobility Command Computer System Squadron at Scott Air Force Base, Ill. He has a

bachelor's degree in engineering from Clarkson University in New York. He was recently selected for promotion to captain.

SSSG/SDWSE
1050 E. Stewart Avenue
Peterson AFB, CO 80914-2902
Voice: 719-556-9906
E-mail: ameele1@cisf.af.mil

References

1. Card, D.N., D.V. Cotnoir, and C.E. Goorevich, "Managing SW Maintenance Cost and Quality," *Proceedings of the International Conference on Software Maintenance*, September 1987.
2. Chapin, N., "The Software Maintenance Life-Cycle," *Proceedings of the International Conference on Software Maintenance*, 1988.
3. Hariza, M., J.F. Voidrot, E. Minor, L. Pofelski, and S. Blazy, "Software Maintenance: An Analysis of Industrial Needs and Constraints," *Proceedings of the International Conference on Software Maintenance*, Orlando, Fla., 1992.
4. Software Engineering Institute, "Software Process Maturity Questionnaire Capability Maturity Model, Version 1.1," Carnegie Mellon University, Pittsburgh, Pa., 1994.
5. Arthur, J. and K. Stevens, "Assessing the Adequacy of Documentation Through Document Quality Indicators," *Proceedings of the International Conference on Software Maintenance*, 1989.
6. Curtis, B., "Conceptual Issues in Software Metrics," *Proceedings of the IEEE International Conference on System Sciences*, 1986.
7. Yuen, C., "An empirical Approach to the Study of Errors in Large Software Under Maintenance," *Proceedings of the International Conference on Software Maintenance*, 1985, pp. 96-105.
8. The Standish Group, "The Scope of Software Development Project Failures," Dennis, Mass., 1995.
9. Gibbs, W. Wayt, "Software's Chronic Crisis," *Scientific American*, September 1994, pp. 72-81.
10. Curtis, B., H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, Vol. 31, No. 11, 1988, pp. 1268-1287.
11. Lubars, M., C. Potts, and C. Richter, "A Review of the Practice in Requirements Modeling," *Proceedings of the International Symposium on Requirements Engineering*, 1996, pp. 2-14.
12. Stark, G.E., "Measurements for Managing Software Maintenance," *Proceedings of the International Conference on Software Maintenance*, Monterey, Calif., November 1996, pp. 152-161.

Four Roads to Use Case Discovery

There Is a Use (and a Case) for Each One

Gary A. Ham
Battelle Memorial Institute

Use case-based requirements definition is a hot topic, particularly in object-oriented software engineering circles. Appropriate content is achieved by looking at potential use cases from four different views. Each view provides unique advantages. Together they offer the information needed to develop the fully elaborated use cases that facilitate clearly defined, understandable, measurable, and testable design.

Requirements analysis includes both the gathering of functional and system requirements and the organization of those requirements into a logical, traceable, and understandable form. It is one of the most discussed and least well-implemented parts of the software engineering process. As a result, poor requirements analysis is a leading cause of failure in systems development [1]. To address this situation, use case-based requirements definition is becoming popular for systems analysis in general and object-oriented development in particular.

Although use cases are well accepted in principal, the form a use case should take, the level of granularity it should encompass, and even the specific definition of the term “use case” are still matters of dispute in the industry. As a result, most Department of Defense (DoD) contracting officials still prefer to see traditional structured methods and good old-fashioned “shall” statements for requirements definition. This article introduces how to “find” use cases and what it takes to elaborate use cases into effective tools for user validation, operational metrics, and system design. Interestingly, use case can be implemented without throwing away the value of the traditional shall statements and without tossing mission-based structured decomposition out the window.

Use Case Definition

A use case is a sequence of events, performed through a system, that yields an observable result of value for a particular actor.¹ The key issue for requirements management in this definition is the words “observable result of value.” The primary goal of requirements definition

should be the provision of value. Because use cases, by definition, fit that goal, they are used as the primary organizational structure for requirements definition. The additional components needed of a fully elaborated use case are

- Actors that collaborate in the use case.
- Events (and associated business rules) in which the actors collaborate.
- Information that is passed and returned in the course of each collaboration.
- Context (environment) in which the use case takes place.

Context can best be defined in terms of additional requirements that affect the use case in terms of inputs, controls, outputs, and mechanisms.² Input descriptions are requirements associated with what input is available and in what form it can or should be provided. Controls impose algorithmic restrictions on how and when the use case must be performed by prescribing rule sets and regulations that are mandatory. Output descriptions add specific formatting and content requirements to the basic use case product. Finally, mechanism prescriptions are associated with architectural requirements in the sense of logical interfaces to current or planned systems. Enabling collaborations with actors beyond the primary actor that will or must interface in the prescribed use case are also identified.³

Four Ways to Create Use Cases

There are essentially four ways to create use cases in sufficient detail to be included in formal requirements in a form suitable to generate implementable systems design and test specifications:

- **Mission decomposition** – a form of traditional hierarchical structured analysis.
- **Unstructured aggregation** – collect and classify traditional shall statements.
- **Scenario story-driven discovery** – use cases are discovered by analyzing written descriptions of day-to-day activity or desired activity.
- **Actor or responsibility discovery** – first define the actors and roles, then define their collaborations and responsibilities.

Mission decomposition begins with a particular mission goal. The goal must be a clear statement. It may not, in and of itself, have clear metrics for its achievement. If so, the goal must be decomposed into components in a fashion analogous to use of the goal, question, metric (GQM) paradigm that is often advocated to discover software metrics [3]. Beginning with mission defined in terms of a goal, question what accomplishments (products, services, etc.) are required to reach the goal. Decomposition continues until all of the lowest-level accomplishments can be described in terms of a measurable result for a specific user in support of the top-level mission. In other words, decomposition continues until each “leaf node” accomplishment contains the basic output specification for a use case. These output specifications then become the definition around which use case elaboration takes place. Elaboration includes identifying the events, actors, information structures, business rules, and non-functional requirements that apply to the particular mission component.

Unstructured aggregation is used to collect and classify requirements col-

lected from various venues in the form of shall statements and business rules.⁴ Any active voice shall statement that describes an individually measurable product or service that must be provided for a particular actor becomes a candidate use case. All other requirements are reviewed for their applicability to the use cases discovered. Generally, these additional requirements are applicable in a descriptive sense as input, output, control, and mechanism requirements depending on how they will affect the further development of the candidate use cases.

A scenario story is a detailed description of all the interactions by one or more users with the system in a set of related events. The story should include details that describe user interaction with the system in a detailed, concretely specified and verifiable form. The scenario story is used for both requirements elicitation and user validation. When written properly, scenario story paragraphs form potential use cases, and sentences within those paragraphs describe the events involved in performing the use case.

Actor, responsibility, and collaboration discovery is a traditional object-oriented analysis technique that begins with finding roles that actors play, what responsibilities they have for task accomplishment, and what other actors they must collaborate with to accomplish those tasks.⁵ Use cases are discovered by identifying productive task results. Subtasks leading to those results become events within an identified use case.

So, which approach to use case discovery is best? Each approach offers advantages. GQM-based mission decomposition offers measurable results and a focus on mission rather than fluff and “nice to have.” Shall statements allow formal integration of nonfunctional and architectural concerns into analysis and provide specific reference to requirements, e.g., performance response times, that may apply to multiple use cases. Scenario stories offer a completeness of detail and an effective user validation viewpoint that is difficult to achieve with other approaches. Scenarios are also of great value in obtaining even-

tual user acceptance of new or changing systems. Finally, no matter which use cases are identified, they cannot be put together until the actors and collaborators in events are identified.

Each approach also has limitations. It is sometimes difficult to obtain mission focus from untrained subject matter experts, even in facilitated workshops. Merely getting consensus on the mission can be an interesting task in some environments. It is much easier to ask, “What do you do each day?” Theoretically, shall statements are individually verifiable and can be clearly written, at least in the microsense. However, because of their “atomic” nature, this is usually not the case. Most of the time, shall statements are poorly organized, ambiguously stated, and difficult to implement or test. They are often redundant and overlapping, yet designers often find large gaps when basic modules are built. Scenarios tend to focus on current process and change based on current process. This makes it harder to think outside of the box. Beginning with scenarios also tends to add requirements that benefit particular users rather than benefit the mission to be accomplished—fluff happens. Finally, the purely bottom-up actor and responsibility approach raises completeness questions and a concern that generated use cases might reflect individuals’ requirements ahead of organizational mission needs. There also are questions about the effectiveness of role and class abstraction in a bottom-up “find the nouns” type of environment.⁶

Since each approach has both benefits and limitations, the question of where to start becomes one of basic expediency. Start with whichever entry point offers the most initial return in information. You can begin with what is most comfortable for the organization under analysis or for the team doing the analysis. You can also reuse existing documentation. If initial scenario stories are available, use them. If prior business process reengineering work has left clear mission descriptions, or defined organizational role and responsibilities definitions, use them. If all you have are large documents filled with poorly structured

(or well structured) shall statements, use them, too. The rest of the analysis information can be added at any point, as long as the use case structure to which it is added remains consistent.

Use cases are not requirements in and of themselves. Instead, use cases provide a showcase in which requirements are precisely organized and illustrated for user validation, system design, and test script development. To be effective, a use case needs the following:

- A measurable contribution to a defined mission in support of a primary actor.
- A clear definition of input, output, control, and mechanism-related requirements and business rules.
- A presentation format that facilitates functional user validation and change elicitation.
- An understandable presentation of roles and collaborations by event in a sequence as a basis to assign and find class operations.

Each of the above needs is best served by a different one of the four approaches. So, achieving a high level of effectiveness implies that all four approaches are eventually needed for complete analysis. Leaving one out will reduce the value of that use case as system design or test script development documentation. As long as a defined process to maintain traceability and coordination between approaches is maintained, the particular initial approach is not material. The measure of success will be the clearly defined, understandable, testable designs that result from fully elaborated use cases.

Disclaimer and Acknowledgments

The views expressed in this article are my own (as the author) and do not formally represent those of the DoD or Battelle Memorial Institute. They represent my distillation of collective team member experience in support of the Computer-Based Patient Record Interoperability using Object-Oriented Technology project for the Office of the Assistant Secretary of Defense for Health Affairs. Csaba Eghazy, Scott Eyestone, Carol Fogelson, Don Heim, and Janet

AMC CSS Achieves CMM Level 3

The Air Mobility Command (AMC) Computer Systems Squadron (CSS), Scott Air Force Base, Ill. received a Level 3 rating during a Software Engineering Institute (SEI) Capability Maturity Model (CMM) assessment. The CSS currently has over 450 employees dedicated to developing, maintaining, and enhancing transportation and command and control software systems for AMC. The assessment culminated 17 months of dedicated hard work.

One requirement to achieve Level 3 was to develop and maintain a usable set of software process assets that improve performance across all projects and provide a basis for cumulative, long-term organizational benefit. They developed a process asset library (PAL) located on the Web at http://cpssweb.safb.af.mil:81/pal/pal_home.htm. The AMC CSS PAL is accessible to everyone within the military and government Internet domains.

Martino provided valuable insight that is reflected in some form in this article. ♦

About the Author



Gary A. Ham is a senior research scientist for Battelle Memorial Institute, National Security Division, Information Systems Engineering and Process Modernization Department in Arlington, Va. A former Marine Corps comptroller and Naval Academy computer science instructor, he currently researches value metrics definition processes to support object-oriented requirements analysis and design of DoD systems. He has a bachelor's degree in economics from Whitman College in Walla Walla, Wash. and a master's degree (with distinction) in information systems management from the Naval Postgraduate School in Monterey, Calif. He is currently a doctoral candidate in information technology at George Mason University in Fairfax, Va.

Principal Research Scientist
Battelle Memorial Institute
2101 Wilson Blvd., Suite 800
Arlington, VA 22201-3008
Voice: 703-575-2118
Fax: 703-671-9180
E-mail: ham@battelle.org

References

1. Research Report, "Chaos," The Standish Group, 1995, <http://www.standishgroup.com/chaos.html>.
2. Jacobson, Ivar, Martin Griss, and Patrick Jonsson, *Software Reuse: Architecture,*

Process, and Organization for Business Success, ACM Press, New York, N.Y., 1997.

3. Fenton, Norman E., *Software Metrics, a Rigorous Approach*, Chapman and Hall, London, 1994.

Notes

1. Ivar Jacobson's basic definition differs slightly: "A use case is a sequence of transactions performed by a system, which yields an observable result of value for a particular actor" [2]. For our purposes, an actor is defined as a participant in a use case event, as an instigator, a provider of service or product, or as a recipient of that service or product.
2. If this sounds a little like Integration Definition for Function Modeling (IDEF0), it should. IDEF0, with a difference in focus from functional decomposition to product or service identification, can effectively be used to identify mission-focused use cases. The required change in mindset may be difficult for traditional IDEF0 modelers. It was for me. If you can make the transition, however, a whole new approach to software metrics based on activity-based costing becomes available.
3. The particular form that a use case should take is less important than the content. The only requirement is a consistent presentation of use case contents that provides clear understandability by subject matter experts. The use of formal notation languages, e.g., Unified Modeling Language and predicate logic, should be left out unless the user community is fully conversant in the notation presented. We use a standard format for our

use cases. This "standard" has, however, been adjusted in each analysis iteration to better meet the understandability needs of our validating users.

4. Business rules are defined to be requirements that contain a conditional phrase, e.g., "if," or "then." Business rules are designed to govern the actions of an event or events, either singly or grouped in a rule base. In some references, nonconditional rules are referred to as business rules. My current project merely calls such rules requirements. We feel the distinction is important because business rule sets can be used within rule engines to process events depending on condition. Straight requirements act regardless of condition.
5. Although analogous, this is not the same as the class, responsibility, and collaboration approach. We are defining roles that will probably (but may not) be assigned to classes as part of the design process. We do not try for class definition in analysis use case development. We save that for design, when architectural dependency issues are more completely specified.
6. Yet, we have used this approach extensively for project management. All of our task statement development and project work breakdown structures are based on the definition of responsibilities and collaborations between project teams where project teams are recognized as actors or classes in the object-oriented sense. Project management is object management to the extent that Gantt charts are defined by a composition of sequence diagrams developed from the original collaboration diagrams.

Doing Requirements Right the First Time

Theodore F. Hammer, *Goddard Space Flight Center*
Leonore L. Huffman and Linda H. Rosenberg, *Unisys Federal Systems*

The criticality of correct, complete, testable requirements is a fundamental tenet of software engineering. The success of a project, both functionally and financially, is directly affected by the quality of the requirements. Also critical is the complete requirements-based testing of the final product. This article addresses three critical aspects of requirements: definition, verification, and management. Project data collected from NASA Goddard Space Flight Center (GSFC) by the Software Assurance Technology Center (SATC) will be used to demonstrate these concepts and explain how any project, large or small, can apply this information.

It is generally accepted that requirements are the foundation upon which the entire system is built. Also accepted is that requirements verification and validation is needed to assure that the functionality specified in the requirements has been delivered. However, all too often, requirements are not satisfied, which means you fix what you can and accept that certain functionality will not be there. A better approach is to get the requirements right the first time. Complete, concise, and clear requirements will give the implementer a precise blueprint with which to build the system. Getting the requirements right is not done by magic but through the application of tools and metric analysis techniques in requirements specification, requirements verification, and requirements management.

Because both parties must understand the requirements that the acquirer expects the provider to contractually satisfy, specifications are usually written in natural language. The use of natural language to prescribe complex, dynamic systems has at least two severe problems: ambiguity and inaccuracy. Many words and phrases have dual meanings that can be altered by the context in which they are used. To define a large, multidimensional capability within the limitations imposed by the linear, two-dimensional structure of a document can obscure the relationships between individual groups of requirements. The first part of this article looks at types of requirements-specification terminology,

some of which can contribute to ambiguity and misinterpretation.

Requirements-based testing is critical to the implementation of software systems. Automated tools, if properly used, open the door to assess the scope and potential effectiveness of the test program. A wealth of information can be obtained through proper implementation of a database that tracks requirements at each level of decomposition and the tests associated with the verification of these requirements. From this database, the project can gain important insight into the relationship between the test and requirements. The second part of this article outlines some of the important insights into NASA project test programs developed from analyses of this type.

Requirements management is a volatile, dynamic process. The skill with which project workers maintain, keep current, track, and trace the project's set of requirements affects every phase of the project's software development lifecycle—including maintenance. Months or years before project completion, effectively managed requirements determine how, when, and how expensively completion will take place.

Before processing requirements, the schema for the requirements management database must be developed. The final portion of this article describes some critical issues identified by the SATC that are needed to effectively manage requirements databases. It also discusses lessons learned on how to effectively design and maintain requirements databases.

Development Environment

To demonstrate how metrics can provide the insight needed to get the requirements right, data from a large NASA project, Project X, will be used. This anonymous project implements a large system in three main incremental builds.¹ The development of these builds is overlapping, design and coding of the second and third builds starting before the completion of the first build. Each build adds new functionality to the previous build and satisfies a further set of requirements.

NASA defines requirements in four levels of detail. "Mission-Level Requirements" for the spacecraft and ground system are System Level 1; they are the highest level and rarely, if ever, change. Level 1 requirements then undergo decomposition to produce "Allocated Requirements," called Level 2; these also are high level, and change should be minimal. Level 2 requirements are then divided into subsystems, and a further level is derived in greater detail, hence, "Level 3: Derived Requirements." Generally, contracts are bid using this level of requirements detail. Each requirement in Level 2 traces bidirectionally to one or more requirements in Level 3. "Detailed Requirements" are found in Level 4; these are used to design and code the system. There also is bidirectional tracing between Level 3 requirements and Level 4 requirements. To verify the requirements, two stages of testing are used. System tests are designed to verify the Level 4 requirements, then acceptance tests are used to verify the Level 3 requirements.

Requirements Specification

The importance of correctly documenting requirements has caused the software industry to produce a significant number of aids [1] to create and manage requirements specification documents and individual specifications statements; however, few of these aids help evaluate the quality of the requirements document or the individual specification statements. The SATC has developed a tool to parse requirements documents. The Automated Requirements Measurement (ARM) software was developed to scan a file that contains the text of the requirements specification. The software searches each line of text for specific words and phrases that are indicated by the SATC's studies to be an indicator of the document's requirements specification quality. ARM has been applied to 56 NASA requirements documents, and seven measures have been developed.

- **Lines of Text** – Physical lines of text as a measure of document size.
- **Imperatives** – Words and phrases that command that something must be done or provided, e.g., shall, must, will, should, is required to, are applicable, and responsible for. The number of imperatives is used as a base requirements count.
- **Continuances** – Phrases that follow an imperative and introduce the requirements specification at a lower level for a supplemental requirements count, e.g., as follows, following, listed, in particular, and support.
- **Directives** – References provided to figures, tables, or notes, e.g., figure, table, for example, and note.
- **Weak Phrases** – Clauses that are apt to cause uncertainty and leave room for multiple interpretations or a measure of ambiguity, e.g., adequate, as applicable, as appropriate, as a minimum, be able to, be capable, easy, effective, not limited to, and if practical.
- **Incomplete** – Statements within the document that have “TBD” (to be determined) or “TBS” (to be supplied).
- **Options** – Words that seem to give the developer latitude to satisfy the

	Lines of Text	Imperatives	Continuances	Directives	Weak Phrases	Incomplete	Options
56 NASA Documents	Minimum	143	25	15	0	0	0
	Median	2,265	382	183	21	37	7
	Average	4,772	682	423	49	70	25
	Maximum	28,459	3,896	118	224	4	32
	Standard Deviation	759	156	99	12	21	20
	Project X	34,664	1,176	714	873	13	480

Table 1. *Requirements specification analysis example.*

specifications but that can be ambiguous, e.g., can, may, and optionally.

It must be emphasized that the tool does not attempt to assess the correctness of the requirements specified. It assesses individual specification statements and the vocabulary used to state the requirements and also has the capability to assess the structure of the requirements document.²

To see how this tool would be used to assess the quality of a requirements document, the Project X Level 3 requirements document was analyzed using the ARM tool. Table 1 shows the results in contrast to statistics from the 56 previous documents.

From this analysis, several things become clear. First, the document shows some strengths: There appear to be a good number of imperatives, and the number of weak phrases is low compared to the family of NASA documents processed through the ARM tool to date; however, the document shows some significant weaknesses. The document has a large amount of text given the number of imperatives. This indicates a wordy document, which can obscure the requirements and prevent them from being clear and concise. The document also has a large number of incomplete requirements that contain TBDs and TBSs—on this point alone, the document can be judged not ready for use. Also, this document has a large number of options, which increases the uncertainty about what is required of the system to be developed. Options

leave decisions about the system to the implementers, many times without sufficient direction or instruction about option selection criteria. As a result, the implementation varies widely, from some of the options to none.

Engineers have always wanted to get the requirements right in the specification, but there has been little available in terms of analysis tools to allow them to visualize the quality of the documentation. Now, with the ARM tool, the quality aspects of the documentation can be visualized, and necessary action can be taken to improve the documentation.

Requirements Volatility

Requirements testing is vital to getting the requirements right. Many times it is overlooked in favor of testing code, but if the software does not conform to the requirements, it is just as defective as if it were full of bugs. Good requirements testing relies on a good verification program, which in turn must rest on an analysis of requirements volatility and linkage. An effective verification program comprises a test profile made after linkage of requirements is analyzed and after considering requirements volatility. Again, data from Project X will demonstrate the utility of metrics in requirements verification.

Requirements stability impacts the verification effort because testing cannot be planned or designed when the requirements are continually in a state of flux. Figure 1 shows how metrics provide insight into requirements stability while also demonstrating the

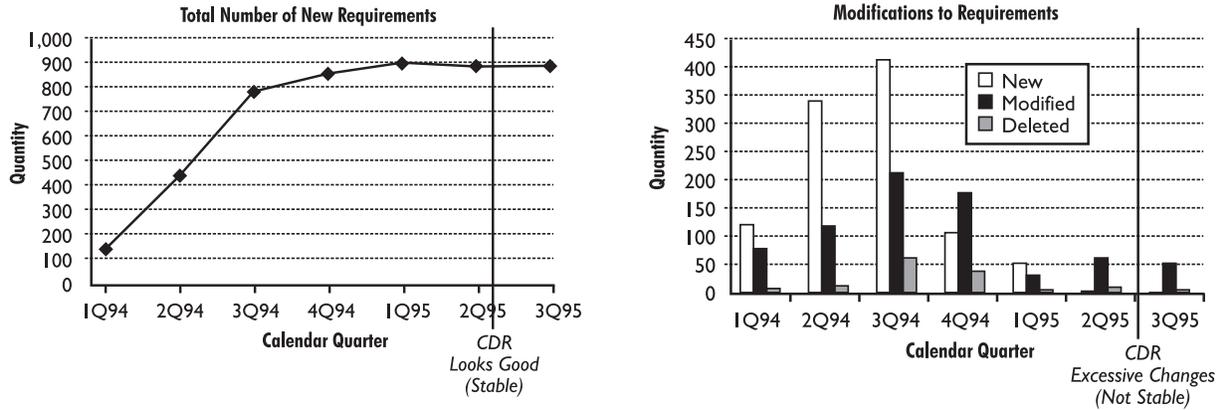


Figure 1. Requirements stabilization—volatility. Combination of both views indicates risk area: Requirements are not yet stable.

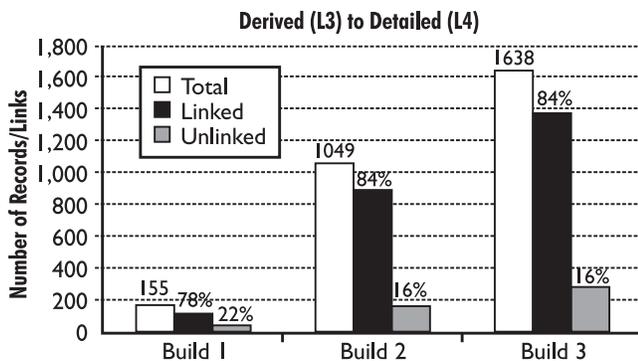
importance of examining an issue from more than one angle. According to the graph on the left side of the figure, the total number of requirements has stabilized in time for the Critical Design Review (CDR); however, the graph on the right shows that the requirements are *not* stable—modifications and deletions are still taking place. This almost constant change in the requirements will endanger the verification program.

Requirements stability can also be viewed in terms of the completeness of requirements traceability. Requirements traceability is the linkage of the requirements at one level to the requirements at the next lower level. Missing linkage may indicate missing requirements. Figure 2 shows the linkage of Level 3 requirements to Level 4 requirements. In all cases, there is missing linkage (white bar of graph) between Level 3 and Level 4 requirements, indicating that the Level 4 requirements may be incomplete for a CDR held for any one of these builds.

Requirements Verification

The objective of an effective verification program is to ensure that every requirement is tested, the implication being that if the system passes the test, the requirement's functionality is included in the delivered system [1, 2]. The traceability of the requirements to test cases therefore needs to be assessed. It is expected that a requirement will be linked to a test case

Figure 2. Requirements traceability.



and may well be linked to more than one test case, as shown in Figure 3 [3, 4].

The important aspect of this analysis is to determine which requirements have not been linked to any test cases.

Figure 4 shows the traceability of requirements to test cases for Project X around the CDR time frame for Build 2. The profiles show several problems. First, the poor traceability between the requirements and test cases for Build 1 indicates that the requirements management tool was not used effectively early in the project lifecycle. Second, there seems to be a mix-up in the test priorities by the implementer. The

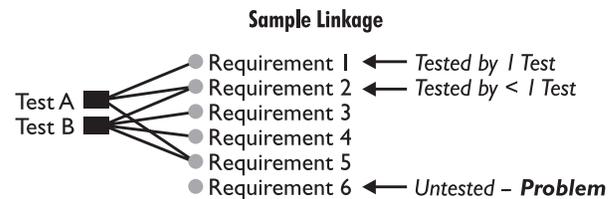


Figure 3. Requirements verification – trace to test linkage.

test program for Build 3 is farther along than that for Build 2, even though Build 2 will be developed and tested before Build 3. Resources may have been inappropriately allocated to the development of the test program for Build 2. Last, the test program for the Level 4 requirements is behind that for the test program for the Level 3 requirements. Again, this is backward. The first tests to be executed should be those for the Level 4 requirements—the system tests—and after that, tests for the Level 3 requirements—the acceptance tests—should be executed.

Requirements Test Cases

Not only is it important to understand whether all the requirements are linked to test cases, the character of the test program also needs to be understood. This can be done by looking at the profile and relationship of requirements to test cases. Figure 5 shows an expected profile of unique requirements per test case based on data from NASA projects [5].

This profile shows the expectation that there will be a large number of requirements tested by only one test case and that there will be some requirements that will be tested by

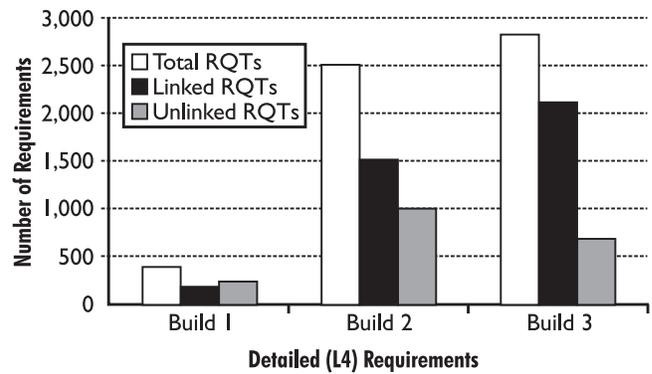
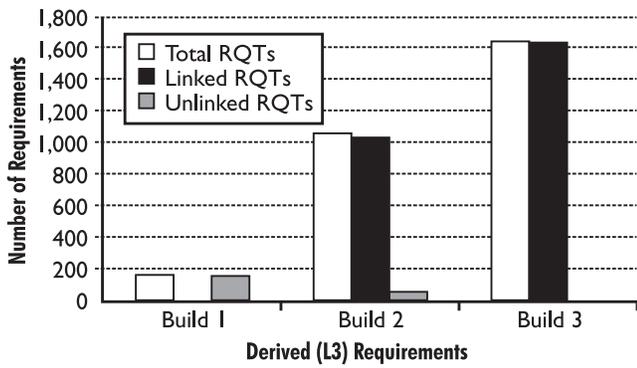


Figure 4. Requirements verification trace to test.

multiple test cases. It is expected that the upper bound of multiple test cases will range in the double-digits because more complicated requirements may require different test cases to thoroughly verify all aspects of the requirements. However, there is a logistical limit on the number of test

number of requirements are covered by just one test, which makes for a simple, easy-to-evaluate test program for a significant part of the system requirements. However, in several instances for both Build 2 and Build 3, there are several tests for unique requirements. Notice that for Build 2, one requirement has been linked to 25 test cases, and in Build 3, that same requirement is linked to 51 test cases. This large number of test cases may well make it impossible to verify that these requirements have been implemented.

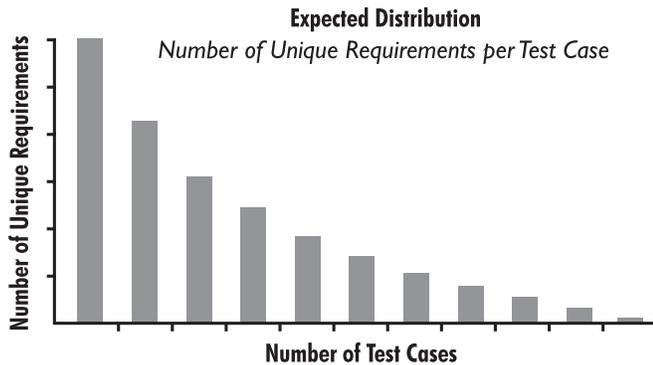


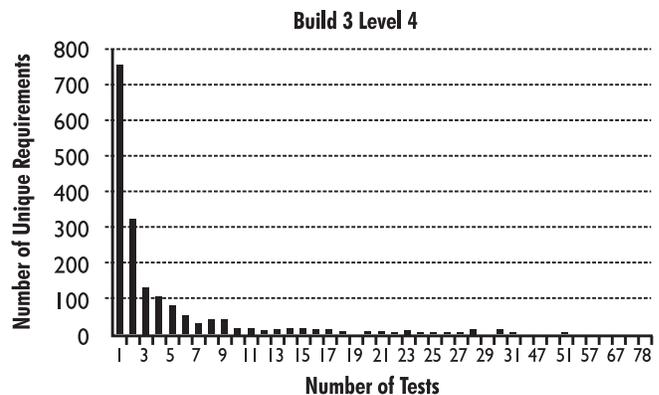
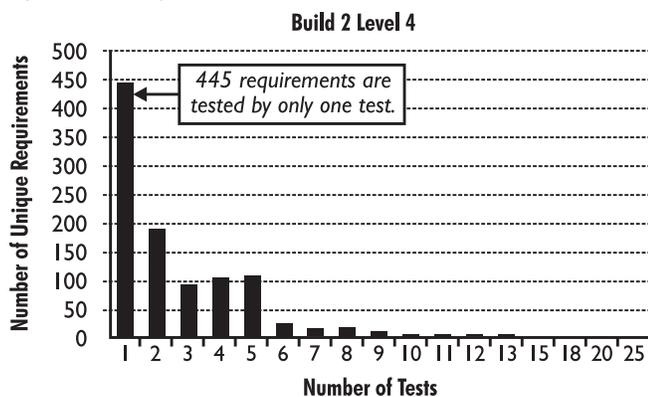
Figure 5. Test program characterization tests per requirement. Some requirements will be tested only once or can be group tested. Complex requirements need multiple tests.

Requirements Management Tools

The use of tools to aid in requirements management has become an important aspect of system engineering and design because of the size and complexity of development efforts. The tools that requirements managers use for automating the requirements engineering process have reduced the drudgery in maintaining a project's requirements set and added the benefit of significant error reduction. Tools also provide capabilities far beyond those obtained from text-based maintenance and processing of requirements. Requirements management tools are sophisticated and complex—the nature of the material for which they are responsible is finely detailed, time-sensitive, highly internally dependent, and can be continuously changing. Tools that simplify complex tasks require skill and a thorough understanding of their capabilities if they are to perform effectively over the lifetime of a project [6].

cases that can be performed; as the number of test cases increases, the difficulty in verifying the requirements increases due to the complication in data analysis, understanding the results of the multiple tests cases, and understanding the impact of multiple test case results on the verification of the requirements. Figure 6 shows the requirements-to-test-case profile for Project X. There is a good indication that a large

Figure 6. Test program characterization tests per requirement.



There are many requirements management tools from which to choose. These range from simple word processors to spreadsheets to relational databases to tools designed specifically for the management of requirements, such as DOORS (Quality Systems & Software, Mount Arlington, N.J.) or RTM: Requirements Traceability Management (Integrated Chipware, Inc., Reston, Va.). The key to selecting the appropriate tool is the functionality provided and the capability to develop metrics from the data.

The metric capability of the tool is important. It should be noted that most of the metrics presented in this article were developed from the data contained in a requirements management tool. Table 2 shows a comparison of the metric capability associated with the various tools. Clearly, the relational database and requirements management tool provide the capabilities needed to effectively support requirements management.

Tool selection is only part of the equation. A thorough understanding of the tool's capabilities and the management processes that will use the tool also is necessary. The tool should not be plugged into the management processes with no thought to the impact on the tool's capabilities. Adjustments may be needed in the management processes and employment of the tool to bring about an efficient requirements management process. Briefly, Project X had the following problems with the requirements management tool.

Project X's focus on establishing a requirements management process was influenced by project organization. The way the project chose to use the tool appeared reasonable on the surface but was fraught with flaws stemming from inexperience, and ultimately it worked against clear management. Specifically, many classes (tables or relations) mirrored organizational structure instead of a single class existing for each development phase. With a multiple test class and requirements class approach, there was a natural tendency for the organizations to "improve" the data schema definitions assigned to them. The result was losses in data integrity and restricted access to important information about the requirements. Some information that should have been available to all project organizations became specific to a particular organization [6].

Because multiple classes were implemented at the test-by-build level, fields were duplicated to each of the test classes; common information then became self-contained within each class. However, confusion developed between the test organizations as to which one was responsible for populating common data, all of which lead to inconsistent data entries and prevented effective data mining [7]. Also, due to the multiple-class approach, links that traced requirements to tests also became extensive and conflicting. Because the project decided to organize the database schema along the lines of the organization, it was necessary to provide the traceability of requirements to requirements and test case to requirements by connections between many classes. This resulted in a complex, undocumentable traceability relationship between the system test cases and the two levels of requirements. Most requirements tools are designed to use

	Word Processor	Spreadsheet	Relational Database	Requirement Tool
Document Size	X			
Dynamic Changes Over Time				X
Release Size	X	X	X	X
Requirement Expansion Profile			X	X
Requirement Types	X	X	X	X
Requirement Verification			X	X
Requirement Volatility	X	X	X	X
Test Coverage			X	X
Test Span			X	X
Test Types	X	X	X	X

Table 2. Requirement repository metric capabilities

minimal classes and effect decomposition within a class, not between classes [6].

Conclusion

To do requirements right the first time, the following components must be present: quality documentation, a complete and appropriately structured verification program, and effective requirements management. Quality documentation is complete, clear, and concise—concepts that used to be considered ethereal and difficult to measure or visualize. Now, with the advent of tools like ARM, metrics can be developed to show the strengths and weaknesses of the requirements documentation. The completeness of the verification program used to be the only aspect that was easily understood. Now, through the use of metrics, project workers not only can gain insight into the completeness of the test program but also can understand the overall characteristics of the verification program. Effective requirements management now demands the appropriate use of management tools or databases or both through the development lifecycle. Through their use, the development of metrics to gain insight into the nature of the requirements is enabled. Metrics provide a powerful tool to gain insight into each of these areas and give the project the ability to get the requirements right the first time. It is no longer a dream but a reality. ♦

About the Authors

Theodore F. Hammer is the NASA manager for the SATC at NASA's GSFC. He is responsible for managing software quality assurance activities for selected spacecraft implementation projects. Prior to this position, he was a member of the Assur-

ance Management Office, where he was responsible for managing the overall quality assurance activities for specific ground system implementation projects, with special emphasis on software quality assurance. He has more than 22 years experience in software development and assurance. He joined NASA GSFC in 1989, where he supported NASA Headquarters Software Management Assurance Program and participated in the review of the early versions of the military software development standard, MIL-STD-498, as well as NASA software development and assurance standards and guidebooks. He has a bachelor's degree in electrical engineering from the University of Maryland and is a member of the American Society for Quality.

Goddard Space Flight Center
Code 302
Greenbelt, MD 20771
Voice: 301-286-7475
Fax: 301-286-1701
E-mail: thammer@pop300.gsfc.nasa.gov

Lenore L. Huffman is a principal engineer with SATC. She has more than 14 years software engineering and quality assurance experience. She is expert in the design, implementation, and execution of data collection, database structures, and metrics reporting and analysis. She also is expert in the design and use of state-of-the-art database reporting systems. She has extensive experience automating configuration management and problem reporting systems and adapting their capabilities to satisfy unique project requirements. She has successfully planned, designed, and implemented software quality assurance projects. Prior to joining the SATC, she developed met-

rics for software at the Space Telescope Institute, and while working at a chemical research center, was awarded several U.S. patents. She has a master's degree in business administration.

Goddard Space Flight Center
Code 300.1, Building 6
Greenbelt, MD 20771
Voice: 301-286-0099
E-mail: Lenore.L.Huffman.1@gsfc.nasa.gov

Linda H. Rosenberg is an engineering section head at Unisys Government Systems in Lanham, Md. She is contracted to manage the SATC through the System Reliability and Safety Office in the Flight Assurance Division at NASA GSFC. She is responsible for risk management training at all NASA centers, and the initiation of software risk management at NASA GSFC. As part of the SATC outreach program, she has presented metrics and quality assurance papers and tutorials at GSFC, the Institute of Electrical and Electronic Engineers (IEEE), and the Association for Computing Machinery (ACM) local and international conferences. She also reviews for ACM, IEEE, and military conferences and journals. She holds a doctorate in computer science from the University of Maryland, a Master's of Engineering Science in computer science from Loyola College, and a bachelor's degree in mathematics from Towson State University. She is a member of IEEE, the IEEE Computer Society, ACM, and Upsilon Pi Epsilon.

Goddard Space Flight Center
Code 300.1, Building 6
Greenbelt, MD 20771
Voice: 301-286-0087
E-mail: Linda.H.Rosenberg.1@gsfc.nasa.gov

References

1. Brooks Jr., Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, Vol. 15, No. 1, April 1987, pp. 10-18.
2. Hammer, T., L. Huffman, L. Rosenberg, W. Wilson, L. Hyatt, "Requirements Metrics for Risk Identification," Software Engineering Laboratory Workshop, Goddard Space Flight Center, December 1996.
3. NASA, *Software Assurance Guidebook*, NASA Goddard Space Flight Center Office of Safety, Reliability, Maintainability, and Quality Assurance, September 1989.
4. Wilson, W., L. Rosenberg, and L. Hyatt, "Automated Analysis of Requirements Specifications," Fourteenth Annual Pacific Northwest Software Quality Conference, October 1996.
5. Hammer, T., "Measuring Requirements Testing," Eighteenth International Conference on Software Engineering, May 1997.
6. Hammer, T., "Automated Requirements Management – Beware How You Use Tools," Nineteenth International Conference on Software Engineering, April 1998.
7. Chen, M., J. Han, and P. Yu, "Data Mining: An Overview from a Database Perspective," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, December 1996.

Notes

1. Various names are used—deliveries, releases, builds—but the term *build* is used in this article.
2. This tool is available at no cost from the SATC Web site <http://satc.gsfc.nasa.gov>.



Impact Estimation Tables

Understanding Complex Technology Quantitatively

Tom Gilb
Independent Consultant

How good is your design suggestion? Does anybody else understand why you think the technology you suggest is such a great idea? Would you like to know how to shoot down those dumb ideas that consultants and your colleagues use to entice your managers? Would you like a great approach to prove your technical expertise to the world? We may have it right here.

Impact Estimation (IE) tables allow you to analyze any technical or organizational idea in relation to requirements and costs. It is a method I have developed over the last 20 years, and it works! To give one example, shortly after we taught the idea to a manufacturing group, they declared it was worth a million dollars. Using IE for the first time, they presented a bid for project money to management and got the full budget they requested—\$1 million more than they had expected!

Aims of IE

IE can be used for a wide variety of purposes [1]. Its most important uses include

- Comparing alternative design ideas.
- Estimating the state of the overall design architecture.
- Planning and controlling evolutionary project delivery steps.
- Analyzing risk.

I use IE tables when evaluating projects to help answer my “Twelve Tough Questions.”

- Why is the improvement not quantified?
- What is the degree of the risk or uncertainty and why?
- Are you sure? If not, why not?
- Where did you get that information? How can I check it out?
- How does your idea measurably affect my goals?
- Did we forget anything critical to survival?
- How do you know it works that way? Did it before?
- Have we got a complete solution? Are all objectives satisfied?
- Are we planning to do the “profitable things” first?
- Who is responsible for failure or success?
- How can we be sure the plan is working during the project—early?
- Is it “no cure, no pay” in a contract? Why not?

A More Quantitative Approach

The basic IE idea is simple: Estimate quantitatively how much your design ideas impact all critical requirements. As simple as this is, software engineers do not normally do it. We judge too narrowly. We only treat the costs, e.g., development costs and operational costs, quantitatively. The qualities, e.g., usability, system availability, and system flexibility, tend to be handled subjectively. There are two important underlying issues here. First, we need to express our requirements in a quantitative manner. Second, we need to gather objective data about our technologies. We can make a start by making use of practical experience data.

How to Quantify and Document the Relationship Between Requirements and Design

First, I will show how to express the relationships between the system requirements and your new design ideas using IE. Later, I will show how this analysis and presentation discipline can be used to analyze many design ideas (the overall system architecture) in relation to all system requirements.

IE focuses on the system qualities and costs. It is a question of how a design idea impacts all the critical *qualities*, e.g., performance, usability, and reliability, and how it impacts all the costs (money, time, people, and space) to build, deliver, and maintain the design idea. A design idea is “effective” to the degree it satisfies specified requirement levels of qualities. A design idea is cost-effective (“efficient”) to the degree it is effective in relation to all costs.

Note: IE does not consider functions. The quality and cost requirements for the system are the prime considerations. Design ideas are evaluated as to how “good” and cost-effective they will be at delivering the qualities required. For example, when considering how to transport a person from point A to point B, it is the quality and cost requirements (such as flexibility of travel times, safety, reliability, comfort, and price) that select the best design idea (such as airplane, foot, rocket, or ambulance).

A Simple Relationship Between Requirements and Design Ideas

I will start simple and expand scope later. Assume I have a requirement (a constraint) that my budget not exceed \$100,000. Also assume that I have a design idea—I will call it

Table 1. A simple IE table.

Design Idea: Big Idea	Real Impact	%IMPACT
Quality = Reliability Plan = 1,000 hours MTBF by end of December 1999	1,000 hours MTBF	100 percent
Cost = Development Budget Plan = \$100,000 up to end of December 1999	\$10,000	10 percent

Credibility Rating	Meaning
0.0	Wild guess, no credibility.
0.1	We know it has been done somewhere.
0.2	We have one measurement somewhere.
0.3	There are several measurements in the estimated range.
0.4	The measurements are relevant to our case.
0.5	The method of measurement is considered reliable.
0.6	We have used the method in-house.
0.7	We have reliable measurements in-house.
0.8	Reliable in-house measurements correlate to independent external measurements.
0.9	We have used the idea on this project and measured it.
1.0	Perfect credibility, we have rock solid, contract-guaranteed, long-term, credible experience with this idea on this project, and the results are unlikely to disappear.

Table 2. An example of how credibility ratings can be assigned.

Big Idea—that I estimate will cost \$10,000 (10 percent of my budget).

I also have a quality requirement—I will call it *Reliability*—which is to reach mean time between failures (MTBF) of 1,000 hours by the end of December 1999. If I believe that *Big Idea* will reach 1,000 hours MTBF within the time scales, it satisfies my requirement for *Reliability* completely (100 percent). You can express this with a simple IE table (Table 1).

Further Improvements to Specifying the Relationship

There are a number of improvements to this basic idea, which make it more communicative and

credible. Following is a brief summary of them.

Impact Relative to a Defined Baseline

For all qualities, it is essential to define a baseline. Usually, the current value achieved by the “old” system is used. For example, if “900 hours MTBF” was the level for reliability of our previous system, we could use that as a reference base. This can be stated using Planguage, my Requirements Planning Language [1], as PAST [September 1997] 900 hours MTBF. It is the minimum level or 0 percent level because only when the system’s MTBF is higher than 900 hours will any progress have been made on improving reliability.

The planned level by December 1999 is 1,000 hours MTBF. Using Planguage, this is expressed as PLAN [December 1999] 1,000 hours MTBF. This is the 100 percent level. When we have improved the MTBF to this level, we shall have completely met our reliability requirement. (Exceeding a requirement is fine as long as we have not incurred additional costs or failed to divert resources that could have been expended on achieving other requirements.)

If the impact of *Big Idea* was estimated at 900 hours, we would be making no forward progress toward our planned level. In other words, the percentage impact of *Big Idea* (from base to plan) is 0 percent. It is unlikely that anyone needs a new idea to get to where they are already.

However, if the impact of *Big Idea* was estimated at 950 hours MTBF, it is a much more interesting design idea. Its percentage impact is 50 percent, halfway between the base of 900 hours (0 percent) and the goal of 1,000 hours (100 percent).

Note, the percentage impact on plan (%IMPACT) can only be estimated if there is both a baseline and a planned

level. %IMPACT is useful because it enables us to “add” the percentage impacts from different scales-of-measure, as will be explained later.

Uncertainty of Impact

All estimates are uncertain. It is useful to estimate how uncertain they are. This helps you understand the risk of not meeting desired goals. So if the uncertainty for the estimate of 950 hours MTBF was plus or minus 10 hours, our estimate of 950 hours becomes 950+/-10, which could be expressed alternatively as a percentage impact of 50 +/-10 percent. The negative number (-10 percent) can be used to modify estimates to discover the worst-case situation.

Evidence for Impact Assertion. If you want any credibility for your assertions, you should be prepared to supply facts to back them up. Instead of waiting to be asked, dig up the facts and document them in advance. This shocks people—they are not accustomed to being offered facts. For example, “*Big Idea* was used for 10 projects last year in our company, and the range of MTBF attributed to it was 940 to 960 hours MTBF, average 950.”

Source of Evidence for Impact Assertion. Of course, some skeptics might like to verify your assertion and evidence, so you should give them a source reference, e.g., “Company Research Report TR-017, pp. 23-24.”

Credibility Rating of the Impact Assertion

We have found it extremely useful (it was a key part of getting the \$1 million mentioned earlier) to establish a numeric *credibility* for an estimate, based on the credibility of the evidence and the source. We use a scale of 0.0 to 1.0 because it can then be used later to modify estimates in a conservative direction. See Table 2 for an example of how credibility ratings can be assigned.

Table 3. Example: The set of data for a single-cell estimate of the impact of *Big Idea* on reliability.

Requirement: Past → Plan	Real Impact (Estimate on real scale for <i>Big Idea</i>)	Relative to plan (and base estimate)	Uncertainty Estimate	Evidence	Source	Credibility
Reliability: 900 → 1,000 hours MTBF	950	50 percent	+/- 10%	Project A 940 hours MTBF	TR-017	0.6

The credibility rating is useful because it forces you to analyze, gather data, and raise the credibility to an acceptable level. Normally, people make no effort here.

Further Analysis of the IE Data

Now assume you have numerous critical qualities and that you have completed an IE table using several design ideas. There are now a number of calculations using the %Impact estimates you can do to help understand the robustness of your proposed solution.

I stress that these are only rough, practical calculations. Adding impacts of different independent estimates for different design ideas that are part of the same overall architecture is dubious in terms of accuracy. But as long as this is understood, you will find them extremely powerful when considering such matters as whether a specific quality goal is likely to be met or which is the most effective design idea. The insights gained are frequently of use in generating new design ideas.

I add an additional cautionary note: I expect this information to only be used as a rough indicator to help designers spot potential problems or select design ideas. Any real estimation of the impact of many design ideas needs to be made by real tests; ideally, by measuring the results of early evolutionary steps in the field.

Impact on Quality

For each individual quality or cost, sum all the percentage impacts for the different design ideas. This gives us an understanding of whether you are likely to make the planned level for each quality or cost. Extremely small quality impact sums like 4 percent indicate high risk that the architecture is probably not capable of meeting the goals. Large numbers like 400 percent indicate you might have enough design or even a "safety margin."

Impact of a Design Idea

For each individual design idea, sum all the percentage impacts it achieves across all the qualities to get an estimate of its overall effectiveness in delivering the qualities. The resulting estimates can be used to help select among the design ideas. It is a case of selecting the design idea with the highest estimate value

Table 4. Example: Adding the percentage impacts for a set of design ideas on a single quality or cost can give some impression of how the designs are contributing overall to the project goals. Note: Design Ideas A, B, and C are independent and complementary.

Quality	Past → Plan	Big Idea
Reliability	900 → 1,000 hours MTBF	50% +/- 10%
Maintainability	10 min. to fix → 5 min. to fix	100% +/- 50%
Estimate of effect of Big Idea on all goals		150% +/- 60%
Averaged Estimate		75% +/- 30%

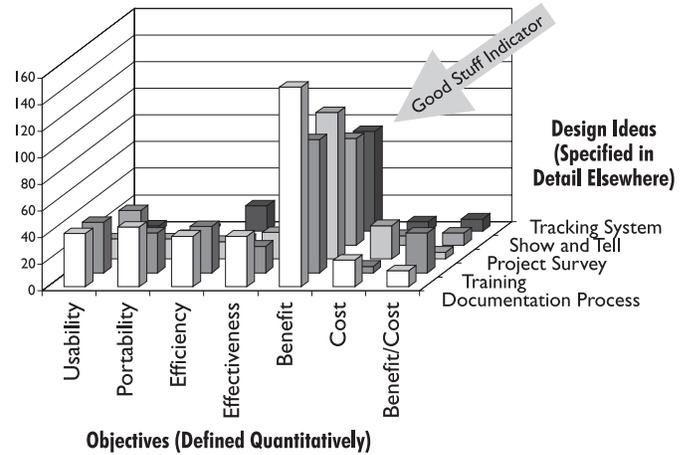


Figure 1. A 3-D representation of an IE table. "Benefit" shows the summed total of all the %Impacts for the qualities. Note: The design idea (Documentation Process) that contributes most toward meeting the requirements is shown to not be the most cost-effective. Note also, the summed total for each of the objectives, i.e., the totals if all the design ideas were implemented, has not been calculated.

and the best fit across all the critical quality requirements. If the design ideas are complementary, the aim is to choose which design idea(s) to implement first. If the design ideas are alternatives, you are merely looking to determine which one to pick.

These estimates are something like universal currency. Each estimate tells how well you are moving toward your goals. It allows you to get some impression of the overall impact of a single design idea. For example, if a design idea has 50 percent impact on each of the different goals, I might console myself by rationalizing that I have designed enough to get halfway to my goals.

In addition to looking at the effectiveness of the individual design ideas in impacting the qualities, the cost of the individual design ideas also needs to be considered, as will be shown in the next section.

Quality-to-Cost Ratio

For each individual design idea, calculate the quality-to-cost ratio, also known as the benefit-to-cost ratio. For quality, use the estimate calculated in the previous section. For cost, use the percentage drain on the overall budget of the design idea or use the actual cost.

The overall cost figure used should take into account both the cost to develop or acquire the design idea and the cost of operationally running the design idea over the chosen time scale. Sometimes, specific aspects of resource utilization also need to be taken into account. For example, maybe staff utilization is a critical factor; therefore, a design idea that does not use scarce programming skills becomes much more attractive.

My experience is that a comparison of the impact vs. the cost of design ideas often wakes people up dramatically to ideas they have previously undervalued or overvalued.



This article can be found in its entirety on the Software Technology Support Center Web site at <http://www.stsc.hill.af.mil/CrossTalk/crostalk.html>. Go to the "Web Addition" section of the table of contents.

National Software Quality Experiment A Lesson in Measurement: 1992-1997

Don O'Neill, *Independent Consultant*

The nation's prosperity is dependent on software. The National Software Quality Experiment is riveting attention on software product quality and revealing the patterns of neglect in the nation's software infrastructure. In 1992, the Department of Defense Software Technology Strategy set the objective to reduce software problem rates by a factor of 10 by the year 2000. The National Software Quality Experiment is being conducted to benchmark the state of software product quality and to measure progress toward the national objective. Motivation for the experiment, methods used to collect data, and an analysis of the findings are presented.

Average Credibility and Risk Analysis

Once you have all credibility data, i.e., the credibilities for all the estimates of the impacts of all the design ideas on all the qualities, you can calculate the average credibility of each design idea and the average credibility of achieving each quality. This information is powerful because it helps you understand the risk involved; for example, "the average credibility, quality controlled, for this alternative design idea is 0.8." Sounds good. This approach also saves meeting time for those who hold the purse strings.

As long as you do not get carried away and attempt too many calculations, it can help your understanding of the risks involved if you modify your estimates using the worst-case error or credibility ratings. Altering an estimate to its worst-case value is a matter of subtracting or adding the worst-case error correction to the estimate. To modify an estimate to take into account its credibility, multiply the estimate by its credibility rating. Multiplying a worst-case estimate by its credibility rating will give you the most pessimistic value. Once you have reworked the estimates, you can then recalculate the totals.

Conclusion

If you want to move software engineering toward "real" engineering and toward better control over your results, introduce systematic and fact-based thinking into software development.

	Design Idea A	Design Idea B	Design Idea C	Sum of Impacts	Sum Uncertainty
Reliability 900 → 1,000 hours MTBF	0 +/- 10%	10 +/- 20%	50 +/- 40%	60%	+/- 70%

Table 5. A measure of the effectiveness of Big Idea can be found by adding together its percentage impacts across all the qualities.

Selective use of IE tables by project management would greatly assist communication with senior management and improve risk control.

Acknowledgments

I thank Barry Boehm of the University of Southern California, who at the 1974 International Federation of Information Processing Societies Conference in Stockholm showed me his Requirements/Properties Matrix, a precursor to Quality Function Deployment, which challenged me to provide the quantification aspects of IE tables. He has always provided an appreciative audience to my evolution of the method and is an inspiration to us all. I also thank Steve McConnell for his initial suggestion, help, and comments on this article and Lindsey Brodie for editing this article. ♦

About the Author



Tom Gilb immigrated from California to Europe in 1956. In 1958, he began working for IBM in Norway, and five years later became a

freelance consultant. He is author of several books, including *Software Metrics* (1976-77), *Software Inspection* (1993), and *Principles of Software Engineering Management* (1998). He spends about half his time working in the United States and half in Europe.

Internet: <http://www.result-planning.com>
E-mail: Gilb@ACM.org

Reference

1. Gilb, Tom, "Requirements-Driven Management Using Planguage," http://www.stsc.hill.af.mil/SW_Testing/gilb.html, <http://www.result-planning.com>, 1995-1996.

Recommended Reading

1. Akao, Yoji, ed., *Quality Function Deployment: Integrating Customer Requirements into Product Design*, Productivity Press, Cambridge, Mass., 1990.
2. Gilb, Tom, *EVO: The Evolutionary Project Manager's Handbook*, <http://www.result-planning.com>.
3. Gilb, Tom, *Principles of Software Engineering Management*, Addison-Wesley Longman, Boston, Mass., 1988.



The Ada Recommendation – Was It Heard?

Rayford B. Vaughn Jr.
Mississippi State University

In January 1997, the National Research Council published a report to the Department of Defense (DoD) entitled Ada and Beyond, Software Policies for the Department of Defense. The author was a member of the committee that produced the report. This article looks at the original recommendations of the report, the process used to produce it, and comments on both intended and unintended results.

Early in 1996, the National Research Council's Computer Science and Telecommunications Board (CSTB), began to search for committee members to serve on a committee chartered to perform a "Review of the Past and Present Contexts for using Ada within the Department of Defense." Twelve people were selected: Barry Boehm (committee chairman), Theodore Baker, Wesley Embry, Joseph Fox, Paul Hilfinger, Maretta Holden, J. Eliot, B. Moss, Walker Royce, William L. Scherlis, S. Tucker Taft, Anthony Wasserman, and me. Paul Semenza, National Research Council (NRC), was assigned to guide the committee, provide administrative support, and to serve as interface between the committee and DoD. This article outlines the deliberations of the committee, the final recommendations, and comments on what has happened since publication of the final report, entitled *Ada and Beyond: Software Policies for the Department of Defense* [1]. It represents my impressions and thoughts and is not an official opinion of the NRC, the committee members, or DoD.

The committee met in a three-day group session four times during April 1996 to October 1996: twice in Washington, D.C., once in Denver, and once in Los Angeles. Significant work was accomplished between meetings, and the committee constantly communicated electronically. During the first session, the committee agreed that the issue was far larger than just a "language decision" and needed to be taken in the larger context of DoD software engineering. It was also determined that the committee needed to hear from the software development community outside DoD. The committee discovered, through DoD

speakers, that support for the Ada Joint Program Office (AJPO) was being dropped—a major concern that resulted in a finding that was considered critical to the future of the language.

Findings and Recommendations

The majority of information in this section was taken from the final report [1]. I provide editorial comments after each rationale to add to background, better understanding of the issues, or some of the influences that were present.

Finding 1: Ada Competitive Advantage

- **Finding:** Ada provides DoD with a competitive advantage for war-fighting software applications, including weapons control, electronic warfare, performance-critical surveillance, and battle management.
- **Recommendation:** Continue vigorous promotion of Ada in war-fighting application areas.
- **Rationale:** Available project data and analyses of programming language features indicate that compared with other programming languages, Ada provides DoD with higher-quality war-fighting software at a lower life-cycle cost. DoD can create a further competitive advantage by strengthening its Ada-based production factors (involving software tools, technology, and personnel) for war-fighting software.

It was understood that there was no clear definition of "war-fighting" software. Clearly, some systems are, e.g., embedded weapons systems, such as cruise missile or AEGIS guidance, and others are arguable, e.g., personnel and logistics, in a support role. Essentially,

there can never be a concise definition of war-fighting software. If ample commercial-off-the-shelf (COTS) products exist that can provide the needed functionality, the system under development is probably not "war fighting" in the context of this report. Likewise, if the system requires software quality and reliability attributes higher than supportable by commercial products, the system may be categorized as war fighting.

Finding 2: Applicability of Policy to DoD Domains

- **Finding:** DoD's current requirement for use of Ada is overly broad in its application to all DoD-maintained software.
- **Recommendation:** Focus the Ada requirement on war-fighting applications, particularly critical, real-time applications in which Ada has demonstrated success. For commercially dominated applications, such as office and management support, routine operations support, asset monitoring, logistics, and medicine, the option of using Ada should be analyzed but should not be assumed to be preferable.
- **Rationale:** For war-fighting software, supporting Ada-based production factors (involving software tools, technology, and personnel) gives DoD a competitive advantage. In this domain, eliminating the use of Ada would both compromise this advantage and diminish the capabilities for maintaining DoD's existing 50 million lines of Ada. In commercially dominated areas, pushing applications toward Ada would create a competitive disadvantage for DoD. Early in the deliberations, the committee discussed extensively the total

elimination of the mandate, particularly if DoD was intent on dropping all support for the AJPO. There were many instances of overapplication of the mandate within DoD where COTS products were bypassed in favor of more expensive Ada development for no reason other than the mandate. Subsystems were rarely considered in the context of the Ada mandate. Given that DoD would continue its support for the AJPO and that there is a stronger Ada production base here in the United States, the committee felt that an advantage accrued if war-fighting software continued to be developed in Ada.

Finding 3: Scope of Policy

- **Finding:** DoD's current requirement for the use of Ada overemphasizes programming language considerations.
- **Recommendation:** Broaden the current policy to integrate choice of programming language with other key software engineering concerns, such as software requirements, architecture, process, and quality factors.
- **Rationale:** The current policy isolates the Ada requirement and waiver process from other software engineering processes, causing programs to make premature or nonoptimal decisions. DoD has already taken steps to broaden the policy focus in its draft revision of its programming language policy (DoD Directive 3405.1).

The committee was given a draft DoD Directive 3405.1 that moved closer to a software engineering focus vs. a language-only focus. This draft was then modified by the committee and provides an appendix to the final report.

Finding 4: Policy Implementation

- **Finding:** DoD's current Ada requirement and the related waiver process have been weakly implemented. Many programs have simply ignored the waiver process. Other programs make programming-language decisions at the system level, but often a mix of Ada and non-Ada subsystems is more appropriate.
- **Recommendation:** Integrate the Ada decision process with an overall Soft-

ware Engineering Plan Review (SEPR) process. To pass such a review should be a requirement to enter the system acquisition Milestone I and II reviews covered by DoD Instruction 5000.2. It should also be required for systems not covered in DoD Instruction 5000.2 and recommended by DoD for DoD-directed software development and maintenance of all kinds.

- **Rationale:** The SEPR concept is based on the highly successful commercial architecture review board practice. The SEPR process involves peer-reviewing not only the software and system development plans but also the software and system architecture (building plan) and its ability to satisfy mission requirements, operational concepts, conformance with architectural frameworks, and budget and schedule constraints; the process also involves reviewing other key decisions such as choice of programming language.

A key concern here was the "ability" of DoD to put individuals with good software engineering backgrounds on the review boards. The review can be a powerful tool and can enforce architectural frameworks developed by the services or DoD if staffed with the right people. They can also be the "common sense" sounding board that a program manager needs when trying to make good technical and cost-effective decisions.

Finding 5: Investment in Ada

- **Finding:** In order for Ada to remain the strongest programming language for war-fighting software, DoD must provide technology and infrastructure support.
- **Recommendation:** Invest in a significant level of support for Ada or drop the Ada requirement. The strategy developed by the committee recommends an investment level of approximately \$15 million per year.
- **Rationale:** With investment, DoD can create a significant Ada-based complex of production factors (involving software tools, technology, and personnel) for war-fighting application domains. Without such

support, Ada will become a second-tier niche language such as Jovial or CMS-2.

There was strong concern voiced by all committee members when it was discovered that DoD planned to drop support for Ada. Essentially, it was felt that Ada was not strong enough to stand on its own today and that the support was fundamental to its success. Great improvement had been made over the years in Ada and its support environment, and this investment would be placed at risk without continued DoD support.

Finding 6: Software Metrics Data

- **Finding:** DoD's incomplete and incommensurable base of software metrics data weakens its ability to make effective software policy, management, and technical decisions.
- **Recommendation:** Establish a sustained commitment to collect and analyze consistent software metrics data.
- **Rationale:** The five sets of findings and recommendations above are based on a mix of incomplete and incommensurable data, anecdotal evidence, and expert judgment. For this study, the patterns of consistency in these sources of evidence provide reasonable support for the results but not as much as could be provided by quantitative analysis based on solid data. A few organizations within DoD have benefited significantly from efforts to provide a sound basis for software metrics; a DoD-wide data collection effort would magnify the net benefits.

The committee found it extremely hard to find data to support any of the testimony to which we were exposed. In fact, it seems to be a systemic problem within DoD that metrics are not heavily supported and collected for review at service level or DoD level.

Reaction and Response

Although the majority of conversation and interest in the committee's recommendations has centered on whether Ada should be mandated for all system development, the report clearly goes beyond

the question of Ada and proposes that the programming language should not be the focus of concern, but software engineering practices should be. Additionally, DoD may have, over the past year, undermined the intent and recommendation of the committee in multiple areas that could put the language at risk as well as the significant investment that has been made over many years by DoD and the commercial community.

The competitive advantage that Ada gave the war-fighting community (Finding 1) was explained within the report in great detail. A key component of maintaining this advantage was continued support from a policy and financial standpoint. DoD has chosen not to invest in Ada through continued support of the AJPO (Finding 5) and not to support Finding 2, which mandated Ada for war-fighting software. The lack of support for these two essential findings thus results in an indirect lack of support for Finding 1 and its associated recommendation.

DoD's failure to support Finding 2—to mandate Ada for war-fighting software but not for commercially dominated software domains—has created concern within the Ada community; in some cases, the NRC report is incorrectly cited as the catalyst for this decision, though it clearly did not recommend such an action for reasons cited in Chapter 3 of [1].

Finding 3 appears to have been partially accepted (in principal), and DoD seems to be moving toward a process that adopts more focus on software engineering decisions vs. language decisions. Little consideration, however, is being given to changes in the system review process and the adoption of the recommended software plan review. This particular finding was important; its adoption would bring DoD more in line with current accepted commercial practice.

Finding 4 is essentially moot with the dropping of the Ada mandate. Its associated recommendation, however, contained a description of a commercial architecture review board process that was deemed necessary by committee

members. It appears that this recommendation will not be adopted by DoD.

Finding 5 was critical to the future of Ada in DoD and required strong financial backing and support for the AJPO. This finding and its associated recommendation met with strong internal opposition within DoD and in particular, disagreement between the Defense Information Systems Agency (DISA) and the Office of the Assistant Secretary of Defense for Command, Control, Communications, and Intelligence (OASD C3I). OASD C3I directed that DISA comply with the recommendation but did not seek any additional funding for DISA to do so. DISA, in a letter to the committee, outlined its position that "the AJPO was created to do a job; it has succeeded, and is no longer necessary." The disagreement between these two agencies was never resolved, the AJPO funding was not substantially increased, the AJPO director position was not filled, and the office's ability to support the Ada program became severely impaired. This course of events was consistent with the stated position of DISA in which the committee was told, "We have not determined a final date for the closing of the AJPO. We selected a date in late third quarter of fiscal 1997 as a target for planning purposes."

Finding 6 continues to be an issue today but has not been acted on in a way that makes the situation then any different today. Metrics gathering and reporting is still a problem within DoD and needs to be addressed.

Summary

It should be clear that a year after the report was released, most of the recommendations and findings have not been followed. There are earlier reports that indicate DoD adopted all but one recommendation, e.g., [3], but as can be seen from the above, little was adopted. The process changes recommended by the report seem to be under careful study by the Office of the Secretary of Defense, but the overall intention of the report was not accepted. The recommendations were meant to work together as a

holistic approach to improve the software development process in DoD. A piecemeal adoption may do more harm than good. As a war-fighting language and a national competitive advantage, Ada would have to be considered in jeopardy at the current time. ♦

About the Author



Rayford B. Vaughn Jr. spent 26 years in the U.S. Army as a software engineer. His key assignments included commander of the Army's Information

Systems Software Center headquartered at Fort Belvoir, Va. and the first director of the Pentagon Single Agency Manager for Information Technology Services. Upon retirement as a colonel in May 1995, he joined Electronic Data Systems (EDS) Military Systems where he served as vice president of DISA Integration Services. While with EDS, he was responsible for all EDS contracts issued by DISA. In October 1997, he accepted a position as associate professor of computer science at Mississippi State University.

Department of Computer Science
P.O. Box 9637
Mississippi State, MS 39762
Voice: 601-325-7450
Fax: 601-325-8997
E-mail: vaughn@cs.msstate.edu
Internet: <http://www.cs.msstate.edu>

References

1. Computer Science and Telecommunications Board, National Research Council, *Ada and Beyond, Software Policies for the Department of Defense*, National Academy Press, Washington, D.C., 1997.
2. Vaughn, Rayford, "The National Research Council's Ada Mandate Study: Insights and Recommendations," *Proceedings of the DoD Software Technology Conference*, 1997.
3. Hamilton, J.A., "Why Programming Languages Matter," *CROSSTALK*, Software Technology Support Center, Hill Air Force Base, Utah, Vol. 10, No. 12, December 1997.

No Hypoxic Heroes, Please!

Biological Limits on Cowboy Programmers

Lewis Gray
Abelia Corporation

What do you say when someone rejects IEEE/IEA 12207, the Software Engineering Institute Capability Maturity ModelSM (CMM)[®], and every other process standard? Here is a response.

James Bach's article "The Micro-dynamics of Process Evolution" [1] moved me to write a response to the entire collection of methodological arguments it exemplifies. Along with other like-minded arguments, Bach's article promotes heroism as a substitute for process. My goal here is to point out an inherent biological limit to dependence on heroes.

Mountaineers who climb high mountains, like Mount Everest, experience an insidious debilitation called *hypoxia* that impairs judgment—even the ability to detect the impairment. Software managers and developers experience a common problem that is similar in some ways to hypoxia. It limits what heroes can be expected to do.

When managers take away process standards as development tools, they take away the very tools that developers need to cope with the problem.

Hypoxia and Stress

I have been reading a lot about Mount Everest in the last year, including Jon Krakauer's *Into Thin Air*; a tragic story about the death of five people near the summit in 1996 [2]. Two of them were widely admired, professional mountaineers and guides. All of them were fit and well trained. Their exceptional drive and focus pushed them through miserable conditions all the way to the top of the highest mountain on earth. Their behavior during the climb is what most of us mean by the word heroic.

Krakauer reports that a major factor in the deaths of these five heroic people

was hypoxia (lack of oxygen). High on Mount Everest, in the death zone above 25,000 feet, the amount of oxygen in the air drops to only one third of what it is at sea level. When the human body is deprived of oxygen to this extent, it always breaks down. Even Sherpas in Nepal, for example, live well below the altitudes of the camp sites on the way to the summit. Everyone who goes to the summit of Mount Everest becomes seriously hypoxic.

Hypoxic people not only do not think well, they also do not know that they do not think well. Anticipating this problem in 1996, the guides set strict rules for how and when their group members would make their summit attempts. In effect, the rules were intended to replace judgment at the most dangerous part of the climb near the summit. One of the rules was to turn around and head back down the mountain at 2 p.m. on summit day, no matter how close to the summit anyone might be at that time.

In the everyday world, there is a common medical condition—stress—that is similar in some ways to hypoxia. Heavy stress impairs our thinking and judgment. We find that we cannot identify and weigh alternatives like we can when we are calm. As with hypoxia, under heavy stress, we often simplify our options into black-or-white problems with an obvious solution. Then, we quickly seize the solution so we can get on to the next problem. It feels right, and we feel like efficient problem solvers.

But from past experience, we all know that this approach to decision-making is seductive and defective.

Lists Are an Antidote

There is a popular antidote for poor decision-making under stress: lists. All

kinds of lists can help, from grocery lists to to-do lists. Project managers use checklists to estimate and control projects. Pilots use checklists to prepare for flight. Scuba divers use checklists before going into the water.

Everyone uses lists for the same basic reason. We all recognize that when we are preoccupied, under pressure, or distracted, we forget things and make errors in judgment. Lists are like the rules that the Mount Everest climbers imposed before their climb. We need them to simulate good judgment at certain critical times.

Many modern software engineering standards, like ISO/IEC 12207 (Information Technology – Software Life Cycle Processes), MIL-STD-498 (Software Development and Documentation), quality standards such as the ISO 9000 series, and process documents like the CMM, are just lists. Speaking as one of the designers of MIL-STD-498 and IEEE/EIA 12207, I can report that these standards were designed to be checklists of tasks to consider during software project planning.

We instinctively use lists for survival. The motivation for using standards is not just good form—not just being the best. We need standards and other lists to avoid disaster (although they may be useful for more than that).

Process Standards

In a development situation, you or I might choose not to do some task in a standard because it might not be appropriate for the project or organization. In many modern standards, the only mandatory activity is tailoring the standard to your particular needs.

Modern process standards are not designed to replace professional skill or experience in software development.

An earlier version of this article, "Gray Rebuts Bach: No Cowboy Programmers!" appeared in IEEE's Computer (April 1998), pp. 102-103, 105.

Capability Maturity Model is a service mark of Carnegie Mellon University. CMM is registered in the U.S. Patent and Trademark Office.

Pilots know how to fly before they are hired by airlines. They do not use checklists as do-it-yourself flying manuals. No one should expect standards like IEEE/EIA 12207 to be do-it-yourself software development manuals for novices. For the software professional, process standards are “pilot checklists” to get software development off the ground. The value of putting the tasks in a standard is it forces standard users to at least acknowledge, and better yet, to attempt to understand the possible negative consequences of not doing the tasks in the standard on their projects. This is the heart of the tailoring process. It is a critical part of successful project planning.

Modern process standards like IEEE/EIA 12207 are more useful today than their predecessor standards such as DOD-STD-2167A and DOD-STD-1679A. The earlier standards did not reflect current thinking that process standards can be standard checklists.

Why use a standard when you can develop your own personal checklist? One reason is that hundreds or thousands of software professionals have contributed tens of thousands of comments designed to polish a standard like MIL-STD-498. It does not seem sensible to many people to completely ignore these insights and start a list from scratch based only on personal, necessarily more-limited experiences. It is sensible to start your own list with a good standard.

Compliance with Standards

So what about a hard-hearted auditor who objects to any deletion of any requirement in a standard such as the ISO 9000 series or the CMM? Perhaps the auditor will not let you tailor the standard even though you feel that some requirements are inappropriate to your particular project.

Does not the audit refute the claim that modern process standards are designed to be checklists for use as memory aids by skilled software professionals? Does not the audit show that the standards are full of requirements that must be satisfied even when it does not make sense to do so, that they are really

employed to substitute the standard writer's judgment for the judgment of real people on the project?

Actually, it does not. The audit is imposed (directly or indirectly) by buyers, who are customers. A company might voluntarily submit to an audit—an ISO 9000 audit, for example—to certify or register a quality system, but it would only do so with the expectation that the audit results would favorably impress potential customers. Foolish buyers or foolish auditors might insist that developers do foolish things, and they might be more of a nuisance wielding a standard than they would be without it. But buyers and auditors are not under the control of the standard.

The only rule for many modern process standards is to tailor them to suit your development conditions. Now, let us say that someone does this poorly—should we blame the authors of the standard?

Hypoxic Heroes Are Vulnerable

When people argue that all process standards hinder software development, as Bach does, they are promoting a “cowboy” approach that glorifies heroic individuals. According to his logic, you cannot be a hero using a process standard. There are no heroes without risk. In fact, the bigger the risk, the bigger the hero and the more the stress.

Without process standards to nag them at times of stress, when they need them the most, cowboy developers will push past their biological limits with no help in sight. It is like putting climbers into the death zone on Mount Everest with no rules for what to do on summit day.

Stress will cut away their competence. They will not notice. And because they rely only on themselves, they will make bad decisions. Now, some of us resolve, instinctively, to follow a common rule, to check off our mental lists before we act. If I read Krakauer correctly, a big part of the reason that the climbers died on Mount Everest in May 1996 was that, tragically, in their impaired, hypoxic state, they broke their own rules.

The lesson I see for software development is that organizations and projects need process standards the most when their employees are most under pressure and have little time for thought. That is when everyone hits a biological limit and when it is most dangerous to let heroes run free without rules or guidelines.

Acknowledgments

Thanks to James Bach for the spirited, good-natured E-mail debate that helped me to identify the most important objections to my arguments and for his generosity in publishing my original rebuttal in his column. Thanks to Kirk L. Kroeker and the other editors at *Computer* for tightening and smoothing my original language. ♦

About the Author



Lewis Gray is president of Abelia Corporation. He is also a software process improvement coach and long-time teacher of software development standards.

He has 17 years experience developing software systems for government, industry, and academia. He was a leader in the development of MIL-STD-498 and in the development of IEEE/EIA 12207. He is the author of many technical papers on process improvement and software engineering. He is the only instructor outside the Software Engineering Institute who is authorized to teach the TXM model of technology introduction. He holds a doctorate in the philosophy of science from Indiana University.

Abelia Corporation
12224 Grassy Hill Court
Fairfax, VA 22033-2819
Voice: 703-591-5247
Fax: 703-591-5005
E-mail: lewis@abelia.com
Internet: <http://www.abelia.com>

References

1. Bach, James, “Microdynamics of Process Evolution,” *Computer*, February 1998, pp. 111-113.
2. Krakauer, Jon, *Into Thin Air*, Villard, New York, 1997.

Sponsor Lt. Col. Joe Jarzombek
801-777-2435 DSN 777-2435
jarzombj@software.hill.af.mil

Publisher Reuel S. Alder
801-777-2550 DSN 777-2550
publisher@stsc1.hill.af.mil

Managing Editor Forrest Brown
801-777-9239 DSN 777-9239
managing_editor@stsc1.hill.af.mil

Senior Editor Sandi Gaskin
801-777-9722 DSN 777-9722
senior_editor@stsc1.hill.af.mil

Graphics and Design Kent Hepworth
801-775-5798
graphics@stsc1.hill.af.mil

Associate Editor Lorin J. May
801-775-5799
backtalk@stsc1.hill.af.mil

Editorial Assistant Bonnie May
801-777-8045
editorial_assistant@stsc1.hill.af.mil

Features Coordinator Denise Sagel
801-775-5555
features@stsc1.hill.af.mil

Customer Service 801-775-5555
custserv@software.hill.af.mil

Fax 801-777-8069 DSN 777-8069

STSC On-Line <http://www.stsc.hill.af.mil>

CROSSTALK On-Line <http://www.stsc.hill.af.mil/Crosstalk/crosstalk.html>

ESIP On-Line <http://www.esip.hill.af.mil>

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address:

Ogden ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205

E-mail: custserv@software.hill.af.mil
Voice: 801-775-5555
Fax: 801-777-8069 DSN 777-8069

Editorial Matters: Correspondence concerning *Letters to the Editor* or other editorial matters should be sent to the same address listed above to the attention of *Crosstalk Editor* or send directly to the senior editor via the E-mail address also listed above.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the *Crosstalk* editorial board prior to publication. Please follow the *Guidelines for Crosstalk Authors*, available upon request. We do not pay for submissions. Articles published in *Crosstalk* remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with *Crosstalk*.

Trademarks and Endorsements: All product names referenced in this issue are trademarks of their companies. The mention of a product or business in *Crosstalk* does not constitute an endorsement by the Software Technology Support Center (STSC), the Department of Defense, or any other government agency. The opinions expressed represent the viewpoints of the authors and are not necessarily those of the Department of Defense.

Coming Events: We often list conferences, seminars, symposiums, etc., that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the *Crosstalk* Editorial Department.

STSC On-Line Services: STSC On-Line Services can be reached on the Internet. World Wide Web access is at <http://www.stsc.hill.af.mil>. Call 801-777-7026 or DSN 777-7026 for assistance, or E-mail to schreif@software.hill.af.mil.

Publications Available: The STSC provides various publications at no charge to the defense software community. Fill out the Request for STSC Services card in the center of this issue and mail or fax it to us. If the card is missing, call Customer Service at the numbers shown above, and we will send you a form or take your request by phone. The STSC sometimes has extra paper copies of back issues of *Crosstalk* free of charge. If you would like a copy of the printed edition of this or another issue of *Crosstalk*, or would like to subscribe, please contact the customer service address listed above.

The **Software Technology Support Center** was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies that will improve the quality of their software products, their efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery. *Crosstalk* is assembled, printed, and distributed by the Defense Automated Printing Service, Hill AFB, UT 84056. *Crosstalk* is distributed without charge to individuals actively involved in the defense software development process.

Cabbies and Techies: Brothers in Pain

Big-name consultants often say that buggy or late software is usually not the fault of coders and program managers, but the fault of upper-level managers. But how could people no more involved in the day-to-day affairs of software engineering than the average turnip be responsible for sabotaging software projects? There's no need to ask a high-priced software consultant: Ask a cab driver.

Software Executive (*stepping into cab at 4:50*): Driver, I have a meeting at my hotel at 5:00. Step on it!

Crusty Big-City Cabbie: Where are ya stayin', pal?

E: Room 420. The bed's nice, but the shower controls are confusing. Nice convention facilities, though. Go!

C: Right. We'll head off in any old direction 'till we see the hotel with the funny shower knobs. But maybe I can get you there faster if you give me some more information, like the color of your bedspread.

E: This is no time for jokes. Start working on that 5:00 deadline and we'll work out the "where" as we go.

C (*pulling away*): Wait a minute ... you gotta hard deadline and no useful directions for me—that'd make you a software executive, right? Your convention's at the Marriott. I can't get you there 'till maybe 5:15.

E: Unacceptable! But with my help we'll easily cut out those extra 15 minutes. First, we'll eliminate any nonvalue-added elements from your driving process—

C: It's rush hour, pal, and this cab ain't no #@!% helicopter. We ain't goin' five miles in 10 minutes.

E: As if you had any data to back that up. Ten minutes requires a net speed of only 30 miles per hour, and the speed limit on this road is 35. I'm calling my client on my cell phone to tell her we'll be early.

C: Yeah, and while you're at it, you can tell her your cab driver is the Easter Bunny.

(*one minute later*)

E: Driver! Why are we sitting still? I'm not seeing any visible signs of progress!

C: It's called a red light, buddy. Whenever I see one, I just gotta stop and admire it.

E: Work harder! This is an unproductive activity! Hit the gas! Move! Move!

C: Right, pal. We'll just drive over the top of this police car up here and continue on our merry way.

E: That's the can-do attitude I need to see! Well, what are you waiting for? Do it!

C: Aw, the light turned green. Now I'll never get to bunk with my cousin Larry at the state pen.

E: That's not my fault. And you—stop hitting those red lights. A competent driver should know how to avoid them. I'm calling my client to tell her we'll be there at 5:01, thanks to you.

C: Blow it out your ear, buddy.

(*30 seconds later*)

E: Hold it! Why are we turning? My map shows the street we were on is the most direct! Turn around!

C: We're takin' the parkway. And stop micromanagin' me, ya chump. I know what I'm doin'.

E: And I know you're not the one paying for this cab fare—stop wasting time and get back on that road.

C (*turning cab around*): Tell ya what: You give the directions, and I'll charge by the minute—I can use the extra dough. But shut yer yap, or I'm adding a surcharge for not beatin' ya with that briefcase.

(*At 5:00*)

E: It's 5:00! Why aren't we there?

C: 'Cuz ya didn't hail a cab at 4:30, ya nitwit.

E: How did you ever keep this job, missing deadlines like this? And look, another red light! I'm losing patience with you!

C: YOU'RE losing patience? Gimme that briefcase—

E: I'm calling my client to tell her we'll be there at 5:05. Your tip is shrinking by the minute!

C: So's your life expectancy! Shaddap!

(*Much later*)

C: Well, Mr. I'm-Paying-So-I'm-Right, thanks to your brilliant navigation, we made it by 6:04.

E (*getting out*): I wasn't the one stopping at all those red lights! Here's 10 bucks. Keep the change.

C: Whoa, pal! Your fare is \$83.50!

E: Wrong—that fare is completely outside my cost plan. Plus, it's over 57 times the cost of a subway fare, and I'm assessing penalties for missing your deadline. A smarter cabbie would have gotten me here by 5:00.

C: A smarter cabbie would have run you over on sight, buddy. Gimme my fare before I get any smarter!

Woman Exiting Hotel (*to executive*): Oh, there you are. I was just on my way to dinner.

E: I would have been here at 5:00 if it weren't for this incompetent cab driver.

W: Don't worry about it. I work with software executives all the time, so I wasn't expecting you until 7:00.

E: Let's do a dinner meeting then. (*To cabbie*) I was at a great restaurant here in town the other night—some kind of ethnic food. They had waiters and tablecloths and menus—the whole nine yards. Take us there.

C: You ain't goin' nowhere 'till I get my \$83.50.

E: Fine, be that way. Here's \$100. On second thought, I think I know where it is. Just follow my directions.

C: If you're navigating, forget it. I gotta have the cab in the garage by midnight.

— Lorin May

Got an idea for BACKTALK? Send an E-mail to backtalk@stsc1.hill.af.mil