ASSIGNMENT 1

Systems Integration

of the

777 Airplane Information Management System (AIMS)

Bob Witwer
Honeywell Inc. — Air Transport Systems

ABSTRACT

The systems integration of the 777 Airplane Information Management System (AIMS), both within the AIMS system and with the other systems on the airplane, represented the most complex system integration effort ever undertaken at Honeywell Air Transport Systems Division. The technological innovations in the AIMS design, coupled with an aggressive program schedule, were major factors in the AIMS challenge. Honeywell and Boeing had to work closely together to complete the design and development of AIMS in a time frame that supported the 777 Early ETOPS goal. With teams from the two companies working as one unit, redundant activities were eliminated, technical and program problems were identified and solved rapidly, and schedule time was saved by both teams helping with tasks that were traditionally considered to be the other team's job.

OVERVIEW

The Integrated Modular Avionics (IMA) architecture implemented by the 777 Airplane Information Management System (AIMS) represents a radical departure from the federated LRU architecture that exists on most commercial transports today. The design, development, integration, and testing of this system would have been a major accomplishment even on a traditional new airplane development program. Two complicating factors on the 777 program were the plan to receive ETOPS (Extended Twin Operations) approval immediately after initial type certification, and the desire to provide an airplane to the airlines that was service-ready at initial certification. To reach a high level of system maturity in time to support the ETOPS mission, Honeywell and Boeing developed a much closer and open approach of working together. This approach was used not only at the program management level, but also to resolve design issues, integrate AIMS lab facilities in both Phoenix and Seattle, support flight tests, and jointly perform system verification.

This approach was a huge success. It enabled the Honeywell-Boeing AIMS team to accomplish far more than what could have been done using a traditional airframer/supplier relationship. This paper will examine, in more detail, some of the key areas where Honeywell and Boeing worked together to make AIMS a reality, specifically: system design, system integration, system verification, and program management. In order to more fully understand and appreciate this discussion, an architectural overview of AIMS follows.

SUMMARY OF AIMS ARCHITECTURE

The philosophy behind the AIMS architecture is to provide a host platform where avionics applications (e.g., Flight Management, Displays) can share common platform resources. A block diagram of the AIMS architecture is in Figure 1 (on next page). The heart of the

Author's Current Address:
Honeywell Inc., Air Trans. Sys., Box 21111, MS: AZ75-203582, Phoenix, AZ 85036-1111

Based on a presentation at the 1995 Digital Avionics Systems Conference.
0885-8985/96/ $5.00 © 1996 IEEE

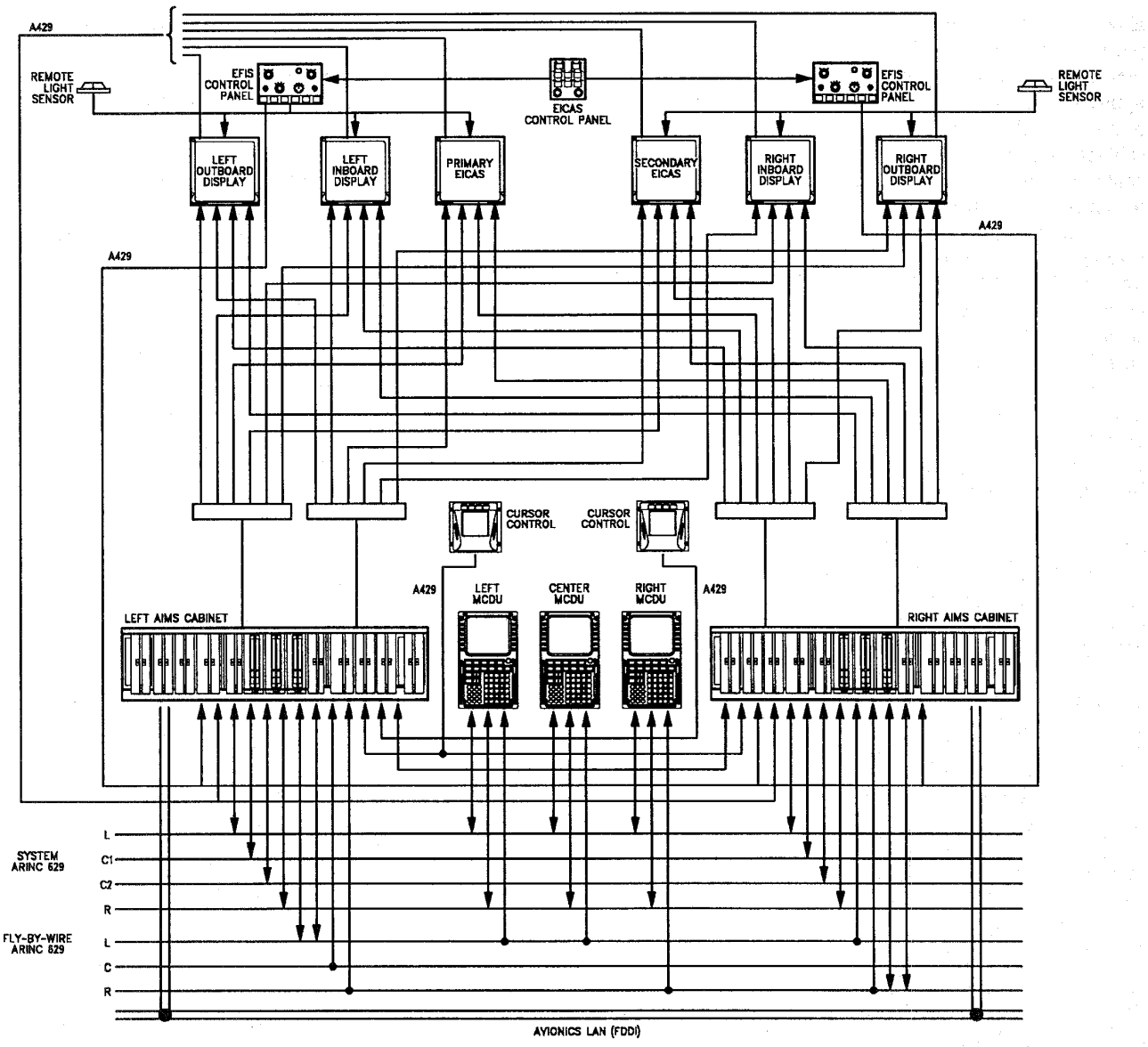


Fig. 1. AIMS Baseline Architecture

AIMS system consists of dual cabinets in the electronics bay that each contain 4 core processor modules (CPMs) and 4 input/output modules (IOMs), with space reserved in the cabinet to add one CPM and two IOMs to accommodate future growth. The shared platform resources provided by AIMS are:

- common processor, power supply, and mechanical housing;
- common input/output ports, power supply, and mechanical housing;
- common backplane bus (SAFEbus™) to move data between CPMs and between CPMs and IOMs; and
- common operating system, built-in test (BIT) and utility software.

Instead of an individual application residing in a separate LRU, applications are grouped together on CPMs. The IOMs transmit data from the CPMs to other systems on the airplane, and receive data from these other systems for use by the CPM applications. A high-speed backplane bus, called SAFEbus™, provides a 60 Mbit/second data pipe between any of the CPMs and IOMs in a cabinet. Communication between AIMS cabinets is through four ARINC 629 serial busses.

The robust partitioning provided by the architecture allows applications to use common resources without any adverse interactions. This is achieved through a combination of memory management and deterministic scheduling of application software execution. Memory is

allocated before run time, and only one application partition is given write-access to any given page of memory. Scheduling of processor resources for each application is also done before run time, and is controlled by a set of tables loaded onto each CPM and IOM in the cabinet. This set of tables operates synchronously, and controls application scheduling on the CPMs as well as data movement between modules across the SAFEbus™.

Hardware fault detection and isolation is achieved via a lock-step design on the CPMs, IOMs, and the SAFEbus™. Each machine cycle on the CPMs and IOMs is performed in lock-step by two separate processing channels, and comparison hardware ensures that each channel is performing identically. If a miscompare occurs, the system will attempt retries where possible before invoking the fault handling and logging software in the operating system. The SAFEbus™ has four redundant data channels that are compared in real time to detect and isolate bus faults.

AIMS is designed to be fault-tolerant such that the aircraft can be dispatched for 10 days without maintenance at 99% probability of success. The applications hosted on AIMS are listed below, along with the number of redundant copies of each application per shipset in parentheses:

- Displays [4];
- Flight Management/Thrust Management [2];
- Central Maintenance [2];
- Data Communication Management [2];
- Flight Deck Communication [2];
- Airplane Condition Monitoring [1];
- Digital Flight Data Acquisition [2]; and
- Data Conversion Gateway [4].

All of the IOMs in the two AIMS cabinets are identical. The CPMs have common hardware for processor, memory, power, and SAFEbus™ interface, but have the capability to include a custom I/O card to provide specific hardware for an application "client." The client hardware in AIMS includes the displays graphics generator, the Data Communications Management fiber optic interface, the Digital Flight Data Acquisition interface to the data recorder, ACARS modem interface, and the Airplane Condition Monitoring memory.

The other flight deck hardware elements that make up the AIMS system are:

- Flat Panel Display Units [6];
- Control and Display Units [3];
- EFIS Display Control Panels [2];
- Display Select Panel [1];
- Cursor Control Devices [2]; and
- Display Remote Light Sensors [2].

SYSTEM DESIGN AND DEVELOPMENT

The AIMS team began the program in October 1990 with some monumental design challenges in front of them. Since the architecture was so different, many new designs were required; CPM, IOM, SAFEbus™ and its interfacing hardware, and Flat Panel Displays are major examples. Boeing and Honeywell worked together on the Specification Control Drawings (SCDs) for AIMS (18 in all) to define the system level requirements. A highlight of this activity occurred in May of 1991, when as many as 65 Honeywell engineers went to Seattle to support a three-week push for the initial SCD release. System requirements were freely challenged by engineers at both companies. This process helped ensure that the requirements that drove the system design were necessary and of high-value.

System interface definition and control posed a major challenge, since AIMS interfaces with the majority of avionics systems on the 777. Boeing developed an electronic Interface Control Document (ICD) to define and control system interfaces at the airplane level. Honeywell developed a complementary Data Capture Tool (DCT) that took in the electronic airplane ICD information that related to AIMS. DCT then allowed AIMS engineers to electronically tag each ICD data item that was set or used by their application, and also provided for the entry of system requirements for application memory and processor throughput. This DCT database became the input for the SAFEbus™ Scheduler Tool. This tool generates the set of SAFEbus™ tables which are loaded onto each CPM and IOM to control the movement of data within the cabinet and to control the scheduling of each application's actual run time on the CPM.

Managing the allocation of cabinet resources (processor throughput, memory, and bus bandwidth) was a big task, and a very dynamic one. Some of the applications were new designs and had no historical resource data to use as a starting baseline. Many applications saw their initial estimates grow once detailed system design began. A central systems integration group had responsibility for working together with the application functions to establish memory, throughput, and bandwidth budgets for each application, and to manage to these budgets throughout the design phase. This activity kept a constant focus on resource-efficient design, and in some cases caused the Honeywell/Boeing team to change the AIMS system requirements to allow functions to fit within their resource allocation.

SYSTEM INTEGRATION

The need to begin application and system integration before the actual airborne hardware could be completed required "simulation" CPMs and IOMs to be developed for use in both Honeywell and Boeing lab

facilities. These modules did not have the throughput or fault containment that the airborne hardware had, but enabled initial application integration to begin seven months before the flight hardware was available. They also provided proof-of-concept validation for many new aspects of the system, such as SAFEbus™, partitioning, and application design.

A schedule for AIMS software loads was defined early in the program to allow AIMS functionality to be developed in a logical progression which supported Boeing factory and flight test needs. Application teams performed software builds (compile/link) privately to integrate their application to a level that would support AIMS level integration. A central Software Integration Team had responsibility for generating the AIMS level software builds on specific dates. Using a central software build team ensured that AIMS level builds had compatible interfaces and functionality between applications.

Lab integration in Phoenix included Boeing on-site personnel that worked with each application team. Integration was also facilitated by a central platform test team made up of both Honeywell and Boeing engineers. This team worked with application engineers to troubleshoot problems with the platform or the application-platform interface, and was crucial during both the simulation hardware and flight hardware integration phases.

Lab integration in Seattle began immediately after the first delivery of simulation hardware and application software, and included Honeywell on-site engineers. A major contributor toward well-coordinated integration testing was the daily integration teleconference between Seattle and Phoenix. This allowed the integration performed in both facilities to be complementary, and minimized redundant testing. It also allowed problems discovered in one facility to be quickly communicated to the other, so that time wasn't wasted investigating problems twice. Boeing integration of AIMS applications took place on Boeing test benches, while initial airplane-level integration was performed using Boeing's 777 Systems Integration Lab (SIL). The SIL contained actual 777 avionics equipment, complete with airplane-level electrical and power interface connections. The use of this facility reduced the amount of on-airplane testing that would have otherwise been required, since it allowed actual system-to-system interfaces to be exercised before going to the airplane.

On-airplane testing took place first in the 777 factory, with AIMS as a central component in the airplane build, test, and troubleshooting process. The AIMS Central Maintenance Computing System allowed Boeing factory personnel to test the other avionics systems on the airplane, while the AIMS Displays System provided maintenance pages which gave a dynamic, graphic indication on the flight deck displays of the data values of the interface signals between AIMS and the other airplane systems. Honeywell rotated Phoenix integration engineers to Seattle for 2 month on-site assignments to support the integration and use of AIMS in the factory. After first flight, Honeywell and Boeing worked together to support a very aggressive flight test schedule, with as many as 6 airplanes flying at any one time. Flight test status and problems were communicated daily from Seattle using electronic mail. This not only proved to be an effective method of initial problem reporting, but was well-received by the hundreds of engineers in Phoenix and Seattle working on AIMS as a way to keep abreast of the status of the 777 flight test program.

SYSTEM VERIFICATION

AIMS contained more than 600K lines of software, so the verification effort was enormous compared to past programs at Honeywell. Honeywell and Boeing engineers working in Phoenix and Seattle participated in the formal verification effort, which included test planning as well as actual test development and execution. By doing joint test planning, Honeywell and Boeing were able to minimize test redundancy and complete system testing in time to support the initial type certification in April of 1995. A test coordination team consisting of engineers and managers from both companies was formed early in the test phase of the program. This team used regular teleconferences (daily during the last months of the program) to remove obstacles from the test process, facilitate the shifting of test resources, and exchange application and overall AIMS test status.

As with the system integration phase, the test phase utilized lab resources from both companies. A great deal of application testing was performed using a mainframe computer-based simulation test bed. Test benches at both companies, populated with real and simulated AIMS hardware, gave application teams the capability to integrate their own functions and to do some low-level integration with other applications. Honeywell designed and built two full flight deck simulation facilities to allow integration and verification testing of the entire shipset of AIMS equipment. These two facilities and the Boeing SIL were also used to run a series of flight and maintenance operational tests. These tests exercised AIMS in the same way as an airline flight or maintenance crew, and helped to validate the AIMS design requirements.

PROGRAM MANAGEMENT

It was recognized early in the program that there was a need for exchanging information between Phoenix and Seattle. An electronic link between Seattle and the Honeywell mainframe computer allowed electronic mail to be exchanged between companies, and this feature was extensively used throughout the program for informal communications. To facilitate the exchange and retrieval of formal communications, Honeywell developed a tool for

managing coordination memos and action items. The system was hosted on the Honeywell mainframe, and was therefore accessible to both companies. Coordination memos and action items could be drafted, sent, sorted, and printed by personnel from Honeywell and Boeing. The on-line accessibility of the tool made formal communication and action tracking efficient.

A single problem reporting and change control database system was used on the program, and was accessible by both Boeing and Honeywell engineers. This saved time over past programs where separate problem reporting systems were used, since it cut down on the number of redundant problem reports written, and eliminated the problems with losing problem reports or inaccurate translation of problems between reporting systems.

The concept of honest status sharing was used from the beginning on the program between both companies. Although some traditional techniques were used to manage the AIMS program (e.g., weekly status faxes and monthly program status reviews), program problems were attacked in a positive way by the Honeywell-Boeing team. Blame was never a focus; coming up with positive actions to fix problems was emphasized throughout the program. This open approach to program management allowed problems to surface and be solved before they could have major negative impact on the program. Taking this approach not only resulted in program success, but it forged working relationships between Honeywell and Boeing that were based on trust and honesty.

SUMMARY

The teaming approach used by Honeywell and Boeing on the 777 AIMS project was instrumental in the success of the program. System design, integration, verification, and program management activities were conducted in an atmosphere of open communication and with a desire by both companies to do whatever was required to get the job done. The result of this commitment to teaming is not only a state-of-the-art avionics system on the world's newest transport airplane. It is a model for a better way to develop avionics systems in the future.

Bob Witwer is manager of the 777 AIMS Systems Integration Department at Honeywell. He has worked on the 777 AIMS program since November of 1990, and has over five years of previous experience in Flight Management Systems Engineering at Honeywell. Bob received a BSEE from the University of Illinois in 1977.

Software Development for the Boeing 777

Ron J. Pehrson, The Boeing Company

For the Boeing Commercial Airplane Group (BCAG), the 777 airplane represented an unprecedented software challenge in terms of the size and complexity of the airplane's systems. The 2.5 million lines of newly developed software were approximately six times more than any previous Boeing commercial airplane development program. Including commercial-off-the-shelf (COTS) and optional software, the total size is more than 4 million lines of code. This software was implemented in 79 different systems produced by various suppliers throughout the world. Connecting these systems is a complex network of broadcast and point-to-point data buses. Moreover, we committed to provide unprecedented product maturity and quality at first delivery.

This article discusses how the software challenges were successfully met, which resulted in the on-schedule delivery of the first 777 to United Airlines four and one-half years after the program's kickoff. As with all successful projects, first and most important, the skill, dedication, and efforts of all the people involved were the key to its success. However, many large software projects have skilled and dedicated workers, yet they are plagued with missing functionality and schedule slides. Rather than address all the aspects of a successful project, this article discusses techniques used to address two areas that typically create major challenges for large embedded software development efforts: requirements specification and management of the software development process.

Requirements Specification

The specification of airplane systems requirements to the suppliers of the systems is done through specification control drawings (SCD). A major component of that specification is the interface control document (ICD). To assure that the systems will combine to perform as intended, extensive effort was put into the development, control, and validation of an airplane-level ICD and detailed SCDs for each of the airplane systems.

Specification Control Drawings

SCDs were developed for each of the systems on the 777. They ranged in size from approximately 100 pages for the simplest systems to over 10,000 pages for the most complex. The SCD is the primary means for Boeing system designers to communicate the requirements to the supplier of the system. It formed the basis of an ongoing dialog between system designer and developer and between the designers of the various systems. Changes to the SCD were subject to rigorous change board review and approval. The reviews assured that all aspects of a change were addressed, and the effects of the change were reflected in all the affected systems. These reviews also controlled the amount of change, assuring that only necessary changes were made to the systems. Control of changes was increasingly important as development progressed and the effects

of changes could increasingly jeopardize the schedule. This led the BCAG to apply increasingly stringent criteria on acceptability of changes. The importance of controlling change cannot be overemphasized.

By going into great detail in many areas of the specification, the SCD captured not only the understanding of how the individual systems must operate but also insured that the integrated functionality of the system would perform as intended. It also facilitated the system-level analyses that allowed validation of the airplane functionality described later.

ICD Database

As the airplane architecture and functionality of the various systems were evolving, the definitions of the interfaces of each system were placed in a large relational database. The database describes over 40,000 digital data items and 3,000 analog signals. Crucial to the successful use of this database was a tool developed to scan for discrepancies in data definition between the provider and user of the interface data.

As the functionality of the systems continued to evolve, a series of interface block points was established and resolution of discrepancies continued to be a major focus. The block points were established to satisfy the incremental development of overall systems functionality. The block points represented a coherent definition of system interfaces that supported a pre-defined set of systems functionality. The definition of these sets included specific capabilities in terms of airplane functions as well as the level of maturity of the system and its ability to tolerate failure conditions. The definition of the block points allowed integration of the systems to proceed in parallel with continued development of functionality within the systems. This not only achieved an accelerated system integration, but also identified integration problems at a time when they could be addressed as the systems development proceeded.

Reports from this database formed the basis of the interface specifications for the individual system ICDs. Rigorous configuration control was applied to each block point. Change boards were conducted for changes both at the major function level and at the airplane level to assure the changes were necessary and that all consequences of the changes were considered throughout the airplane.

In the first two months, analysis of the database uncovered over 50,000 discrepancies. An intensive, high priority effort reduced the discrepancies to 3,000 within two months. Use of the ICD database and associated tools allowed us to deal with the increased interface complexity of our system, yet have unparalleled success in integrating the systems in the laboratory and on the airplane.

System Functional Analysis

It is important to have well-defined and carefully controlled requirements, but it is equally important to minimize changes to those requirements as the systems are implemented. On a large, complex system such as the 777, it is common to find

specification problems late in the program when the systems are brought together in the integration facility or even in flight test. To prevent these problems, a disciplined series of functional analyses was performed throughout the development of the system specifications. In all cases, both nominal and failure conditions were considered. This effort was led by a dedicated system integration organization that coordinated and organized the process. In all cases, there were regular management reviews of compliance and completion of the process.

The first phase of analysis was performed early in the airplane program during the development of the SCDs and ICD. Using regulatory requirements and high level functional definitions of the airplane, the functionality of each system was reviewed. The reviews covered both nominal and off-nominal operating conditions. Allocation and definition of requirements were performed. The output of this process was reflected in the SCDs and ICD.

The next phase of analysis focused on the interfaces between the systems. This analysis was performed on selected systems, based on complexity, criticality, and customer visibility of the system. For each selected system, the inputs and outputs were reviewed. The goal was to verify that all users of data were using it correctly. This activity was performed after the first phase of analysis, but at a very early stage in the development of the systems by the suppliers.

The last phase of analysis was at the airplane functionality level. These analyses were performed as the development of the systems was well underway, but prior to final system integration. This activity validated correct performance of the airplane systems as a whole. For each significant airplane-level function that required multiple systems, we verified that the function was properly performed and that failure conditions were properly handled.

These were all manual analyses performed early enough in the program to minimize impact on development of the individual systems and in time to significantly reduce the problems found as systems were delivered and integrated. The result of the three phases of analysis was that problems in the specification of the airplane systems were identified and corrected at the earliest possible point, thus mitigating the impact of corrections on the overall program schedule.

Managing the Software Development Process

Even with the best specifications, completion of large, complex embedded software systems on schedule has proved elusive. Systems are frequently declared as 90 percent complete only to find they are less than half done. Major delays in product deliveries are often only "discovered" when it is too late to recover from the situation.

With an aggressive flighttest program, the most ambitious in Boeing commercial airplane history, and 79 systems to integrate, it was essential that we had visibility on the real status of the software development on all our systems. With requirements complete for

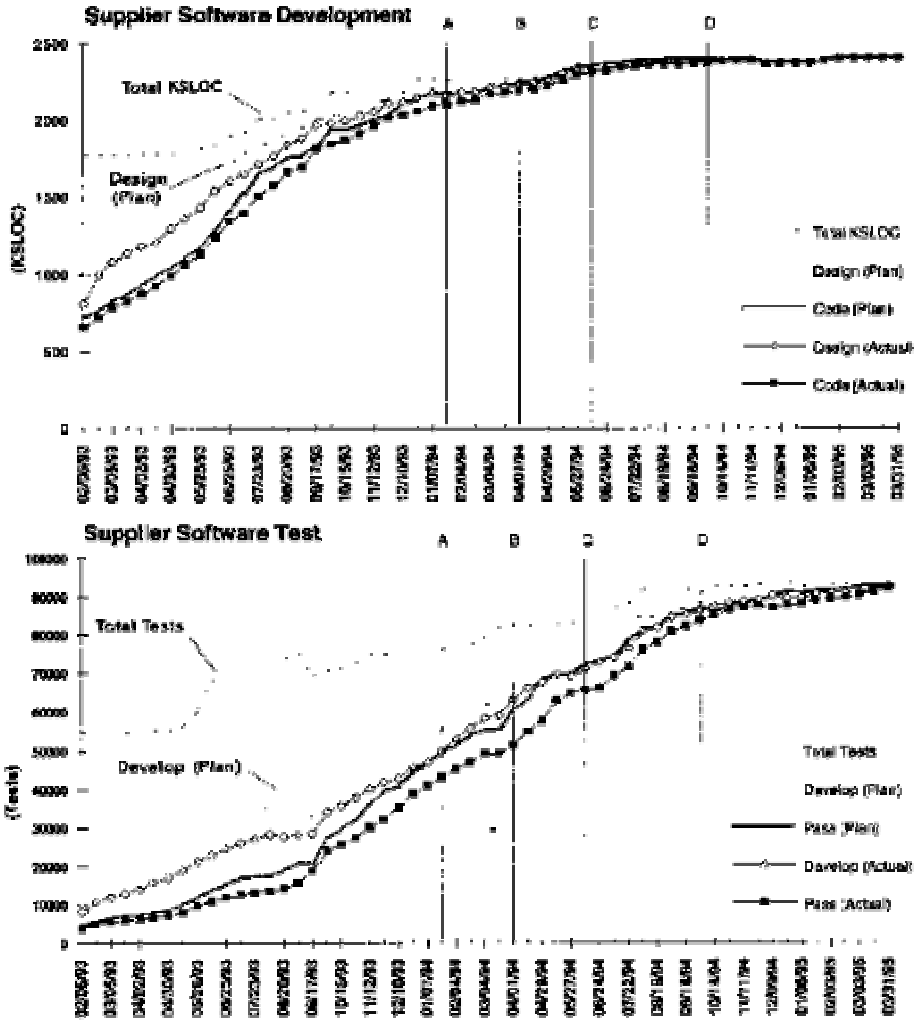
nearly all our systems and software development well underway, an airplanewide software metrics program was instituted. The metrics included development plans and progress and utilization of computational resources.

Development Metrics

The development metrics tracked progress against the plans for design, code, test procedure creation, and test completion. They included the predicted total software size (in lines of source code) and total number of tests. The metrics charts showed key milestones in the airplane program. These milestones represented interim points to measure our progress against the ultimate completion of the program. Associated with these milestones were success criteria based on completion of design, code, and test of the software products.

Figure 1 shows the total airplane roll-up of these metrics. The metrics also measured utilization of computational resources (throughput and memory); however, this discussion will focus on the design, code, and test metrics.

Figure 1: Total Metrics Roll Up for the 777



Datelines legend for Figure 1:
A-02/01/94 Target-PFOD C **B**-04/09/94 Actual-Roll out **C**-06/12/94 Actual-First Flight
D-10/04/94 Target-ETOPS FF

The process of using these metrics was as follows:

· Each supplier was requested to prepare plans for their design, code, and test activities. These plans showed expected totals and the planned completion status for each of the biweekly reporting periods until the task is complete. Even at this early stage in the metrics process, we received our first benefits as we discovered that some suppliers' initial plans did not support the program milestones. This proved invaluable and in a few cases was the only major corrective action we needed to take to assure the supplier supported the program.

· Following the initial plan submittal, there were biweekly updates that showed the actual status of the development in terms of completed design, code, and tests. Any changes in the estimated total size of the effort were also provided and the plans modified to correctly reflect the new total. Plans could also be changed at any time; however, previously reported plans and actual status could not be adjusted. The metrics information was shared with the developers of the systems. This led to important discussions on how we were going to succeed at an early enough point in the program that we could actually do something about it.

· Indications of a healthy program are fairly obvious a plan that supports program milestones and status that consistently tracks the plan. Programs that needed special attention often were several weeks behind the plan line, had numerous replans, or had plans that required unprecedented productivity to be successful. We quickly established reasonable productivity figures that could easily test the feasibility of suppliers' plans based simply on head count, work to go, and time to go. The metrics provided an excellent vehicle for discussions about how the program was going to deal with their current status and get back on a schedule that supported the overall airplane program.

The above process was as important as the measures themselves in assuring our success. However, there were certain characteristics of the metrics program that were key to supporting this process and making it all work. They were as follows:

· Uniformity.
· Frequent updates.
· Clear definition.
· Objective measures.
· Replans, as needed, are allowed and even encouraged.
· Past plans and actuals are held constant.

The uniform nature of the metrics enabled comparison across systems and supported communication of objective status to all levels of program management. This was particularly important with the large number of organizations involved in the software

development. We were also able to combine status information from several different systems provided by a single supplier. This provided unique opportunities to discuss how the supplier was supporting our overall program and to focus needed resources to solve schedule problems.

Considerable effort was made to clearly define measures. This led to a 21-page set of instructions to our suppliers on how to prepare metrics data. The data items measured were objective and easily observed. The combination of these meant there was little confusion about what the metrics meant and real conclusions could be drawn from the data. Moreover, without this we could not have achieved the desired uniformity.

Two aspects of the metrics plans were critical: replanning when needed was encouraged, and past data was never changed. The essence of a plan is it shows how to get from here to there. Once you have significantly deviated from a plan it no longer serves that purpose. Throughout the metrics process we used deviation from the plan as an indicator of problems. Since replanning was encouraged, the only reason to not be close to your plan is you don't have a plan. We found that projects that were several weeks behind their plan did indeed need help.

This approach to software project metrics repeatedly "saved our bacon." Starting with initial plans, they indicated where program milestones were not being supported. Continuous monitoring through testing identified schedule problems early in the development process. The metrics were invaluable in showing us where program risk points were soon enough to take corrective action.

What About Ada?

Prior to the start of the 777 program, BCAG determined that Ada would be the standard programming language for airborne systems. This decision was based on the intrinsic value of having a standard language and the fact that Ada was designed to facilitate sound software engineering. It was also hoped that we could leverage off the considerable efforts spent on Ada in the development of defense systems.

What we have learned so far about the use of Ada on the 777 is a mixed message. Ada was used on 60 percent of the systems and represented 70 percent of the lines of code developed on the 777. We found no correlation between the language used and the number of problems found on the system. We found instances where Ada was used effectively, and the developers felt it substantially reduced software integration problems. In other cases, development was hampered by problems with compilers and other support tools. Many suppliers chose to use a restricted subset of Ada, which led to fewer problems but lesser benefit.

In situations where the imposition of Ada would have created more risk than benefit, other languages such as C or Assembly were allowed. By taking a pragmatic position on this standard, we removed risk and helped the program succeed. It is likely that much of the benefit of Ada will be seen later in maintenance of the software, but that will be

difficult to quantify. The richness and complexity of the language helped knowledgeable users with mature tools achieve modest productivity gains. However, the complexity of the language caused headaches for other users who had to work through compiler problems.

We continue to evaluate our development standards. I expect we will retain Ada as the standard language. A standard language allows the use of tools to aid in the development of software that would be difficult or impossible to implement in a multilanguage environment. As the systems get larger and more complex and the developers and tools they use mature, the value of Ada should increase.

Conclusions

Many aspects of the 777 program were done right to support successful completion of the software systems. Rigorous requirements development process and vigilant management of software development were two keys to that success. This required specific processes and tools as well as the commitment of the program to follow through on their execution. The results were requirements that were stable and complete at an earlier stage in the program and supplier software development processes that had sufficient oversight to allow corrective action and on time completion of the product.

Ronald J. Pehrson
Manager, Embedded Software Engineering
Boeing Commercial Airplane Group
Everett Division _ Mail Stop 03-JC
P.O. Box 3707
Seattle, WA 98126-2207
Voice: 206-294-1115
Fax: 206-294-6504
E-mail: rjp8002@kgv1.profs.ca.boeing.com

About the Author

Ron Pehrson has over 20 years experience in software engineering with the Boeing Company. He has worked as a design engineer on such projects as B-1 and B-2 avionics, B-52 offensive avionics system, and B-1B avionics simulation. He was a software design supervisor on the B-1B weapon system trainer, advanced tactical fighter demonstration/validation, Boeing Commercial Airplane Group central software engineering, and the 777 propulsion and avionics software. He is the manager of embedded software engineering for the Boeing Commercial Airplane Group. Pehrson has both a Bachelor of Science in mathematics and a Master of Science in computer science from the University of Washington.

Source: http://www.stsc.hill.af.mil/crosstalk/1996/01/Boein777.asp

Testing Boeing's 777:

Aerospace giant leaves nothing to chance.

Written by Mark Lapin for Test & Measurement World, April 1995

This article was prepared with the cooperation of:

Art Fanning, Implementation Manager, Boeing Integrated Aircraft Systems Lab

Bob Dawes, Test Manager for Boeing Systems Integration Lab

Carl Tenning, Design Supervisor, 777 Electrical Power Systems

Tom Moore, Senior Engineer, Boeing Commercial Airplane Group

Boeing's 777, the plane with the luckiest number in the air, was designed and tested by people with a profound aversion to trusting in luck, chance or anything other than confirmed and repeatable test results.

From the earliest stages of design, Boeing was determined to make its new 777 twinjet transport the most "customer driven" aircraft the company had ever launched. To that end, Boeing interviewed customers around the world, asking them to describe their ideal plane. The overwhelming consensus was that customers wanted a new airliner to be 100% service-ready from the first day of delivery.

"Manufacturers have been trying to deliver service-ready aircraft from the time the industry began," said Art Fanning, Implementation Manager of Boeing's Integrated Aircraft Services Laboratory. "So we took that answer as kind of an indictment of the way the industry had been doing things. Boeing decided we were going to have to do things differently if we wanted to get a different result."

The way Boeing and other aircraft manufacturers (commercial and military) had been doing things was to use the first airplane of a new generation, or "airplane one," as the primary integration vehicle. The first time that all the different systems that make up a modern airliner actually came together was on the vehicle designated for test flights. And the

first time engineers had the opportunity to study the total "system of systems" operating under real world conditions was when that vehicle took to the air during its first test flight.

"In building the first of anything," said Fanning, "you have an opportunity to complete the design. Even if the design is right, you may not have thought through all the interactions. You discover things like an interface with two connections on one side and three on the other. In the past, we knew what bits and pieces needed to be fixed, we just didn't have time to correct them before the first scheduled delivery. So we decided we had to see how everything worked together in the lab before we got to the production line."

Since there was no test lab in existence big enough and sophisticated enough to test all the systems of a modern airliner like the 777 in an integrated and realistic manner, Boeing decided to build one. The result of that decision is the IASL or Integrated Aircraft Systems Laboratory, probably the most advanced, comprehensive and integrated test lab in the aircraft industry.

Overview of the IASL-- a very sizable investment in supplying service-ready aircraft

The major advance achieved by the IASL is not just that it moves integrated testing further up-stream in the production process, but that it conducts laboratory tests with an unprecedented degree of realism. As Bob Dawes, Test Manager of the Systems Integration Laboratory, put it, "We don't just simulate the real world. This is the real world."

Built on the Duwamish Waterway in South Seattle, the IASL covers more than half a million square feet, employs more than 1,100 people, and houses 70 functional test systems for the 777. Sixty four of these systems for stand-alone tests. The other six conduct tests requiring "airplane level" integration. The highpoints of the complex are:

the Systems Integration Lab (SIL) where the electrical, electronic and avionics systems of the 777 undergo integrated testing;

the Electrical Power Systems Lab (EPSL) where engineers test the 777's crucial power generation systems;

the Flight Controls Test Rig or Iron Bird, used to validate the flight control system and electrical/hydraulic support systems;

and the fully operational flight decks used to simulate how the plane will handle under all flight conditions.

Boeing invested $30 million simply in preparing the site for the IASL. Constructing the building cost another $109 million. These expenses pale beside the value of the test equipment housed inside the complex--estimated at around a quarter of a billion dollars. "Add to that the value of things we test, the hardware and pieces of the airplane manufactured by Boeing or supplied by vendors," said Fanning, "and you have a pretty sizable investment in supplying service-ready aircraft, which indicates how strongly we feel about that goal."

Completed in October of 1992, the IASL had all its major test systems up and running by mid-1993. Thus almost a year and a half before "airplane one" rolled down the runway for the first test flight of the 777, IASL personnel had integrated final versions of the aircraft's myriad systems and were test flying "airplane zero" in the lab.

"Similar testing had been done on a smaller scale," said Fanning, "by us or other companies. But at this point in time, I don't think there's anything else that approaches the magnitude or capabilities or the degree of integration that the IASL complex has."

Stand-alone Laboratories

The stand-alone labs in the IASL provide individual engineering disciplines with the ability to conduct

validation testing on multiple elements of their systems. Among the many individual disciplines represented are avionics, mechanical/hydraulic systems and environmental controls.

The 777 Environmental Control Systems organization is a good example of stand-alone lab capability. Current validation testing includes the cabin air conditioning and temperature control system; the air supply control system; the air foil and cowl thermal anti-ice systems; and the duct leak and overheat protection system. To make the testing as realistic as possible, 777 environmental control engineers incorporate all of the interfacing signals from other nearby systems, either through simulation or by using actual system hardware.

Environmental Testing in the Components Test Lab

Thermal and vibration stress-cycling are not a major part of the IASL's mission because the lab works primarily with aircraft systems or black boxes, which are manufactured and stress-tested by outside suppliers before delivery to Boeing. However, there is one lab in the complex which concentrates on stress testing. The Components Test Lab screens electrical connectors, wires, circuit breakers, relays and other components for the 777's electrical systems. The lab is equipped with 17 environmental test chambers. It tests the response of components to extremes of temperature (-70^0 to 200^0C), humidity (25 to 95%) and altitude (up to 110,000 feet). There is also a salt spray chamber for corrosion testing; an oven with a maximum temperature of 650^0C; and two thermal shock chambers, one with a range from 260^0C to 200^0C, the other for hot-cold thermal testing from -73^0 C to $+100^0$C. Some of the environmental chambers are equipped for functional testing on aircraft systems although the main emphasis of the lab is on testing components. Once a component has been qualified as a Boeing standard, it joins a list from which designers of electrical systems for the 777 and other Boeing aircraft can draw as needed.

The Systems Integration Lab-- a full airplane flown

by pilots from the experimental flight test program

Although traditional environmental testing sense is limited at the IASL, the effort to reproduce the real-world operating environment of aircraft systems is the central goal of most labs in the complex. The technological highpoint of the IASL is probably the Systems Integration Lab. It is here that engineering test pilots fly airplane zero, which consists of virtually all of the electrical, electronic and avionics equipment used in a 777. The SIL tests 900 production wire bundles out of the 1,200 bundles used in an actual aircraft. The wiring connects to over 100 black boxes or Line Replaceable Units (LRUs). All that's missing are rows of seats and the outer skin of the plane.

"The lab is essentially a full airplane," said Bob Dawes, Test Manager of the SIL. "The people who fly it are pilots from our experimental flight test program. Lab personnel support the pilots and surround the real aircraft boxes with test system after test system used to simulate aerodynamic behavior and record data."

Up to 43 Harris Nighthawk computers are available to do real-time processing for the SIL and other integration labs. The Nighthawks provide the environment around the SIL airplane. They simulate air speed, altitude, temperature, the runway environment, the navigational environment, etc. In one test, the Nighthawks were used to simulate the operational environment that aircraft systems would face if the 777 were taking off in a fierce crosswind with one failed engine. Other tests were conducted to see how the aircraft would respond to the environment of Bolivia's La Paz airport which is some 13,300 feet above sea level. While the Nighthawks simulated the high-altitude environment, test pilots powered up the SIL airplane and went through a complete test flight. Other simulations involved flights over the North Pole to test navigation systems and non-normal, non-routine events such as catastrophic engine failures. Systems performance and interaction are recorded by a data acquisition

system capable of capturing up to 57,000 parameters at up to 24 million bits per second.

"From an engineering standpoint," said Dawes, "the lab tests aircraft boxes in the environment they really work in. It's bus-loading with all the boxes talking as they do on a real airplane. We're not just seeing how systems perform individually. We're seeing how they communicate in the real world. Military programs also have labs called SILs. But even in the military, they don't approach the fidelity and extensive reproduction of the airplane that we have."

A typical test flight on airplane zero

A typical test in the SIL begins with the generation of a test plan that sets forth the test requirements such as high or low altitude, hot or cold day, polar or trans-oceanic flight. On the day of test, all participants gather for a pre-flight conference. They go through the steps of the test line by line. The test pilots often look at things from a different perspective than the test engineers and request modifications in the plan. At the conclusion of the conference, the flight crew climbs up to the flight deck. Depending on the test requirements, they may find the aircraft totally cold and power it up through a complete pre-flight check list. The test plan may call for the introduction of failures, in which case the pilots react with corresponding non-routine procedures.

A great deal of realism goes into these test flights. Lab personnel enact the role of the ground crew and feed power to the plane the way a ground crew would on the airport ramp . The pilots call for disconnection over their headsets and the ground crew disconnects. After the crew powers up the Auxiliary Power Unit, computers simulate turbine and airflows. Actual 777 electronics generate all signals to the pilots. If the test calls for a "hung start" or failure to start engines, pilots have to respond with appropriate measures.

When the airplane is powered up, the pilots test taxi and steering controls. They use the aircraft's internal

and external communications systems. They taxi to the imaginary runway, take off and fly the profile of the day, which may call for an eight-to-ten hour flight over the Northpole, or may involve up to 50 take-offs and landings in the space of an hour. The simulation may involve various kinds of landings-- skipped landings, bounced landings, or other tests to check landing logic and verify that the aircraft knows when it is in the air and when it is on the ground. At the conclusion of the profile, the crew taxis in, parks the aircraft, goes through the complete power-down procedure, then disembarks and walks down the hall to a conference room for post-flight debriefing. During the debriefing, which may last up to two hours, the pilots comment on controls and anomalies. These comments add another perspective to the volumes of data stored by the data acquisition system which monitors the SIL aircraft continuously during the course of the test.

Torturing the aircraft

The activities of the SIL have evolved as the 777 has moved through the development cycle. The chronological sequence of tests conducted in the SIL shows the many milestones involved in moving a modern aircraft from the design stage to first-flight readiness.

Wiring checkout (4/93- 9 /93); Testing the first production set of cables, the lab found over 100 problems, which were corrected in subsequent production runs.

Power-up Testing or Initial Integration (10/93 - 11/93); Using the same procedures and check-lists that a ground crew would use on a regular flight, lab personnel powered-up actual aircraft systems in the lab until they achieved a full-airplane power-up.

Functional Test Validations (11/93 - 12/93); During the manufacturing functional test phase, suppliers of various systems for the 777 came into the lab, hooked up their Automatic Test Equipment, and tested the systems in the SIL just as they would test an airplane

on the factory floor. These activities not only verified that the systems in the SIL functioned properly, they also helped factory technicians debug their test processes and equipment.

Developmental and Initial Systems Validation (1/94 - 3/94); These activities led to early identification and correction of over 650 airplane problems.

Systems Validation, the Gauntlet (4/94 - 6/94); During the period leading up to the first test flights of the 777, the SIL put aircraft zero through the "gauntlet." Activities included "torturing the aircraft," faulting pieces of equipment so systems would reconfigure automatically, going through non-routine procedures, getting a good end-to-end check of all systems.

Validation of Aircraft Maturity and Service Operations (6/94 - Present); Since the first flights, the SIL has been working on system upgrades, software verification, safety testing, maintenance manual validation, and training of customer-service personnel.

By working three shifts and seven day weeks, the SIL has logged some 3,800 ground-test hours, 1,400 flight-test hours, and 17,600 test conditions. The lab has also developed over 700 test plans and familiarized more than 40 pilots with 777 systems.

The Electrical Power Systems Lab -- integrating systems from multiple vendors

The electrical power generation and distribution system of the 777 has many levels of redundancy designed to ensure safe operation of the aircraft under any foreseeable condition. Equipment comes from a variety of vendors who supply the aircraft’s six in-flight-operable electrical generators, as well as electrical control systems, power supply units, batteries, battery chargers and electrical load distribution panels. Each of these systems is fully checked out by its supplier, but the first time all of it comes together for a real-world workout is in the

Electrical Power Systems Lab (EPSL).

The EPSL provides the laboratory test environment for verification, validation and certification testing of the aircraft electrical power generation and distribution system. Much of the testing done in the EPSL is required for FAA certification. Typical FAA tests involve power transfers, voltage regulation, protective functions and system indications for normal and abnormal conditions.

To ensure that the lab test environment represents the aircraft, production aircraft generators are driven by 800 hp electrical motor drives which duplicate engine speeds and acceleration rates. The generators are connected via production quality feeders to the actual electrical control and distribution equipment. Loads for testing the electrical power system are provided by 24 alternating current load banks rated at 75 kva each, 16 direct current load banks rated at 100 amps each, and various aircraft fans and motor loads.

To support certification testing, the main emphasis of the lab is on monitoring system performance. To that end, the lab uses a high-speed digital data acquisition system from DSP Technology. The DSPT system is capable of monitoring 128 channels of data. It provides a variety of data traces, including voltage and current parameters for measuring operating characteristics, and breaker and switch positions for timing information. System messages are recorded using various ARINC 629 monitoring tools and/or the aircraft's avionics equipment. Although Boeing has built electrical test rigs for each new generation of aircraft, the 777 test rig is the first to use a digital data acquisition system as its primary data acquisition tool.

Testing Under Abnormal Conditions

A significant advantage of realistically integrating the aircraft's electrical power system in a laboratory environment is that tests can be performed under very abnormal conditions such as open circuits or

short circuits. The lab monitors system responses such as fault-clearing times, current spikes and voltage sags. Data and error messages are studied to verify that the system operated correctly, that the fault was isolated properly, and that the failure mode has been properly identified and documented.

"Since we purchase equipment from multiple suppliers, we think it makes a lot of sense to put everything together in our lab," said Carl Tenning, Design Supervisor of the EPSL. "The military probably has a similar facility and generator manufacturers have similar capabilities. But no other airframe manufacturer does integration testing on production hardware with our degree of fidelity, except on the airplane itself."

Conclusion

Although testing for FAA certification is an important part of the work of the Electrical Power Systems Lab, the vast majority of tests conducted at the IASL are not mandated by governmental agencies. "They're deemed necessary by Boeing to satisfy ourselves and our customers," said Art Fanning. Out of 130 different kinds of tests performed at the IASL, about a dozen involve certification. Boeing conducts the other tests because the company sees three principal benefits from integrated, on-ground testing before trial flights begin.

The first benefit of building the first of a new generation of aircraft in the lab is that the design is completed earlier. In a "system of systems" as complex as a new airplane, there are bound to be places where the design is not complete. Frequently, this occurs where systems meet each other. While building "airplane zero," lab personnel can identify these areas early on, properly complete the design and feed the information forward in time to meet the production schedule for the first airplanes. The second benefit is that once airplane zero is built, it can be used to confirm that the design functions as intended in normal operations. If it doesn't, there is more opportunity to correct the design and

incorporate those changes in the first production airplanes. Finally, once the systems function properly in normal operations, test engineers can introduce abnormalities. They can induce a problem in one box and study the interactions throughout the aircraft, making sure that one fault does not initiate a series of other failures.

"The real purpose of the IASL is to see the systems working together in the lab before they need to come together on the first airplane," said Art Fanning. "If there are problems, this gives us the opportunity to correct them much earlier. The result is that our flight tests are much more productive because we don't use valuable test flight time fixing mundane things that should have been corrected elsewhere. And our first airplanes incorporate the design changes needed to make the aircraft service-ready on the day of delivery."

####

U.S. BUSINESS NEWS

Streamlined Plane Making

Boeing , Airbus Look to Car Companies' Methods to Speed Up Jetliner Production

By DANIEL MICHAELS and J. LYNN LUNSFORD
Staff Reporters of THE WALL STREET JOURNAL
April 1, 2005; Page B1

Bitter rivals Airbus and Boeing Co. don't agree on much, but these days their production gurus chant a common mantra: Let's copy Toyota, the company that reinvented car making.

The giant jet makers and their suppliers are going back to school to learn about efficient production from companies that churn out vehicles that are just a fraction of a plane's size and complexity. Cutting production costs and speeding assembly is a vital step in the Airbus-Boeing duel to stay competitive. Each is scrambling to hold down prices and increase sales to airlines, even as carriers' profits continue to wallow in the worst aviation crisis ever.

The new efficiency focus faces limits, though, in part because airlines demand far more customization than car buyers do. Safety regulations are also much more numerous and onerous for jetliners than cars. But the plane makers have recently grasped a big truth long ago pioneered by Toyota Motor Corp.: Working hard to keep things simple saves tons of money.

So airplane people are now designing parts with an eye to how fast they can be assembled. Both Boeing and Airbus have slashed their parts inventories, copied the way car makers organize factories and trimmed production times. Airlines, long treated like royalty, are getting less choice in the options available to them, from plane colors to cockpit layouts. Boeing has also managed to apply to gargantuan planes a technique long ago adopted by Henry Ford: assembly lines.

Not long ago, the idea of aping a mass-market car manufacturer seemed preposterous to aerospace people. Compared with a jetliner, cars are low-tech,

COMPANIES

Dow Jones, Reuters

Boeing Co. (BA)

PRICE	58.38
CHANGE	-0.08
U.S. dollars	11:00 a.m.

Toyota Motor Corp. ADS (TM)

PRICE	74.85
CHANGE	0.47
U.S. dollars	10:59 a.m.

Goodrich Corp. (GR)

PRICE	38.23
CHANGE	-0.06
U.S. dollars	10:59 a.m.

United Technologies Corp. (UTX)

PRICE	101.12
CHANGE	-0.54
U.S. dollars	10:59 a.m.

DaimlerChrysler AG (DCX)

PRICE	44.12
CHANGE	-0.60
U.S. dollars	10:59 a.m.

* At Market Close

RELATED INDUSTRIES

cheap and puny. Car makers churned out millions of light vehicles last year, each priced in the tens of thousands of dollars. Annual output at Boeing and Airbus together was just 605 planes, some priced at almost $200 million. Car models change every few years, while jetliner models change over decades.

FOUR MILLION PARTS

In the past, Airbus and Boeing designed every piece of their planes. Now they still design whole jetliners, but leave engineering and production of many important components to subcontractors.

"We always thought airplanes were different because they had four million parts," says Alan Mulally, head of Boeing's commercial aircraft division, compared with the 10,000 or so parts in a car. "Well, airplanes aren't different. This is manufacturing."

The plane makers are getting a hand from suppliers that cut their teeth making parts for car companies, such as Goodrich Corp. and Britain's GKN PLC, a recent entry in the aerospace industry after decades spent making complex auto parts. GKN, in fact, has moved key automotive staff over to aviation projects.

Airbus is now squeezing suppliers in a bid to cut more than $1.29 billion from its cost base by 2006, says its chief operating officer, Gustav Humbert. The technique, he says, is "continuous improvement" -- an approach first promoted at Toyota.

Airbus aims by 2006 to build a single-aisle plane from scratch in just six months, half the time taken in 2003, and wide-body planes in 12 months, a 20% reduction. Working faster means Airbus can produce more planes at its existing factories, and it expects to free up more than $1.3 billion in cash by shortening the time it keeps its parts in stock -- another economy pioneered by car makers.

As Boeing prepares to build its proposed fuel-efficient 787 Dreamliner jet, project manager Mike Bair wants to make the plane so modular that the last stage of assembly takes just three days. Its predecessor model, the 767, took up to a month to assemble. Because individuality sends costs soaring, he is also emulating car companies by offering a standard set of features on the 787. Customers can no longer dictate cockpit layouts and have limited choice on items such as electronics and interiors. Boeing, which once offered more than a dozen shades of white paint, now offers just two.

Not so long ago, every jetliner was custom-built. Airbus, which began its business in the 1970s, several decades after Boeing, from the start inched toward the auto-making model by building airplanes in sections, such as wings and cockpits. Final assembly became essentially a high-tech version of snapping together a plastic model airplane.

But in the past few years, Airbus and Boeing have pushed the process much further and started outsourcing entire components, just as car makers outsource systems like transmissions. Rather than give a contractor blueprints to fabricate a part, they have begun offering only general requirements and asked suppliers to propose designs -- another key to the Toyota system.

Hamilton Sundstrand, a division of United Technologies Corp., is designing and producing the critical cabin air-conditioning and temperature-control system for both the double-decker Airbus A380 and the 787. Jamco Corp. of Japan is designing and producing key structural elements made of advanced composite materials for the A380.

On shop floors, the plane makers have also learned from car makers. At an Airbus factory in Wales, where it builds wings, production teams used to walk far to the stockroom for bags of bolts and rivets, and frequently left them scattered about -- a wasteful and unsafe practice -- because they lacked nearby storage. Airbus, in fact, checked out production procedures at car maker DaimlerChrysler AG, which has a stake in the plane

maker's parent company, European Aeronautic Defence & Space Co.

Using work-analysis methods developed by the auto industry, project teams studied which fasteners were needed where, and when, and then organized racks on the shop floor. Now, carefully labeled bins contain tidy sets of supplies needed for specific tasks. The change has sped up work and saved over $100,000 in rivets and bolts at the Welsh factory alone, Airbus says. Boeing has made similar changes at its Seattle-area plants.

GKN has adopted a similar auto-industry innovation and eliminated many warehouses at the English plant where it builds parts for Canada's Bombardier Inc., a maker of regional jets. Now, suppliers ship directly to the shop floor and restock as necessary. This "frees up tons of capital" from inventories, says Rob Soen, the European head of procurement for GKN Aerospace and a 23-year veteran of the car industry.

But the jet makers aren't moving wholesale to car-like production methods. Swapping one production method for another can require huge investments in new equipment and staff training, and often requires shutting down a production hall to make the switch. That's a big reason why aviation manufacturers have moved slowly.

Airbus, for example, moves planes through successive stations during assembly but still maintains much of the old piecework approach. Production managers say this gives flexibility because a glitch that slows one plane won't stall a whole assembly line.

Boeing, though, made one of the most dramatic production changes yet in 2001 when it began putting together planes on a huge moving line -- à la Henry Ford and his Model T. The motion "lent a sense of urgency to the process that we really didn't have when the planes were sitting still," says Carolyn Corvi, the executive who oversaw the change. Ms. Corvi and other top Boeing executives made multiple visits to Toyota when they were first beginning to study how to convert the production process to a moving line.

When many workers initially balked at the production line and unions filed complaints, Boeing took extra pains to win them over. The change paid off: Boeing halved the time it takes to assemble a single-aisle 737, and has started putting its other planes -- including its oldest and largest product, the 747 -- on moving lines.

Write to Daniel Michaels at daniel.michaels@wsj.com and J. Lynn Lunsford at lynn.lunsford@wsj.com

ASSIGNMENT 2

manager

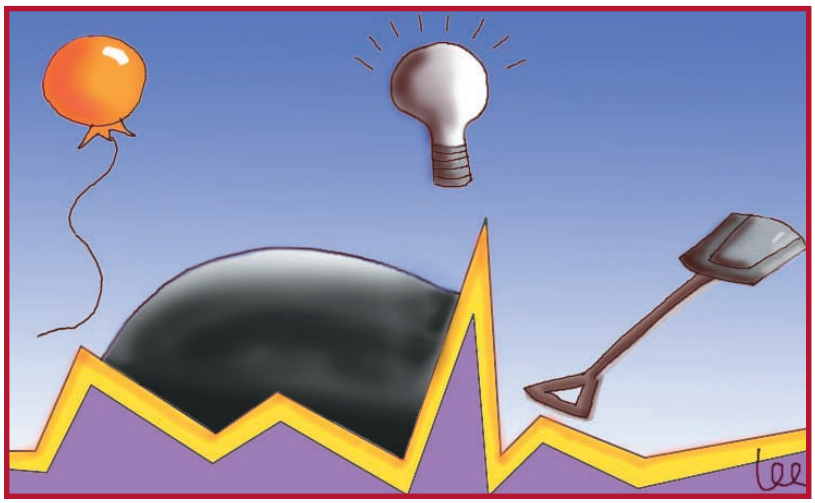
Editor: Donald J. Reifer ■ Reifer Consultants ■ d.reifer@ieee.org

Eight Lessons Learned during COTS-Based Systems Maintenance

Donald J. Reifer, Victor R. Basili, Barry W. Boehm, and Betsy Clark

In the May 2001 issue of *Computer*, two of us published an article called "COTS-Based Systems Top 10 List" (pp. 91–93). The list identified 10 hypotheses that served as challenges for enhancing our empirical understanding of commercial off-the-shelf software. These hypotheses were primarily related to COTS-based systems development.

CBSs remain one of three focus areas (the other two are defect reduction and agile methods) of both our US Federal Aviation Agency-sponsored research and our National Science Foundation-sponsored Center for Empirically Based Software Engineering (CeBASE) efforts. As our original article said, COTS software usage remains relatively immature as it progresses through a peak of inflated expectations, a trough of disillusionment, a slope of enlightenment, and a plateau of productivity. Based on presentations at the recent 2nd International Conference on COTS-Based Software Systems (*ICCBSS 2003 Proceedings*, Springer-Verlag), risks inherent to COTS use are large—as are the potential returns. Pursuing custom solutions remains unattractive primarily because it takes so much time and effort to develop software products.



We have recently extended the Top 10 list of challenges in the original *Computer* article to encompass the life cycle's maintenance phase. During maintenance, COTS products undergo a technology refresh and renewal cycle. As part of this activity, maintainers decide whether to upgrade their COTS products or retain old versions. If they choose to retain old versions, they'll eventually reach the point where the vendor no longer supports those versions. If they choose to update, they must synchronize the associated update with their release cycle and with product updates other vendors are making. They must also coordinate the update of wrappers and glue code so that they will work with the new versions.

Because COTS maintenance is relatively im-

0740-7459/03/$17.00 © 2003 IEEE

mature, we present empirical results gathered to date as lessons learned rather than as either results or hypotheses. We hope these results will help those trying to manage COTS maintenance.

Lesson learned 1

The refresh and renewal process for CBSs must be defined a priori and managed so COTS package updates can be synchronized with each other and the organization's release and business cycle. If they aren't, updates might occur sporadically during the maintenance part of the cycle and the risk of technology obsolescence might increase dramatically.

Source. A recent US Air Force Scientific Advisory Board study (SAB-TR-99-03, April 2000) surveyed 34 COTS-based systems to look at COTS software management within weapon systems.

Implications. Currently, few COTS software lifecycle models address CBS maintenance processes. Guidance is needed to define refresh and renewal process activities. We must also define criteria for making decisions regarding when to incorporate updates within releases, along with those criteria's associated risks and business implications.

Lesson learned 2

COTS software capability and quality evaluation must be managed as a continuing task during the maintenance phase.

Source. Most publications that discuss CBS processes advocate that companies establish a market watch function (see lessons-learned papers in COTS workshops such as ICCBSS 1 and 2 and in research reports by the National Research Council of Canada, the Software Engineering Institute, and the University of Southern California).

Implications. Most COTS software studies recommend that firms not only establish a market watch function to keep track of where their packages are heading but also that they continuously assess their options. A market watch looks at the marketplace as a whole, monitoring a specific vendor's health and viability as well as what competitors are coming out with. COTS evaluation gives you a detailed assessment of package capabilities, quality issues, and future options. It typically involves conducting some form of operational demonstration.

Lesson learned 3

The cost to maintain COTS-based systems equals or exceeds that of developing custom software. Maintenance in this context involves updating CBSs with new releases, modifying wrappers and glue code, and incorporating fixes and repairs into the system.

Source. Reifer Consultants recently studied the cost of COTS software across 16 systems, some of which employ over 40 different packages, across three large firms. Costs average 10 percent of the development cost per year over a projected 10-year life for the system. Although releases occur every year, COTS technology refreshes occur every two years or across two releases. Defect rates per release for CBSs are poorer than for custom-built software, averaging 10 to 40 percent higher.

Implications. Even though firms can save time and effort during development using CBSs, they should evaluate the total lifecycle cost of options prior to making commitments. Such analysis could identify risks that negate many of the advantages that CBSs bring to the table. For example, firms must coordinate glue code updates along with package improvements. Considering that a line of glue code costs, on average, three times that of a line of custom code to develop and maintain, maintenance effort can get quite expensive. In situations where CBSs have a long life, custom solutions might work out to be cheaper than COTS alternatives. Project managers whom RCI interviewed also said that, unlike custom systems, COTS-based systems need a continual stream of funding throughout their life cycle. Such funding is necessary to keep up with a dynamic marketplace in which vendors are continually releasing new versions. Funding was an issue with several of the projects in this study because their maintenance budgets often get cut. The managers believe that this hurts a CBS more than a custom system because they can delay maintenance on the latter if they have to because of budget limitations.

The cost to maintain COTS-based systems equals or exceeds that of developing custom software.

Lesson learned 4

The most significant variables that influence the lifecycle cost of COTS-based systems include the following (in order of impact):

- Number of COTS packages that must be synchronized within a release
- Technology refresh and renewal cycle times
- Maintenance workload (the amount of effort software engineers expend to handle the task at hand) for glue code and wrapper updates
- Maintenance workload to reconfigure packages
- Market watch and product evaluation workload during maintenance
- Maintenance workload to update databases
- Maintenance workload to migrate to new standards
- COTS maintenance license costs

Source. The RCI study mentioned earlier was a source here also. The study identified these parameters using

a survey that asked those responsible for maintenance for insight. The number of packages requiring synchronization was twice as sensitive as the need to migrate to new standards.

Implications. Cost models like USC's COCOTS (see http://sunset.usc.edu for information) should be updated to encompass the full CBS life cycle. Currently, they focus on estimating the costs associated with evaluating, adapting, and deploying COTS software packages during development and maintenance. In the future, such models should incorporate additional variables such as the last three bullets on our list to permit those assessing lifecycle costs to estimate the full cost of the maintenance portion of the CBS life cycle.

Lesson learned 5

Maintenance complexity (and costs) will increase exponentially as the number of independent COTS packages integrated into a system increases.

Source. USC's COCOTS team has initial results of a study of 20 projects.

Implications. Projects should understand the maintenance implications of integrating a large number of COTS products into a system. In addition to the effort involved in the initial integration, they should consider that each product will evolve in its own way, according to different timetables, at the vendors' discretion. They will have to expend considerable effort to handle these products' continuing evolution (for example, understanding the impact of an upgrade on the rest of the system or making changes to glue code.

Lesson learned 6

Software engineers must spend significant time and effort up front to analyze the impact of version updates (even when the decision is made not to incorporate the updates).

Source. Initial results of the COCOTS team's 20-project study suggest that analysis efforts during maintenance directed toward updates can tax the organization severely. This is particularly true for safety-critical systems.

Implications. Maintenance modeling must assume that CBSs incur fixed and variable costs. Fixed costs are those associated with market watch and continued product evaluation. Variable costs are a function of the work performed to incorporate updates, fixes, changes, and optimizations into the impending release. The workload performed by the fixed staff must be optimized (balanced) as part of this process.

Lesson learned 7

Flexible CBS software licensing practices lead to improved performance, reliability, and expandability.

Source. RCI performed surveys in 2000 and 2001 on best acquisition practices for the US Army (see www.reifer.com for a paper on innovative licensing).

Implications. The studies identified partnering instead of conflict management as the preferred approach to licensing. Shared goals lead to products with improved "goodness of fit" and "functionality" for the buyer. Leveraging relationships to achieve shared goals is highly desirable. Innovative contracting under such arrangements lead to deep volume discounts and priority service and bug fixes. Traditional approaches to licensing, where contracts instead of relationships govern, lead to distrust and poor results.

Traditional approaches to licensing, where contracts instead of relationships govern, lead to distrust and poor results.

Lesson learned 8

Wrappers can be effectively used to protect a CBS from unintended negative impacts of version upgrades.

Source. Several projects were interviewed for the COCOTS database. One project successfully used wrappers for information hiding so that different versions of COTS products (or different products) could be swapped without affecting the rest of the system.

Implication. CBS architectures should accommodate COTS changes throughout the system life cycle.

To make better decisions relative to CBSs, we need empirical knowledge. To gain this knowledge, we must more fully understand the lifecycle processes people use when harnessing COTS packages. The initial findings reported here are but the first step in our attempts to capture this empirical knowledge. We plan to continue collecting data and investigating the phenomenology of COTS-based systems. This work complements the more general results available at our CeBASE Web site (http://cebase.org/cbs) and the SEI Web site (www.sei.cmu.edu/cbs). We welcome your comments, input, and contributions.

Donald J. Reifer is a visiting associate with the Center for Software Engineering at the University of Southern California and president of Reifer Consultants Inc. Contact him at dreifer@earthlink.net.

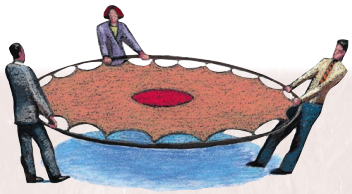
Victor R. Basili is a professor in the Computer Science Department at the University of Maryland and director of the Fraunhofer Center–Maryland. Contact him at basili@cs.umd.edu.

Barry W. Boehm is director of the Center for Software Engineering at the University of Southern California. Contact him at boehm@sunset.usc.edu.

Betsy Clark is president of Software Metrics and works with USC on several CBS projects. Contact her at betsy@software-metrics.com.

COTS Integration: Plug and Pray?

Barry Boehm and Chris Abts,
University of Southern California



The advantages of COTS products can be undermined by poor support and changing feature sets. Learn the common COTS pitfalls and how to avoid them.

For most software applications, the use of commercial off-the-shelf products has become an economic necessity. Gone are the days when upsized industry and government information technology organizations had the luxury of trying to develop—and, at greater expense, maintain—their own database, network, and user-interface management infrastructure. Viable COTS products are climbing up the protocol stack, from infrastructure into applications solutions in such areas as office and management support, electronic commerce, finance, logistics, manufacturing, law, and medicine. For small and large commercial companies, time-to-market pressures also exert a strong pressure toward COTS-based solutions.

COTS PLUSSES AND MINUSES

However, most organizations have also found that COTS gains are accompanied by frustrating COTS pains. Table 1 summarizes a great deal of experience on the relative advantages and disadvantages of COTS solutions.[1] One of the best COTS integration gain-and-pain case studies[2] summarizes the experiences of David Garlan's group at CMU. Garlan's group tried to integrate four COTS products into the Aesop software architecting environment—the OBST object management system, the Mach RPC Interface Generator, the SoftBench tool integration framework, and the InterViews user interface manager—only to find a number of architectural mismatches among the products' underlying assumptions. For example, three of the four products are event based, but each has different event semantics, and each assumes it is the sole owner of the event queue. Resolving such model clashes escalated the original two-person, six-month project into a five-person, two-year project: a factor of four increase in schedule and a factor of five increase in effort.

Such experiences, and the more general technical and business issues shown in Table 1, indicate that COTS integration differs significantly from traditional software development and requires significantly different approaches to its management. At the USC Center for Software Engineering COTS Integration Affiliates' Workshop, we identified four key COTS integration issues: functionality and performance, interoperability, product evolution, and vendor behavior.

FUNCTIONALITY AND PERFORMANCE

You have no control over a COTS product's functionality or performance. If you modify the source code, it's not really COTS—and its future becomes your responsibility. Even as black boxes, big COTS products have formidable complexity; Windows 95, for example, has roughly 25,000 entry points.

Let the buyer beware

COTS-based projects cannot make blanket assumptions about system requirements or embrace traditional process models. One mistake developers can make is to use the waterfall model on a COTS integration project. With the waterfall model, you specify requirements and these determine the system's capabilities. With COTS products, it's the other way around: The capabilities determine the "requirements," or delivered-system features. If your users have a "requirement" for a blinking cursor and the best COTS product doesn't provide it, you're out of luck.

Another potential danger involves using evolutionary development with the assumption that every undesired feature can be changed to fit your needs. COTS vendors do change features, but in response to the overall marketplace, not to individual users' needs. It's also unwise to assume that advertised COTS capabilities are necessarily real. COTS vendors may have had the best of intentions when they wrote the marketing literature, but that won't help you when the advertised feature isn't there.

Barry Boehm, Computer Science Department, University of Southern California, Los Angeles, CA 90089; boehm@sunset.usc.edu

Table 1. COTS advantages and disadvantages.

Advantages	Disadvantages
Immediately available; earlier payback	Licensing, intellectual property procurement delays
Avoids expensive development	Up-front license fees
Avoids expensive maintenance	Recurring maintenance fees
Predictable, confirmable license fees and performance	Reliability often unknown or inadequate; scale difficult to change
Rich functionality	Too-rich functionality compromises usability, performance.
Broadly used, mature technologies	Constraints on functionality, efficiency
Frequent upgrades often anticipate organization's needs	No control over upgrades and maintenance
Dedicated support organization	Dependence on vendor
Hardware/software independence	Integration not always trivial; incompatibilities among vendors
Tracks technology trends	Synchronizing multiple-vendor upgrades

Use risk-driven process models

Given the vagaries of requirements in COTS-based software development, developers should adopt or modify more dynamic, risk-driven process models, such as risk-driven spiral-type process models. Assess risks via prototyping, benchmarking, reference checking, and related techniques. Focus each spiral cycle on resolving the most critical risks. The Raytheon "Pathfinder" approach—using top people to resolve top risk items in advance—is a particularly effective way to address these and other risks.

You should also perform the equivalent of a "receiving inspection" upon initial COTS receipt. This practice ensures that the COTS product really does what it is expected to do. Keep requirements negotiable until the system's architecture and COTS choices stabilize. This prevents you from promising features and capabilities that the system cannot support easily—or at all. Finally, involve all key stakeholders in critical COTS decisions. These stakeholders can include users, customers, developers, testers, maintainers, operators, or others as appropriate.

INTEROPERABILITY

Most COTS products are not designed to interoperate with each other. The Garlan experience[2] provides a good case study and explanation for why interoperability problems can cause COTS integration cost and schedule overruns.

Roadblocks to interoperation

If you commit prematurely to incompatible combinations of COTS products, you lessen the chance they will operate with your own software. This situation can happen in many ways: through haste, desire to show progress, politics, short-term emphasis on rapid application development, or an uncritical enthusiasm for features or performance.

Trying to integrate too many incompatible COTS products can also cause problems. As Garlan[2] shows, four such products can be too many. In general, trying to integrate more than a half-dozen COTS products from different sources should place this item on your high-risk assessment list. Nor should you defer COTS integration till the end of the development cycle. Doing so puts your most uncontrollable problem on your critical path as you approach delivery.

Finally, avoid committing to a tightly coupled subset of COTS products with closed, proprietary interfaces. Such products restrict your downstream options; once you're committed, it's hard to back out.

Ensure smooth interconnections

To make sure your in-house software works with the COTS products you purchase, use the Life Cycle Architecture milestone[3] as an anchor point for your development process. In particular, include demonstrations of COTS interoperability and scalability as risks to be resolved and documented in the feasibility rationale. Use the AT&T/Lucent Architecture Review Board (ARB) best commercial practice[4] at the Life Cycle Architecture milestone. Over a 10-year period, AT&T documented at least a 10 percent savings from using ARBs.

Finally, strive to achieve open architectures and COTS substitutability. In the extremely fast-moving software field, the ability to adapt rapidly to new best-of-breed COTS products is critical.

PRODUCT EVOLUTION

You have no control over a COTS product's evolution, which responds only to the overall marketplace. Upgrades are frequently not upwardly compatible; old releases become obsolete and unsupported by the vendor. If COTS architectural mismatch doesn't get you initially, COTS architectural drift can easily get you later. Our Affiliates' experience indicates that complex COTS-intensive systems often have higher software maintenance costs than do traditional systems—for example, when the application is relatively stable and the COTS products are relatively volatile. Good practices, such as batching COTS upgrades in synchronized releases, can lower these costs.

Evolutionary dead ends

"Snapshot" requirements specifications and corresponding point-solution architectures make it too difficult to evolve your software. These are bad practices for traditional systems; with uncontrollable COTS evolution, the maintenance headaches become even worse. If you understaff maintenance personnel or undertrain them in COTS adaptation, you invite long-term problems. Integrating COTS products with your own software, then maintaining the combined system, is a challenging task under the best circumstances; without proper training, that challenge becomes significantly greater.

Tightly coupled, independently evolving COTS products cause an increase in maintenance overhead. Using just two such products will make your system's maintenance difficult; using more than two will make the problem much worse. Finally, it's wrong to assume that uncon-

trollable COTS product evolution is just a maintenance problem: It can attack your development schedules and budgets as well.

Nurture beneficial mutations

You can offset the costs and risks associated with long-term COTS product evolution by sticking with dominant open commercial standards. These standards make COTS product evolution and substitutability more manageable.

For COTS product selection criteria, use likely future system and product line needs (evolution requirements) as well as current needs. These evolution criteria can involve portability, scalability, distributed processing, user interface media, and various kinds of functionality growth. Use flexible architectures that facilitate adaptation to change. Strong choices include software bus, encapsulation, layering, and message- and event-based architectures.

Carefully evaluate COTS vendors' track records with respect to product-evolution predictability. Widely varying feature sets, too-frequent updates, and dramatic shifts in product capabilities can cause problems in the long term.

Finally, establish a proactive system-release strategy, synchronizing COTS upgrades with system releases. Planning the ongoing integration of evolving COTS products with your own internally developed software helps ensure that both continue to function harmoniously.

VENDOR BEHAVIOR

COTS vendor behavior varies widely with respect to support, cooperation, and predictability. Given the three major sources of COTS integration difficulty we've already cited, an accurate assessment of a COTS vendor's ability and willingness to help with these areas is tremendously important. The workshop identified a few assessment heuristics that proved helpful, such as that the value of a COTS vendor's support follows a convex curve with respect to the vendor's size and maturity. Specifically, according to our Affiliates' experience, mid-sized companies occupy the highest arc of the curve and usually provide the best support because, as a recent radio commercial generalizes, "Small companies are too small to help; big companies are too big to care."

Perishable promises

Beware of uncritically accepting vendors' statements about their COTS products' capabilities and support. Shifting markets, mergers and buyouts, or unforeseen technological developments can convert a vendor's best intentions into broken promises. A lack of fallbacks or contingency plans can also derail your project. Any project that doesn't allow for such contingencies as product substitution or escrow of a failed vendor's product is one that courts disaster.

Foster realistic expectations

To ensure that you establish the best vendor relationships possible, you must perform extensive evaluation and reference-checking of a COTS vendor's advertised capabilities and support track record. Web searches, interviews with the vendor's other clients, and industry publications can all help determine a given vendor's credibility.

Because COTS products are likely to remain an essential component of your software for a long time, it can help to establish strategic partnerships or other incentives for COTS product vendors to provide continuing support. These incentives can include financial assistance, early experimentation with a new COTS vendor's capabilities, and sponsored COTS product extensions or technology upgrades. Further protect yourself by negotiating and documenting critical vendor support agreements. You can establish a "no surprises" relationship with vendors by determining, in advance, exactly what's expected of both parties.

Sources and Resources

The best source I know for COTS integration information is the CMU Software Engineering Institute's Web page on its COTS-Based Systems (CBS) initiative at http://www.sci.cmu.edu/cbs/cbs_description.html. The USC-CSE Web page for the Constructive COTS Integration (COCOTS) cost estimation model has pointers to numerous COTS integration information sources. It's at http://sunset.usc.edu/COCOTS/cocots.html.

The following five major recent books on software reuse offer valuable perspectives on integrating reusable components in general, of which COTS is a special case.

I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse*, Addison Wesley Longman, Reading, Mass., 1997.

W. Lim, *Managing Software Reuse*, Prentice Hall, Upper Saddle River, N.J., 1998.

J. Poulin, *Measuring Software Reuse*, Addison Wesley Longman, Reading, Mass., 1997.

D. Reifer, *Practical Software Reuse*, John Wiley & Sons, New York, 1997.

W. Tracz, *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*, Addison Wesley Longman, Reading, Mass., 1995.

The Lim and Reifer books offer the most COTS-specific insights.

The upcoming 1999 International Conference on Software Engineering (ICSE '99) in Los Angeles, May 16-22, 1999, includes a COTS Integration industry experience session that features Lockheed Martin's Dorothy McKinney and Texas Instruments' Marie Silverthorn, with the SEI's Tricia Oberndorf as discussant. Among ICSE '99's architecture experience case studies is an insightful paper by David Barstow on how his sports information software's architecture evolved in response to the rapid evolution of Netscape and related COTS products. The associated Symposium on Software Reuse, May 21-23, will have a specific reuse focus that includes COTS integration. See the ICSE '99 Web site at http://sunset.usc.edu/icse99/index.html.

By accelerating the speed with which new software products can reach their users, while simultaneously offsetting up-front development costs, COTS products have become an increasingly attractive option for budget-conscious software developers. But these advantages bear a sometimes hidden price tag. In the short term, differences between the underlying design of an organization's internally developed software and that of the COTS products it chooses to integrate with that software can cause unforeseen problems. In the long term, fluctuating COTS developer support and the unpredictable evolution of the COTS product itself can hobble any organization that incorporates that product into its software, crippling the hybrid system or even rendering it completely unusable.

To guard against such grim situations, approach any COTS integration project clear-eyed and wary. By following the recommendations we've given in this article, you can go a long way toward maximizing COTS' advantages and protecting yourself against its more expensive dangers. ❖

Barry Boehm developed the Constructive Cost Model (Cocomo), the software process Spiral Model, and the Theory W (win-win) approach to software management and requirements determination.

Chris Abts is developing the constructive COTS Integration (COCOTS) cost model. Contact him at cabst@sunset.usc.edu.

References

1. National Research Council, *Ada and Beyond: Software Policies for the Department of Defense*, National Academy Press, Washington, D.C., 1997.
2. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse is So Hard," *IEEE Software*, Nov. 1995, pp. 17-26.
3. B. Boehm, "Anchoring the Software Process," *IEEE Software*, July 1996, pp. 73-82.
4. J. Marenzano, "System Architecture Review Findings," D. Garlan (ed.), *Proc. ICSE 17 Architecture Workshop*, Carnegie Mellon Univ., Pittsburgh, 1995.

Software Engineering Metrics for COTS-Based Systems

The growing reliance on commercial off-the-shelf components for large-scale projects emphasizes the need for adequate metrics to quantify component quality.

Sahra Sedigh-Ali
Arif Ghafoor
Purdue University

Raymond A. Paul
US Department of Defense

The paradigm shift to commercial off-the-shelf components appears inevitable, necessitating drastic changes to current software development and business practices. Quality and risk concerns currently limit the application of COTS-based system design to noncritical applications. New approaches to quality and risk management will be needed to handle the growth of CBSs.

Our metrics-based approach and software engineering metrics can aid developers and managers in analyzing the return on investment in quality improvement initiatives for CBSs. These metrics also facilitate the modeling of cost and quality, although we need more complex models to capture the intricate relationships between cost and quality metrics in a CBS.

COTS COMPONENTS

With software development proceeding at Internet speed, in-house development of all system components may prove too costly in terms of both time and money. Large-scale component reuse or COTS component acquisition can generate savings in development resources, which can then be applied to quality improvement, including enhancements to reliability, availability, and ease of maintenance.

Prudent component deployment can also localize the effects of changes made to a particular portion of the application, reducing the ripple effect of system modifications. This localization can increase system adaptability by facilitating modifications to system components or integration code, which are necessary for conforming to changes in requirements or system design.

COTS component acquisition can reduce time to market by shifting developer resources from component-level development to integration. Increased modularity also facilitates rapid incremental delivery, allowing developers to release modules as they integrate them and offer product upgrades as various components evolve.

These advantages bring related disadvantages, including integration difficulties, performance constraints, and incompatibility among products from different vendors. Further, relying on COTS components increases the system's vulnerability to risks arising from third-party development, such as vendor longevity and intellectual-property procurement. Component performance and reliability also vary because component-level testing may be limited to black-box tests, and inherently biased vendor claims may be the only source of information.[1]

Such issues limit COTS component use to noncritical systems that require low to moderate quality. Systems that require high quality cannot afford the risks associated with employing these components.

METRICS FOR COTS-BASED SYSTEMS

In deciding between in-house development and COTS component acquisition, software engineers must consider the anticipated effect on system quality. We can define software quality in several ways:

- satisfaction level—the degree to which a software product meets a user's needs and expectations;
- a software product's value relative to its various stakeholders and its competition;

0018-9162/01/$10.00 © 2001 IEEE

- the extent to which a software product exhibits desired properties;
- the degree to which a software product works correctly in the environment it was designed for, without deviating from expected behavior; and
- the effectiveness and correctness of the process employed in developing the software product.[2]

Quality factors and quality metrics

Norman Schneidewind discusses quality in terms of quality factors and quality metrics.[3] He defines a quality factor as "an attribute of software that contributes to its quality, where quality is the degree to which software meets customer or user needs or expectations." For example, Schneidewind mentions reliability as a quality factor. Direct measurement of quality factors is generally not feasible, so we often measure them indirectly—for example, by counting the number of failures reported for a particular module.

Schneidewind defines a quality metric as "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that may affect its quality." In contrast with quality factors, which are user-oriented, quality metrics are developer-oriented because developers can use them to estimate quality at a very early stage in the software development process. Before using metrics for design or integration decisions, software engineers should validate them, establishing a statistical relationship between metrics and quality factors and ensuring that the metrics provide a correct estimate of the attribute visible to the user.

In addition to traditional software metrics, COTS-based systems require metrics that capture attributes such as integration complexity and performance. Combining component-level metrics to obtain system-level indicators of quality is a challenging issue that is further complicated by COTS components' black-box nature, which masks their internal workings and restricts system developers to accessing their interfaces.

To be thorough, we should test the operational profiles of both the COTS components with respect to both their own operational profiles and that of the overall system. But when inaccessibility of the source code for some components prevents such comprehensive testing, we can use metrics to guide the software development process. In addition to metrics data, certain aspects of the software product impact and guide software development decisions:

- the system's expected functionality and the customer's requirements;
- the makeup of the various organizations involved in the project and the level of maturity and capabilities of the participating teams;
- the developers' use of innovative processes and the methods they adopt as a part of the software engineering environment to manage cost and value, including details of development process models such as the waterfall or spiral models; and
- features of the preexisting COTS components that the system will use.

The unpredictable quality of third-party software creates a unique set of risks for software systems using COTS components.

Risk management

The unpredictable quality of third-party software creates a unique set of risks for software systems using COTS components. The CBS development process, then, should include risk management, which identifies high-risk items that can jeopardize system quality and attempts to resolve them as early as possible to ensure high quality and rapid delivery. The two major steps in risk management are

- *risk assessment:* assess the probability and magnitude of loss for each risk item and prioritize risk items according to their expected loss; and
- *risk control:* generate and execute plans to resolve the risk items.

Developers apply these two steps repeatedly throughout the software development life cycle.[4]

In CBSs, risk management focuses on evaluating alternative components that meet system requirements, either selecting the component that fits best or choosing in-house development. In either of these tasks, developers or integrators can decide to relax the requirements to allow a particular choice, using risk management to determine the extent of tolerable relaxation.[5]

Risk and quality-management metrics

Metrics can guide risk and quality management, helping to reduce risks encountered during planning and execution of software development, resource and effort allocation, scheduling and execution, and product evaluation.[4] Risks can include performance issues, reliability, adaptability, and return on investment. Risk reduction can take many forms, such as using component wrappers or middleware, replacing components, relaxing system requirements, or even issuing legal disclaimers for certain failure-prone software features. Metrics let developers identify and isolate these risks, then take corrective action.

The key to success is selecting appropriate metrics—especially metrics that provide measures applicable over the entire software cycle and that address both software processes and products. In choosing metrics, developers should consider several factors:

Table 1. System-level metrics for component-based systems.

Category	Metric	Evaluates or measures
Management	Cost	Total software development expenditure, including costs of component acquisition, integration, and quality improvement
	Time to market	Elapsed time between development start and component acquisition to software delivery
	Software engineering environment	Capability and maturity of the environment in which the software product is developed
	System resource utilization	Use of target computer resources as a percentage of total capacity
Requirements	Requirements conformance	Adherence of integrated product to defined requirements at various levels of software development and integration
	Requirements stability	Level of changes to established software requirements
Quality	Adaptability	Integrated system's ability to adapt to requirements changes
	Complexity of interfaces and integration	Component interface and middleware or integration code complexity
	Integration test coverage	Fraction of the system that has undergone integration testing satisfactorily
	End-to-end test coverage	Fraction of the system's functionality that has undergone end-to-end testing satisfactorily
	Fault profiles	Cumulative number of detected faults
	Reliability	Probability of failure-free system operation over a specified period of time
	Customer satisfaction	Degree to which the software meets customer expectations and requirements

- the intended use of the metrics data;
- the metrics' usefulness and cost-effectiveness;
- the application's functional characteristics, physical composition, and size;
- the installation platform;
- the software engineering environment of the development phase;
- the software engineering environment of the integration phase; and
- the software development or maintenance life-cycle stage of both the components and the system.[6]

Table 1 shows our set of 13 system-level metrics for CBS software engineering. These metrics help managers select appropriate components from a repository of software products and aid in deciding between using COTS components or developing new components. The primary considerations are cost, time to market, and product quality.

We can divide these metrics into three categories: management, requirements, and quality.

Management. These metrics include cost, time to market, system resource utilization, and software engineering environment. Developers can use management metrics for resource planning or other management tasks or for enterprise resource planning applications.

- The cost metric measures the overall expenses incurred during the course of software development. These expenses include the costs of component acquisition and integration and quality improvements to the system.
- The time-to-market metric measures the time needed to release the product, from the beginning of development and COTS component acquisition to delivery. A modified version of this metric can evaluate the speed of incremental delivery, measuring the amount of time required to deliver a certain fraction of the overall application functionality.
- The software engineering environment metric measures the capability of producing high-quality software and can be expressed in terms of the Software-Acquisition Capability Maturity Model.[7]
- System resource utilization determines the percentage of target computer resources the system will consume.

Requirements. Developers use requirements metrics to measure the CBS's conformance and stability so they can monitor specifications, translations, and volatility, as well as the level of adherence to the requirements. COTS components are often unstable, and component-level stability can affect requirements stability if developers adapt requirements to incorporate changes to selected components.

Quality. These metrics include adaptability, complexity of interfaces and integration, integration test coverage, end-to-end test coverage, reliability, and customer satisfaction.

- Adaptability measures a system's flexibility, evaluating its ability to adapt to requirements changes, whether as a result of system redesign or to accommodate multiple applications.
- Complexity of interfaces and integration provides an estimate of the complexity of interfaces, middleware, or glue code required for integrating different COTS products. Overly complex interfaces complicate testing, debugging, and maintenance, and they degrade the system's quality.
- Integration test coverage and end-to-end test coverage indicate the fraction of the system's functionality that has completed those tests, as well as the effort testing requires.[8] Developers can use known measures to evaluate coverage, such as statement or path coverage, depending on the level of access to system source code.

- Reliability estimates the probability of fault-free system operation over a specified period of time. To obtain this metric, developers use techniques similar to the techniques they use in traditional systems, including fault injection into the integration code.
- Customer satisfaction evaluates how well the software meets customer expectations and requirements. Beta releases can help estimate predictors of customer satisfaction before final product delivery. Sample predictors include schedule requirements, management maturity, customer culture, marketplace trends, and the customer's proficiency. Such estimates can guide development decisions such as release scheduling and can aid in developing a test plan that accurately reflects the product's field use.

CBS metrics differ from traditional metrics in that they do not depend on the components' code size, which is generally not known. If developers require a size measure, they can use alternate measures such as the number of use cases—business tasks the application performs—that a given component supports.

CBS metrics also approach time to market differently. Component acquisition changes the concept of time to market because developers may not know the component development time and cannot incorporate it into time calculations. For CBSs, a simple delivery rate measure can replace the time-to-market measure. One proposed measure divides the number of use cases by the elapsed time in months.[9]

Because our metrics are interdependent, understanding the relationships between them can aid decision making regarding CBS quality-improvement investments. The most obvious relationship is between cost and quality metrics, such as reliability. However, more subtle relationships exist, such as among time to market, test coverage, and reliability. Delayed product release because of testing and debugging can result in reduced revenues or, in extreme cases, loss of the market to a competitor with an earlier release. On the other hand, premature product release can lead to lower reliability. Understanding the relationships among time to market, test coverage, and reliability can help in selecting a suitable release schedule.

Developers can combine the cost metric and the system resource utilization metric to determine whether the budget allows purchasing additional computer resources that will enhance the product's quality. Another effective strategy involves using the software engineering environment in conjunction with the quality metrics to encourage vendors to improve their software development process and adhere to standards, thus increasing the likelihood that users will select their component.

Table 2. Software quality cost categories.

Category	Typical costs of CBS software quality
Appraisal costs	Integration or end-to-end testing, quality audits, component evaluation, metrics collection, and analysis
Prevention costs	Training, software design reviews, process studies, component upgrades
Internal failure costs	Defect management, design and integration rework, component replacement, requirement relaxation
External failure costs	Technical support, maintenance, defect notification, remedial component upgrade or replacement

COST OF QUALITY

The cost of quality (CoQ) represents the resources dedicated to improving the quality of the product being developed. For example, increasing or maintaining reliability incurs costs that can be considered the costs of reliability. The overall CoQ is the sum of such costs plus other costs that we cannot directly attribute to factors that quality metrics measure. Quality costs, then, represent "the difference between the actual cost of a product or service and what the reduced cost would be if there were no possibility of substandard service, failure of products, or defects in their manufacture."[2]

We concern ourselves with the cost of software quality (CoSQ) metric—corresponding to the cost metric in Table 1—which we divide into two major types: cost of conformance and cost of nonconformance.

The cost of conformance derives from the amount the developer spends on attempts to improve quality. We can further divide conformance costs into prevention and appraisal costs. Projects incur prevention costs during activities targeted at preventing defects, such as training costs, software design reviews, and formal quality inspections. Likewise, activities that involve measuring, evaluating, or auditing products to assure conformance to quality standards and performance incur appraisal costs. These activities include code inspections, testing, quality audits, and software measurement activities such as metrics collection and analysis.

The cost of nonconformance includes all expenses the developer incurs when the system does not operate as specified. Internal failure costs stem from nonconformance occurring before the product ships to the customer, such as the costs of rework in programming, defect management, reinspection, and retesting. External failure costs arise from product failure after delivery to the customer. Examples include technical support, maintenance, remedial upgrades, liability damages, and litigation expenses.

Table 2 shows the various categories of software quality costs for CBSs.

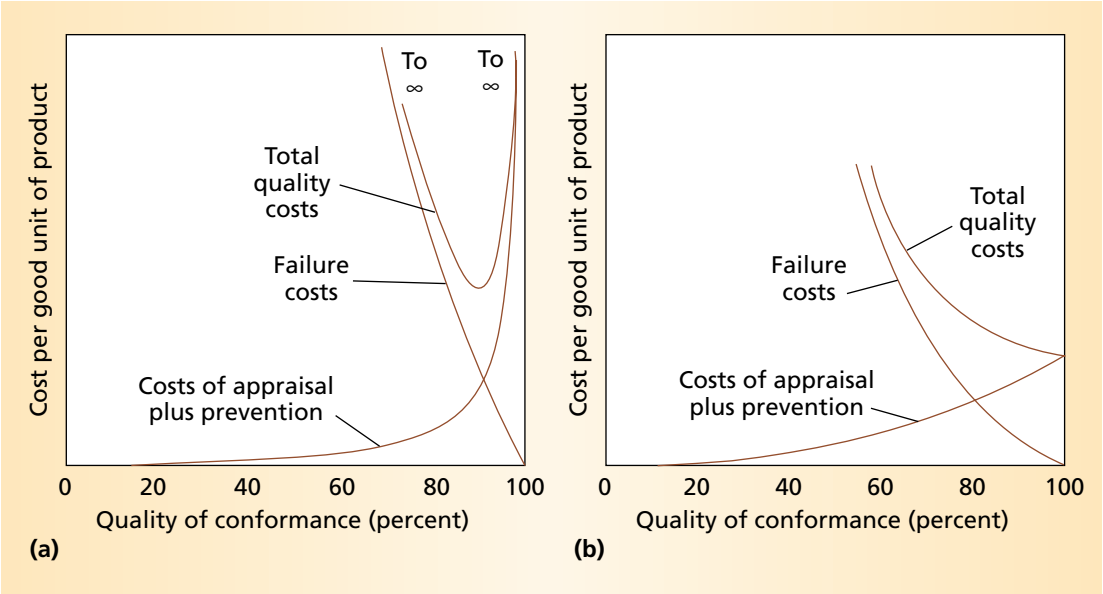


Figure 1. Models depicting the relationship between costs and quality. (a) The optimum quality cost model shows the relationship between cost per good unit of product and quality of conformance. (b) The revised model benefits from technological developments that reflect the ability to achieve very high quality at finite cost. Figures adapted from Campanella.[2]

CoQ and CoSQ models

In any development process, models that depict the relationship between costs and quality can guide decisions regarding investments in quality improvement. Discussions of such models in the economics and management literature generally depict a nonlinear relationship between CoQ and quality.[2] Accurate cost-quality models can be invaluable to managers and developers, guiding resource and cost management and other aspects of the software development process.

Figure 1a depicts the classic model of optimum quality costs. In this model, which shows the relationship between the cost per good unit of product and the quality of conformance, expressed as a percentage of total conformance, prevention and appraisal costs rise asymptotically as the product achieves complete conformance.

Recent technological developments inspired a revised model that reflects the ability to achieve very high quality, or "perfection," at finite costs. Shown in Figure 1b, this model, proposed by Frank Gryna, has two key concepts:

- moderate investments in quality improvement result in a significant decrease in the cost of nonconformance, and
- focusing on quality improvement by defect prevention results in an overall decrease in the cost of testing and related appraisal tasks.

We can analyze these models in terms of our proposed quality metrics. The quality of conformance in the original model can represent one quality metric, such as adaptability or reliability. Accordingly, the vertical axis represents a CoSQ component—namely, the portion of quality costs dedicated to improving the particular quality factor. Intuitively, the same nonlinear relationship should hold. Increasing the investment in improving a certain quality factor should increase the value of the corresponding metric, and the amount of this increase should taper off as the product achieves high quality levels. "Perfect" quality may not be achievable at finite costs, particularly in CBSs, where we cannot accurately determine the quality and performance of the COTS components.

Although we may be able to determine the overall CoQ with reasonable accuracy, determining the amount dedicated to improving a particular quality factor is difficult because all factors interrelate. For quality metrics such as customer satisfaction, the relationship between cost and quality may be too complex for such a simple model, as increased investments in quality improvement may be invisible to the customer. For example, users may find 95 percent reliability satisfactory, making further investments in reliability pointless. Further, customer satisfaction may increase in jumps, resulting in a discontinuous cost-quality curve, although empirical studies should verify this behavior.

Capability maturity models

Quality improvement's return on investment depends on the software engineering environment. Stephen Knox discusses the cost of software quality based on the Software Capability Maturity Model.[10] The SW-CMM maintains that a software development envi-

ronment has a measurable process capability analogous to industrial processes. The SW-CMM quantifies the capability and maturity of the software development process using five levels, ranging from a chaotic, ad hoc development environment to one that is mature and optimizing. These levels can also express the software engineering environment metric we propose. Based on the data presented, Knox makes two assumptions:

- The total cost of quality at SW-CMM Level 1 equals approximately 60 percent of the total cost of development.
- The total cost of quality will decrease by approximately two-thirds as the development process reaches SW-CMM Level 5, or full maturity.

Figure 2 shows the various software cost-of-quality categories, as well as the total cost of software quality, according to Knox, for the five SW-CMM levels.

A combination of Knox's model and traditional models provides a more accurate view of the CoQ in CBSs. A three-dimensional model based on CoQ, quality, and the software engineering environment can help determine financially sound investments in quality, based on the development environment. In the case of CBSs, where different vendors can have widely varying software engineering environments, such a model can help guide the vendor-selection process.

In place of Knox's SW-CMM levels, we can use the Software Acquisition Environment CMM to express the software engineering environment.[7] The levels of acquisition maturity range from Initial, at Level 1, to Optimizing, at Level 5. This model defines key process areas for Levels 2 through 5, in which a key process area states the goals the software must satisfy to achieve each level of maturity. SA-CMM and SW-CMM share a synergistic relationship, and we can use them in parallel by defining a software engineering environment metric with two weighted components, one corresponding to each CMM.

Applying the metrics

One objective of evaluating costs of quality is to determine ways to reduce them. A basic method involves investing in prevention costs, with the goal of eliminating nonconformance costs. As confidence in system quality increases, we can afford reductions in appraisal costs, leading to a reduction in total CoQ.

We can approach investments in quality improvement from the perspective of return on investment and increased conformance to requirements such as reliability,[11] then use the metrics to evaluate the actual quality improvement achieved as a result of a particular investment in software quality improvement.

Cost-benefit analysis of traditional software systems concludes that quality improvements yield the greatest returns early in the life cycle.[11] In CBSs, we cannot make quality improvements during the early stages of the acquired components' development. To compensate for this drawback, we must spread quality improvement efforts through the various stages of system design and development. In the design phase, such initiatives include

- identifying cost factors and cost-benefit analyses that address the unique risks associated with CBSs,
- determining the level of architectural match between the application and the COTS components, and
- evaluating the complexity and cost associated with integration, interoperability, and middleware development.

Our metrics can help decide between in-house development and COTS acquisition and, if the latter, how to select the most suitable component. In the development phase, metrics can help estimate the costs associated with the traditional development process. During the entire life cycle, metrics can guide the estimation of costs associated with the unique testing requirements of COTS-based systems, such as integration testing, end-to-end testing, and thread testing. After delivery, we can use cost metrics for trend analysis of the COTS market.

In the cost-benefit analysis of CBSs, we must avoid premature judgment, as the benefits of COTS component acquisition may materialize gradually. The paradigm shift from conventional to COTS-based

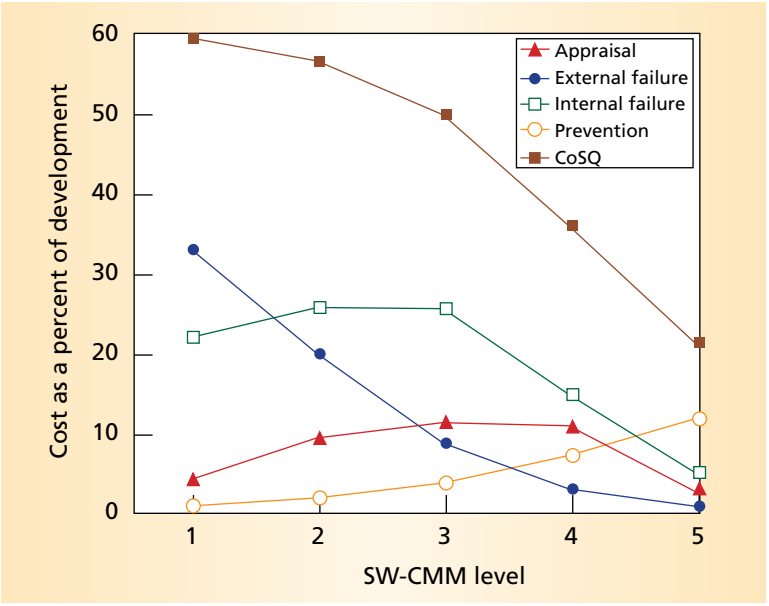


Figure 2. Knox's model for the cost of software quality, which tracks individual software cost-of-quality categories as well as the total cost of software quality. Figures adapted from Campanella.[2]

software engineering requires a considerable initial investment. Short-term analysis results may favor in-house development over COTS component acquisition, which argues for considering the software life cycle and level of reuse when making such decisions.[12] COTS products change rapidly, with long-term effects, and research on CBS development is still in the early stages. Given that cost-effectiveness and quality are the two major factors in deciding for or against component acquisition, we face an urgent need for empirical and analytical research that will lead to more accurate models of cost and quality in CBSs.

References

1. B. Boehm and C. Abts, "COTS Integration: Plug and Pray?" *Computer*, Jan. 1999, pp. 135-138.
2. J. Campanella, *Principles of Quality Costs: Principles, Implementation, and Use*, ASQ Quality Press, Milwaukee, Wis., 1999.
3. N.F. Schneidewind, "Software Metrics for Quality Control," *Proc. 4th Int'l Software Metrics Symp.*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 127-136.
4. R.A. Paul et al., "Software Metrics Knowledge and Databases for Project Management," *IEEE Trans. Knowledge and Data Eng.*, Jan./Feb. 1999, pp. 255-264.
5. P. Brereton and D. Budgen, "Component-Based Systems: A Classification of Issues," *Computer*, Nov. 2000, pp. 54-62.
6. R.A. Paul, "Metrics-Guided Reuse," *Int'l J. Artificial Intelligence Tools*, vol. 5, nos. 1 and 2, 1996, pp. 155-166.
7. J. Cooper, M. Fisher, and S.W. Sherer, *Software Acquisition Capability Maturity Model (SA-CMM) Version 1.02*, tech. report CMU/SEI-99-TR-002, Carnegie Mellon Software Eng. Inst., Pittsburgh, 1999.
8. R.A. Paul et al., "Assurance-Based Y2K Testing," *Proc. 4th Int'l Symp. High-Assurance Systems Eng.*, IEEE CS Press, Piscataway, N.J., 1999, pp. 27-34.
9. M. Tsagias and B. Kitchenham, "An Evaluation of the Business Object Approach to Software Development," *J. Systems and Software*, June 2000, pp. 149-156.
10. S.T. Knox, "Modeling the Cost of Software Quality," *Digital Technical J.*, Fall 1993, pp. 9-16.
11. S.A. Slaughter, D.E. Harter, and M.S. Krishnan, "Evaluating the Cost of Software Quality," *Comm. ACM*, Aug. 1998, pp. 67-73.
12. C. Sledge and D. Carney, "Case Study: Evaluating COTS Products for DoD Information Systems," *SEI Monographs on the Use of Commercial Software in Government Systems*, Carnegie Mellon Software Eng. Inst., Pittsburgh, 1998.

Sabra Sedigh-Ali *is a PhD candidate in the School of Electrical and Computer Engineering at Purdue University. Her research interests are software testing and quality management and component-based software development. She received an MS in electrical engineering from Purdue University. Sedigh-Ali is a student member of the IEEE and the ACM. Contact her at sedigh@ecn.purdue.edu.*

Arif Ghafoor *is a professor in the School of Electrical and Computer Engineering at Purdue University. His research interests are multimedia information systems, database security, and distributed computing. He received a PhD in electrical engineering from Columbia University. Ghafoor is a fellow of the IEEE. Contact him at ghafoor@ecn.purdue.edu.*

Raymond A. Paul *is the deputy director of investment and acquisition in the Office of the Assistant Secretary of Defense. His research interests are system and software testing and software measurements. He received a DE from the University of Tokyo. Paul is a senior member of the IEEE and the ACM. Contact him at raymond.paul@osd.mil.*

ASSIGNMENT 3

Integrating Testing and Implementation into Development

Maaret Pyhäjärvi, Conformiq Software Ltd.
Kristian Rautiainen, Helsinki University of Technology

Abstract: Cost of defects increases significantly the later the defects are found. Testing is the means to find defects, and we view testing in the broader perspective of maximizing customer satisfaction and providing feedback for process refinement, in addition to just detecting and getting defects corrected in the software. Testing is an integral activity in software development. Testing should be included early in the software development, but achieving this right-from-the-beginning integration of testing and implementation into efficient development has proven a challenge in practice. This article presents problems with use of the V-model as a reference model for this integration. We identify differences in testing in plan-driven and agile development approaches, and explain how to use a general framework for managing software product development, Cycles of Control, to structure the links between implementation and development and to use software development project dynamics for the benefit of the project.

Keywords: Software Testing, Test Planning, Agile Testing, Exploratory Testing

EMJ Focus Areas: Systems Engineering, Program & Project Management

Developing software is a challenging, intellectual task. The essence of software—complexity, conformity, changeability, and invisibility (Brooks, 1987)—makes introducing defects into the software in the construction process a fact. Software does not have a manufacturing stage in the sense of traditional products with statistical quality control, but stays in the design stage, also during software maintenance (Beizer, 1990). Testing in order to find the defects is an essential activity, but it can be difficult to integrate into software development. Effort estimates for testing vary from 30 to 90% (Beizer, 1990), with half of the development costs being the typical estimate. Test processes are considered excellent candidates for improvement (Rico, 2000), yielding results for Cycle Time Reduction, Productivity Increase, Quality Increase, and Return-on-Investment, such as 6-fold, 6-fold, 6-fold, and 9:1. Optimizing software development as a whole, consisting of implementation (including requirements) and testing, would be an opportunity for improvement.

The state-of-the-art model on how testing should be coordinated with development is the so-called V-model described in Craig and Jaskiel (2002). The V-model forms the basis of what is taught on how to structure testing in relation to development. As such, it was also the basis for testing in our research on managing software product development in small companies. In applying the V-model in practice, it became evident that the model could be easily misused to form structures that are rigid and contain overlapping tasks.

The V-model depicts a plan-driven approach to developing software, as it is essentially based on the waterfall model, presented by Royce (1970) with more degrees of freedom than is depicted nowadays (Schach, 2002). In contrast to the traditional plan-driven software development processes, several agile process models have been proposed (e.g., Beck, 2000; Highsmith, 2000; Schwaber and Beedle, 2002). Empirical studies have shown that many companies in Internet software and PC software use flexible processes (Cusumano and Selby, 1998; Cusumano and Yoffie, 1999), and such flexible processes have been found to result in increased customer satisfaction (MacCormack et al., 2001).

According to Boehm and Turner (2003) we should tailor our development lifecycles to balance between plan-driven and agile development. The agile processes do not give clear instructions on how to do testing, except for extreme programming (Beck, 2000), and basing testing on a plan-driven approach if our goal is realizing benefits of agility and flexibility is very difficult in our experience.

About the Authors

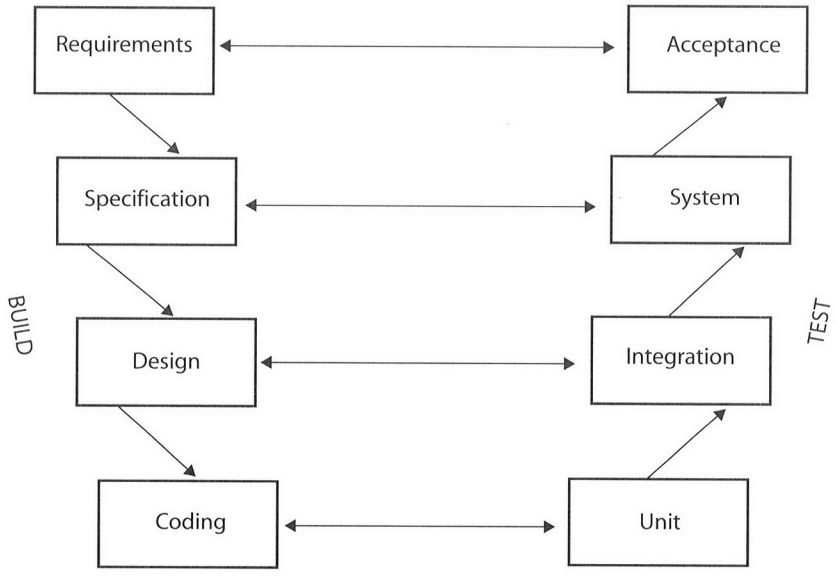
Maaret Pyhäjärvi works as senior testing consultant at Conformiq Software Ltd., a company specializing in software testing. She has worked as a tester and test manager in companies, as well as a researcher and teacher of software testing at Helsinki University of Technology (HUT). Her focus is on improving the testing process and integrating testing into development in different contexts and packaging lessons learned from various application areas.

Kristian Rautiainen received his MSc at HUT and is currently working on his DSc (Tech) degree. He has been teaching software processes at HUT since 1998 and his research interests include software processes and software engineering management. He is currently exploring how to manage software product development in SMEs.

Contact: Maaret Pyhäjärvi, Conformiq Software Oy Ltd., Lars Sonckin kaari 16, 02600 Espoo, Finland; phone +358 40 823 3777; maaret.pyhajarvi@conformiq.com

Refereed management tool manuscript. Accepted by special issue editor Hans Thamhain. Previous version presented at HICSS-36 in 2003.

Exhibit 1. The V-Model of Testing



In this article we present our findings on applying the V-model in practice and suggestions on how to augment the V-model in efforts to integrate implementation and testing into development. With integration we mean that actual test execution, also for higher levels of tests, can take place early in development instead of just planning for testing to be done in the later phases of a project. We do this by using the V-model as a basis for defining the needed levels of testing and the Cycles of Control (CoC) framework (Rautiainen et al., 2002) to help understand the software development project dynamics and links between testing and implementation.

The rest of the article is structured as follows: research goals and methodology, the V-model and challenges in its use, combining the V-model and the CoC framework, and a short discussion and managerial implications.

Research Methodology

The research presented in this article has been performed as action research in close cooperation with several Finnish software companies, guaranteeing practical relevance. The article is based on research conducted in two phases. In the first phase the focus was on small software product companies, which were involved in the research in order to improve the management of their software development. The researchers' role was mostly consulting. We conducted semi-structured interviews and reviewed available documentation to understand the situation at hand as well as the situation after improvements.

In the second phase the second author continued the work with small software product companies, while the first author switched jobs to a company specializing in software testing. The first author was involved in a test process improvement (TPI) (Koomen and Pol, 1999) benchmarking of 15 companies, as well as improvement of the whole testing process of developers, testers, customers, and users. The focus was thus widened from small software product companies to include large software product companies and tailor-made software from the perspectives of customer and developer-made testing. In the benchmarking, as well as in the analysis of the situation with developer testing, we conducted semi-structured interviews and reviewed existing documentation. To understand the coordination between testing and implementation, we used the CoC framework as a basis, as the TPI model is focused on high-level testing by a separate testing group, with its basis in the V-model's built-in assumptions.

Summarizing experiences reported by others in literature helped us in explicating the reasons behind the difficulties we had observed in the companies. This helped us formulate our approach better. There seemed to be little information on the test process and its integration to the whole development lifecycle in the software engineering journals and conferences. Results of empirical and case research within the testing community is typically published in magazines such as *Software Testing and Quality Engineering Magazine* and in non-academic conferences, into which we widened our scope, having realized the mathematical nature of testing research in the academic conferences.

The V-Model of Testing

Overview. In the field of software testing, the V-model is the state-of-the-art taught on practically every course on testing. The V-model (see Exhibit 1) splits the testing process into levels on which testing is carried out incrementally in conjunction with system implementation.

The V-model is an extension of the simple waterfall model, where each process phase concerned with implementation has an associated verification and validation activity called *test level.* Testing on each level should be controlled to avoid overlapping. The V-model shows how the testing activity can—and should—be taken into account much before there is some source code to actually be tested. Traditionally the individual test plans for the test levels are seen as the links between these activities (Pol et al., 2002; Craig and Jaskiel, 2002), coordinated with a master test plan and controlled with entry and exit criteria.

Unit testing and integration testing are typically seen as the low-level tests (Pol et al., 2002) and as such suggested to be

executed by developers themselves. Some interpretations see the integration level as a higher-level test and include new resources in testing this point of view. For someone working as a tester or test manager, as the first author has, it is typical to think of the separate group's testing as the system testing, if working within the organization developing software. If working at the customer or user organization, the same approach is applied to acceptance testing. For these high-level tests the lower levels are expected to have been performed, but these are typically planned by the testers to target as many of all possible defects as possible.

The V-model and its variations essentially bring forth three important points: testing a smaller part before putting it into a larger system is a good approach, there are several points of views to test for, and testing efforts can start with planning as soon as the higher level requirements have been identified.

Problems with the V-model. The V-model, as well as its variations, are just extensions of the waterfall model. The waterfall model includes the assumption that a complete set of requirements is defined in advance. There is some allowance for redefinition and redesign, but changes become more expensive as the process goes forward, because so much depends on what has already been done. Reluctance to fix bugs such as requirement bugs found in acceptance testing or specification bugs found in system testing, as these are seen as the V-model waterfall's last phases, places testers as change recommenders late in the project, when every change they recommend will carry a significant price (Kaner et al., 2002). Reviewing the requirements and specifications does not fully solve the problem, as people tend to know what they want only after it is tangible in their own perspective. Late in the V-model, the features have been designed and committed to and the available money has been spent on them. The remaining free variables at the time of high-level tests are time-to-completion and quality. To counter these risks, incremental development is an increasingly popular mode of development (Iansiti and MacCormack, 1997). Basing testing on the expectations set by the V-model in such a context is difficult, especially due to the increased role of regression testing invisible in the V-model.

One important context in which the V-model is considered to be appropriate is contracted software if a customer is likely to change his mind and unlikely to willingly take responsibility of the costs associated with the changes. The V-model's built-in waterfall creates a clear contract with clear rules for allocation of risk (Kaner et al., 2002). After signing off requirements, every change request is a scope change associated with a cost to the customer. This may not serve the customer best, and for in-house development, the cost structure is not favorable.

The V-model used as a basis for creating schedules results in defining a testing phase—a reserved time for the activity—for each of the levels. The levels should, however, be interpreted as continuous activity with a perspective. The form in which the V-model is presented communicates this idea poorly. These structures are, in our experience, common and ineffective, as well as difficult to change.

Having given up on the waterfall assumption for testing, the structure for coordinating testing with the V-model does not provide enough details to support the integration of testing and implementation. From such a context, the V-model as a basis for testing activities has been strongly criticized (Marick, 1999). Marick points out that any model for testing that is just an extension to a development model will not suffice in providing the structure testing needs. The essential problems with these models are that they ignore that software is developed as a series of handoffs and these handoffs are the structure that enables testers to provide early value for the project. Each handoff changes the behavior of the system and thus the need for intelligent ways of focusing regression testing is needed. Basing the testing process on assumptions about existence, accuracy, completeness, and timeliness of development documentation is not always feasible. There will always be changes and tests cannot be designed based on a single version of a document. Also, it is not necessary to execute all tests derived from a single document together, but the tests should be designed based on outputs of each activity, e.g., level of requirements documentation, and then execute all tests in the order that makes the most sense, not the order dictated by the V-model. Marick's ideas (1999) were criticized by traditional testing V-model advocates (Goldsmith, 2002; Goldsmith and Graham, 2002a; Goldsmith and Graham, 2002b) as claiming that system development can proceed without adequate requirements and that it is predicated upon poor practices just because they are common. They note that, "if you leave test design until the last moment, you won't find the serious errors in architectural and business logic until the very end" (Goldsmith and Graham, 2002b), which assumes that, for example, system testing would be a phase, instead of a continuous activity. We see this as a typical example of conflict between advocates of agile and plan-driven approaches.

The V-model is essentially document-driven. Our experiences with product development have shown that documents are important to the level they are actually used, either internally or by the customer. This is especially true in a small company context where there is always a tradeoff to be made with every hour of effort used.

Testing is supposed to find bugs, and correcting these bugs changes the program. Even if the need of correction is explicated as in Spillner et al. (2002), there would still be several levels of regression testing to be performed. Even if the contents of the test levels would otherwise be specified, this has proven difficult in practice.

The V-model advocates early test design, definition of test cases as soon as the appropriate level of documentation has been finished. It has been suggested that the V-model's early test planning approach would help programmers avoid defects by using detailed test cases testers have written based on first versions of documentation, like reported in Kaner et al. (2002); however, reviews and inspections are likely to be more efficient in order to help correct defects early than relying on pre-writing tests that will never be run (Kaner et al., 2002). The idea should be that the views that testing can provide are used early to improve the specification (Beizer, 1990), not to define the details of test cases that might be thrown away and never used if plans change.

If several projects are coordinated together for implementation of one system, applying the V-model has been deemed difficult. One system's acceptance tests should take place before system testing the whole, and when changes are made the acceptance test may need to be re-executed for every installation of the subsystem in the shared environment. System testing such a large system should be a continuous activity.

Exhibit 2. Factors Discriminating Plan-Driven and Agility

Factor	Plan-driven Discriminator	Agility Discriminator
Size	Methods evolved to handle large products and teams; hard to tailor down to small projects	Well matched to small products and teams; reliance on tacit knowledge limits scalability
Criticality	Methods evolved to handle highly critical products; hard to tailor down efficiently to low-criticality products	Untested on safety-critical products; potential difficulties with simple design and lack of documentation
Dynamism	Detailed plans and "big design up front" excellent for highly stable environment, but a source of expensive rework for highly dynamic environments	Simple design and continuous refactoring are excellent for highly dynamic environments, but present a source of potentially expensive rework for highly stable environments
Personnel	Need a critical mass of scarce experts able to tailor methods during project definition, but can work with fewer later in the project, unless the environment is highly dynamic. Can usually accommodate some people able to follow step-by-step instructions	Require continuous presence of critical mass of scarce experts able to tailor methods; risky to use non-agile people, who need step-by-step instructions
Culture	Thrive in a culture where people feel comfortable and empowered by having their roles defined by clear policies and procedures; thrive on order	Thrive in a culture where people feel comfortable and empowered by having many degrees of freedom; thrive on chaos

In small companies and projects with less than 10 developers, it has been difficult to separate one level from another in a practical fashion. Implementing all levels has resulted in confusion of the differences that are difficult to explain. Especially with continuous system testing by a separate group, integration testing is done from a technical perspective as part of unit testing the implemented interface, and incremental system testing checks the newly available features, leaving separate integration testing difficult to justify.

There are several variations of the V-model, each complementing some of the difficulties in communicating testing with the model. Examples of variations in number of levels are British Computer Society (1999), IEEE (2001), and Beck (2000); variation in explicating testing and implementation on both sides of the V-model in Spillner et al. (2002); and applying V-model with iterations in Broekman and Notenboom (2003).

Combining Plan-driven vs. Agile Approaches

Differences in Development Approach. Plan-driven development and agile development have different home grounds and critical factors (Boehm and Turner, 2003). The critical factors are summarized in Exhibit 2, which is a direct quote from Boehm and Turner (2003) on how the differences on home grounds and risks are seen in practice.

A risk based approach for selecting and balancing between a plan-driven and agile approach is suggested in Boehm and Turner (2003). If agile risks dominate, the development approach should take the plan-driven approach and vice versa.

If the approach is plan-driven, applying the V-model, which has its roots in the same home ground, is applicable. If the approach taken in development is agile, testing needs to be structured differently.

Differences in Testing. Whereas the plan-driven testing could be referred to as "traditional testing," comparison to the agile home ground and discriminators (Boehm and Turner, 2003) and agile values (Beck et al., 2001) shows that the agile testing approach would be exploratory testing, described in Bach (2003). One should note that exploratory testing is not plan-driven, but it is still a planned and systematic approach in testing. In Exhibit 4 we have summarized some of the essential differences in how testing is described in these approaches. As a representative of the plan-driven approach, we use the ISEB certificate (British Computer Society, 1999) and as a representative of the agile approach we use the ideas presented in "Lessons Learned in Software Testing" by Kaner et al. (2002). As insightfully noted by Boehm and Turner (2003), projects may need tailoring of their lifecycle to include both of the approaches, and this is what the context-driven school of testing (Kaner et al., 2002) also suggests.

When coordinating testing and implementation as parts of development, we have essentially four ways of combining plan-driven and agile implementation efforts and plan-driven and agile testing. We have experiences in implementation and testing being plan-driven and both being agile-oriented and working together productively; however, mixing the approaches is an interesting question. Our experiences have shown that introduction of agile working methods in a small company context may be severely handicapped by having a plan-driven testing approach. We have little experience in combining plan-driven implementation and agile testing, but the cultural barrier in introducing that seems to be quite high.

Our Approach. To augment the V-model in efforts to integrate implementation and testing into development and bridge the two views, we suggest a two-phased approach. First, we suggest that the positive in the V-model should be emphasized, and the negative impact the typical interpretations based on the form of the model should be minimized. As suggested above, three important lessons in the V-model apply: early involvement of testing, testing smaller parts first to minimize time for locating the defect, and use of several points of view. This

Exhibit 3. Comparison of Plan

Aspect	Plan-driven Testing	Exploratory Testing
Order of test execution	Low-level tests are executed before high-level tests	Testing should take place on all levels in an order that makes most sense
Focus in controlling the testing	V-model and size of testable items on each level	Depth of testing for the system currently at hand based on its maturity
View of test strategy	Seen as something above test project plans, combining several projects	Seen as the project's tangible testing approach combining test techniques to quality criteria, project environment and product elements
Test design	Early test design emphasized	On-time test design emphasized
Test cases	Test case can be executed and interpreted by someone other than who designed it. The test case must define the expected output	Questions to the software that may also be open-ended, even though thinking of the hypothesis is important, documentation need varies
Test techniques	Preferably mathematical form, enables same results for different users	Preferably heuristic form, enables useful results for different users
Role in test design	Test design first by tester to find faults in high-level requirements	Test first design by developer
Culture	Control, in form of emphasizing entry and exit criteria in testing	Collaboration, in form of emphasizing testing as service providing value constantly
Types of processes	Step-by-step processes, requiring as little interpretation as possible	Backward-forward processes, leaving room for deciding when each process phase should be done
Role emphasis	Emphasizes test manager and the manager's skills	Emphasizes each tester and his skills
Focus of improvement	Focus on developing methods	Focus on developing skills
View on practices	Best practices exist and should be applied	All so-called best practices are just heuristics, rules of thumb
Requirements specification	If you don't have a specification, you can't test	Requirements are useful fiction at best
Validation and testing	Testing is verification and validation is done if needed	Validation is an essential part of testing

is done with selection of appropriate testing levels. Second, we suggest applying the CoC framework to find the different timeframes in which testing and implementation are integrated most effectively.

Selecting Testing Levels. When structuring testing and implementation in development, the first thing in structuring testing to fit the needs is to assess the goals and needs of each of the test levels. Each level carries a handoff and communication overhead, and thus the number of levels should be minimized. In the extreme programming model (Beck, 2000), the unit and integration testing levels are combined to form the unit testing perspective by developer. The system and acceptance testing levels are combined to form the acceptance testing perspective by the customer. As integration takes places all the time in very small increments and the system is built based on customer stories of features, this combination is rational.

Exhibit 4 describes the options to consider with test levels. Based on our experiences, we identify factors that influence the selection of test levels. As a general rule, we suggest minimizing the number of test levels applied.

Applying CoC Framework. In efforts to understand software product development and how to control it, a framework for managing software product development was introduced, called Cycles of Control (CoC) (Rautiainen et al., 2002). With the limitations in the V-model as described, our experiences suggest that CoC can be effectively used in communicating the interfaces in testing and development to facilitate both plan-driven as well as agile approaches.

The details of testing provided in software process models do not help testers understand their role in relation to the process. A tester's role is to find and report defects and verify that the reported defects have been resolved, either by a programmer fixing them or by management deciding that they will not be fixed for some reason. The CoC framework helps in understanding testing in relation to other software development activities. It sets four timeframes on which one needs to address certain issues in development. The time frames—depicted as cycles—are presented in Exhibit 5. The leftmost cycle, named *Strategic Release Management*, organizes and deals with all ongoing major activities requiring attention from product development. *Release Project Management* deals with issues on the level of individual

Exhibit 4. Selection of Test Levels

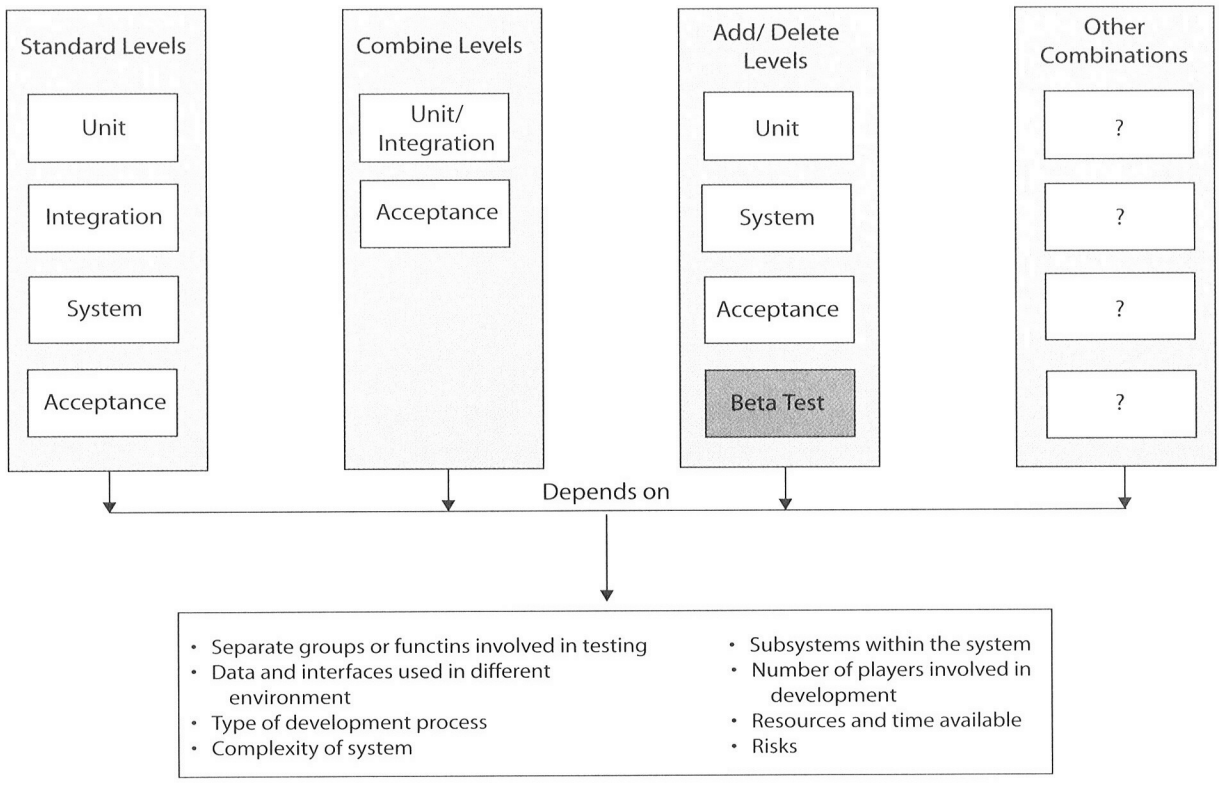
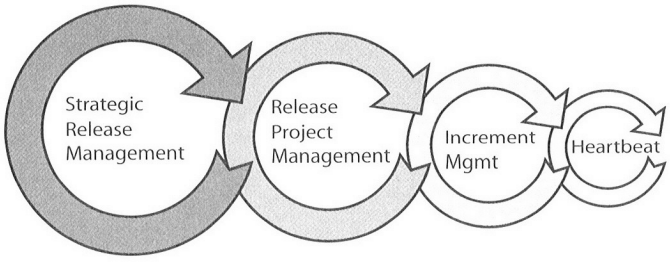


Exhibit 5. The CoC Framework



projects aiming for a product release. *Increment Management* deals with managing individual increments producing a part of a release project's deliverables. *Heartbeat* deals with coordinating status within the project or team. From the perspective of testing, we can separate two types of heartbeats: management heartbeat could be a biweekly meeting to coordinate what has been done and what will be done next, whereas implementation/testing coordination heartbeat could take the form of daily-build/daily-test. This will help the testers stay in pace with the developers and both parties can use the latest information. At the end of an increment, the team should have completed a feature or set of features that are tested from some perspectives. For example, functional testing should have been done to verify and validate the added functionality. Using short increments, e.g., one month, makes completing all testing at the end of each increment impossible. For example, customer acceptance tests can be done only when the system including the new increment is delivered to the customer. Also, time-consuming testing, such as running different performance tests cannot be done in time for the end of the same increment in which the tested functionality has been developed. Therefore part of the testing is always done in parallel rather than in pace with implementation. This should be considered when planning the project, i.e., through planning a stabilizing increment at the end of the project, where no new functionality is added, but rather the testing is allowed to "catch up" and the bugs are fixed.

Discussion and Managerial Implications

In this article, we discussed the use of a general iterative and incremental framework defined for managing product development—CoC—to help with integrating testing into software development projects. We suggested that the number of test levels should be minimized and some of the typical interpretations of the V-model avoided. To provide the detail of integration missing from the V-model, we suggest use of CoC-framework to assess the different time frames of handoffs and communication between roles and teams. Our experiences suggest that added detail in the pacing in CoC is critical for success in the integration.

We conclude this article with implications to planning development that project managers and test managers should explicitly take into account in planning software development:

- In scheduling a project, test levels should not be addressed as phases, but as continuous activities.
- The test phases to be defined in schedules should proceed by depth of test from smoke test (basic functionality) to function test (testing one functionality at a time to capability test (testing several functionalities but limiting data) to reliability test (testing several functionalities with varying data).
- The actual differences in different test levels should be addressed and the number of levels minimized; however, at

least two levels are needed to provide the technical and the end-user perspectives. Combining several projects brings forth new levels of testing needed.

- Instead of basing a test level's tests on a level of requirements, plan for designing and executing tests in the order that makes most sense for the project as early as possible
- Early involvement of test execution requires projects to focus on architectures that enable early testing; typically it should be possible to add and test features one by one.
- Instead of early test case definition to documented test cases, consider including the testing perspective in reviews.
- Enable on-time feedback for developers from testing to control the testing costs.

Our efforts to research this area continue. We aim at understanding more on dynamics of levels and mixes of levels and timescales for different types of development projects, as well as being able to provide more detail to contexts influencing different types of selection in development and testing as an integral part of it.

References

Bach, James, "Exploratory Testing Explained," in *The Testing Practitioner*, E. van Veenendaal, ed. UTN Publishers (2003).

Beck, K., *Extreme Programming Explained*, Addison-Wesley (2000).

Beck, K., M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Agile Manifesto" (2001).

Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold (1990).

Boehm, B., and V. Basili, "Software Reduction Top 10 List," *Computer, 34:1* (2001), pp. 135–137.

Boehm, B., and R. Turner, "Using Risk to Balance Agile and Plan-driven Methods," *IEEE Computer, 36:6* (2003).

British Computer Society, "Foundation Syllabus V2.0 in Software Testing," (1999).

Broekman, B., and E. Notenboom, *Testing Embedded Software* (2003).

Brooks, F., "No Silver Bullet: Essence and Accident in Software Engineering," *IEEE Computer*, 20:4 (1987), pp. 10–19.

Craig, R., and S. Jaskiel, *Systematic Software Testing* (2002).

Cusumano, M.A., and R.W. Selby, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, Simon & Schuster Inc. (1998).

Cusumano, M.A., and D.B. Yoffie, "Software Development on Internet Time," *IEEE Computer* (1999), pp. 60–69.

Goldsmith, Robin F., "This or That, V or X?" *Software Development* (2002).

Goldsmith, Robin F., and D. Graham, "Test-driven Development," *Software Development* (2002a).

Goldsmith, Robin F., and D. Graham, "The Forgotten Phase," *Software Development* (2002b).

Highsmith, III J., *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House Publishing (2000).

Iansiti, M., and A. MacCormack, "Developing Products on Internet Time," *Harvard Business Review, 75:5* (1997).

IEEE. SWEBOK—Guide to the Software Engineering Body of Knowledge. Abran, Alain, Moore, J. W., Bourque, P., Dupuis, R., and Tripp, L. L. IEEE Trial Version 1.00 (2001).

Kaner, C., J. Bach, and B. Pettichord, *Lessons Learned in Software Testing—A Context-driven Approach*, Wiley Computer Publishing (2002).

Koomen, T., and M. Pol, *Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing*, ACM Press (1999).

MacCormack, A, R. Verganti, and M. Iansiti, "Developing Products on 'Internet Time': The Anatomy of a Flexible Development Process," *Engineering Management Review, 29:2* (2001), pp. 90–104.

Marick, B., "New Models for Test Development," *Proceedings of Quality Week* (1999).

Pol, M., R. Teunissen, and E. van Veenendaal, *Software Testing: A Guide to the TMap Approach*, Addison-Wesley (2002).

Rautiainen, K., C. Lassenius, and R. Sulonen, "4CC: A Framework for Managing Software Product Development," *Engineering Management Journal, 14:2* (2002), pp. 27–32.

Rico, David F., "Using Cost-Benefit Analyses to Develop Software Process Improvement (SPI) Strategies," (2000), A DACS State-of-the-Art Report.

Royce, W. W., "Managing the Development of Large Software Systems," *Proceedings of Wescon* (1970), pp. 1–9.

Schach, Stephen R., *Object-Oriented and Classical Software Engineering*, WCB/McGraw-Hill (2002).

Schwaber, K, and M. Beedle, *Agile Software Development with Scrum*, Prentice Hall (2002).

Spillner, A., H. Bremen, and K. Vosseberg, "The W-model Strengthening the Bond Between Development and Test," *Proceedings of STAREAST 2002* (2002).

ASSIGNMENT 4

Regular Paper

Requirements Development, Verification, and Validation Exhibited in Famous Failures

A. Terry Bahill[1,*] and Steven J. Henderson[1,2]

[1]*Systems and Industrial Engineering, University of Arizona, Tucson,* AZ 85721-0020

[2]U.S. *Military Academy, West Point,* NY 10996

Received 11 February 2004; Accepted 31 August 2004, after one or more revisions
Published online in Wiley InterScience (www.interscience.wiley.com).
DOI 10.1002/sys.20017

ABSTRACT

Requirements Development, Requirements Verification, Requirements Validation, System Verification, and System Validation are important systems engineering tasks. This paper describes these tasks and then discusses famous systems where these tasks were done correctly and incorrectly. This paper shows examples of the differences between developing requirements, verifying requirements, validating requirements, verifying a system, and validating a system. Understanding these differences may help increase the probability of success of future system designs. © 2004 Wiley Periodicals, Inc. Syst Eng 8: 1–14, 2005

Key words: design; inspections; case studies

1. INTRODUCTION

Requirements Development, Requirements Verification, Requirements Validation, System Verification and System Validation are important tasks. This paper starts with a definition and explanation of these terms. Then it gives two dozen examples of famous system failures and suggests the mistakes that might have been made. These failures are not discussed in detail: The purpose is not to pinpoint the exact cause of failure, because these systems were all complex and there was no one unique cause of failure. The systems are discussed at a high level. The explanations do not present incontrovertible fact; rather they represent the consensus of many engineers and they are debatable. These explanations are based on many papers, reports, and movies about these failures and discussion of these failures in

*Author to whom all correspondence should be addressed (e-mail: terry@sie.arizona.edu.

Contract grant sponsor: AFOSR/MURI F4962003-1-0377

many classes and seminars since 1997. It is hoped that the reader will be familiar with enough of these systems to be able to apply the concepts of requirements development, verification and validation to some of these systems without an extensive learning curve about the details of the particular systems. When used in classes and seminars, this paper has been given to the students with a blank Table II. The students were asked to read this paper and then Table II was discussed row by row.

2. REQUIREMENTS DEVELOPMENT

A functional requirement should define what, how well, and under what conditions one or more inputs must be converted into one or more outputs at the boundary being considered in order to satisfy the stakeholder needs. Besides functional requirements, there are dozens of other types of requirements [Bahill and Dean, 1999]. Requirements Development includes (1) eliciting, analyzing, validating, and communicating stakeholder needs, (2) transforming customer requirements into derived requirements, (3) allocating requirements to hardware, software, bioware, test, and interface elements, (4) verifying requirements, and (5) validating the set of requirements. There is no implication that these five tasks should be done serially, because, like all systems engineering processes, these tasks should be done with many parallel and iterative loops.

There is a continuum of requirement levels as more and more detail is added. But many systems engineers have been dividing this continuum into two categories: high-level and low-level. High-level requirements are described with words like customer requirements, top-level requirements, system requirements, operational requirements, concept of operations, mission statement, stakeholder needs, stakeholder expectations, constraints, external requirements, and what's. Low-level requirements are described with words like derived requirements, design requirements, technical requirements, product requirements, allocated requirements, internal requirements, and how's. Some of these terms have different nuances, but they are similar. In this paper, we will generally use the terms high-level and low-level requirements, and we will primarily discuss high-level requirements.

3. VERIFICATION AND VALIDATION

Because the terms verification and validation are often confused, let us examine the following definitions:

Verifying requirements: Proving that each requirement has been satisfied. Verification can be done by logical argument, inspection, modeling, simulation, analysis, expert review, test or demonstration.

Validating requirements: Ensuring that (1) the *set* of requirements is correct, complete, and consistent, (2) a model can be created that satisfies the requirements, and (3) a real-world solution can be built and tested to prove that it satisfies the requirements. If Systems Engineering discovers that the customer has requested a perpetual-motion machine, the project should be stopped.

Verifying a system: Building the *system right*: ensuring that the system complies with the system requirements and conforms to its design.

Validating a system: Building the *right system*: making sure that the system does what it is supposed to do in its intended environment. Validation determines the correctness and completeness of the end product, and ensures that the system will satisfy the actual needs of the stakeholders.

There is overlap between system verification and requirements verification. System verification ensures that the system conforms to its design and also complies with the *system requirements.* Requirements verification ensures that the *system requirements* are satisfied and also that the technical, derived, and product requirements are verified. So checking the *system requirements* is common to both of these processes.

There is also overlap between requirements validation and system validation. Validating the top-level system requirements is similar to validating the system, but validating low-level requirements is quite different from validating the system.

Many systems engineers and software engineers use the words verification and validation in the opposite fashion. So, it is necessary to agree on the definitions of verification and validation.

The Verification (VER) and Validation (VAL) process areas in the Capability Maturity Model Integration (CMMI) speak of, respectively, verifying requirements and validating the system. Validation of requirements is covered in Requirements Development (RD) Specific Goal 3 [http://www.sei.cmu.edu/cmmi/; Chrissis, Konrad and Shrum, 2003]. The CMMI does not explicitly discuss system verification.

3.1. Requirements Verification

Each requirement must be verified by logical argument, inspection, modeling, simulation, analysis, expert review, test, or demonstration. Here are some brief dictionary definitions for these terms:

Logical argument: a series of logical deductions

Inspection: to examine carefully and critically, especially for flaws

Modeling: a simplified representation of some aspect of a system

Simulation: execution of a model, usually with a computer program

Analysis: a series of logical deductions using mathematics and models

Expert review: an examination of the requirements by a panel of experts

Test: applying inputs and measuring outputs under controlled conditions (e.g., a laboratory environment)

Demonstration: to show by experiment or practical application (e. g. a flight or road test). Some sources say demonstration is less quantitative than test.

Modeling can be an independent verification technique, but often modeling results are used to support other techniques.

Requirements verification example 1: The probability of receiving an incorrect bit on the telecommunications channel shall be less than 0.001. This requirement can be verified by laboratory tests or demonstration on a real system.

Requirements verification example 2: The probability of loss of life on a manned mission to Mars shall be less than 0.001. This certainly is a reasonable requirement, but it cannot be verified through test. It might be possible to verify this requirement with analysis and simulation.

Requirements verification example 3: The probability of the system being canceled by politicians shall be less than 0.001. Although this may be a good requirement, it cannot be verified with normal engineering test or analysis. It might be possible to verify this requirement with logical arguments.

3.2. Requirements Validation

Validating requirements means ensuring that (1) the *set* of requirements is correct, complete, and consistent, (2) a model that satisfies the requirements can be created, and (3) a real-world solution can be built and tested to prove that it satisfies the requirements. If the requirements specify a system that reduces entropy without expenditure of energy, then the requirements are not valid and the project should be stopped.

Here is an example of an invalid requirements set for an electric water heater controller.

If 70° < Temperature < 100°, then output 3000 Watts.

If 100° < Temperature < 130°, then output 2000 Watts.

If 120° < Temperature < 150°, then output 1000 Watts.

If 150° < Temperature, then output 0 Watts.

This set of requirements is incomplete, what should happen if Temperature < 70°? This set of requirements is inconsistent, what should happen if Temperature = 125°? These requirements are incorrect because units are not given. Are those temperatures in degrees Fahrenheit or Centigrade?

Of course, you could never prove that a requirements set was complete, and perhaps it would be too costly to do so. But we are suggesting that many times, due to the structure of the requirements set, you can look for incompleteness [Davis and Buchanan, 1984].

Detectable requirements-validation defects include (1) incomplete or inconsistent sets of requirements or use cases, (2) requirements that do not trace to top-level requirements [the vision statement or the Concept of Operation (CONOPS)], and (3) test cases that do not trace to scenarios (use cases).

At inspections, the role of Tester should be given an additional responsibility, requirements validation. Tester should read the Vision and CONOPS and specifically look for requirements-validation defects such as these.

3.3. System Verification and Validation

One function of Stonehenge on Salisbury Plain in England might have been to serve as a calendar to indicate the best days to plant crops. This might have been the first calendar, and it suggests the invention of the concept of time. Inspired by a visit to Stonehenge, Bahill built an Autumnal Equinox sunset-sight on the roof of his house in Tucson.

Bahill now wants verification and validation documents for this solar calendar, although he should have worried about this before the hardware was built. This system fails validation. He built the wrong system. The people of England must plant their crops in the early spring. They need a Vernal Equinox detector, not an Autumnal Equinox detector. The ancient ones in Tucson needed a Summer Solstice detector, because all of their rain comes in July and August. System validation requires consideration of the environment that the system will operate in.

In 3000 B.C., the engineers of Stonehenge could have verified the system by marking the sunset every day. The solstices are the farthest north and south (approximately). The equinoxes are about midway between the solstices and are directly east and west. In the 21st century, residents of Tucson could verify the sys-

tem by consulting a calendar or a farmer's almanac and observing the sunset through this sight on the Autumnal Equinox next year. If the sunset is in the sight on the day of the Autumnal Equinox, then the system was built right. When archeologists find Bahill's house 2000 years from now, he wants them to ask, "What do these things do?" and "What *kind* of people built them?"

System-validation artifacts that can be collected at *discrete gates* include white papers, trade studies, phase reviews, life cycle reviews, and red team reviews. These artifacts can be collected in the proposal phase, at the systems requirements review (SRR), at the preliminary design review (PDR), at the critical design review (CDR), and in field tests.

System-validation artifacts that can be collected *continuously* throughout the life cycle include results of modeling and simulation and the number of operational scenarios (use cases) modeled.

Detectable system-validation defects include (1) excessive sensitivity of the model to a particular parameter or requirement, (2) mismatches between the model/simulation and the real system, and (3) bad designs.

At inspections, the role of Tester should be given an additional responsibility, system validation. Tester should read the Vision and CONOPS and specifically look for system-validation artifacts and defects such as these.

A very important aspect of system validation is that it occurs throughout the entire system life cycle. You should not wait for the first prototype before starting validation activities.

3.4. External Verification and Validation

System verification and validation activities should start in the proposal phase. Verification and validation are continuous processes that are done throughout the development life cycle of the system. Therefore, most of theses activities will be internal to the company. However, it is also important to have external verification and validation. This could be done by an independent division or company. External verification and validation should involve system usage by the customer and end user in the system's intended operating environment. This type of external verification and validation would not be done throughout the development cycle. It would not occur until at least a prototype was available for testing. This is one of the reasons the software community emphasizes the importance of developing prototypes early in the development process.

4. FAMOUS FAILURES

We learn from our mistakes. In this section, we look at some famous failures and try to surmise the reason for the failure so that we can ameliorate future mistakes. A fault is a defect, error, or mistake. One or many faults may lead to a failure of a system to perform a required function [www.OneLook.com]. Most well-engineered systems are robust enough to survive one or even two faults. It took three or more faults to cause each failure presented in this paper. System failures are prevented by competent and robust design, oversight, test, redundancy, and independent analysis. In this paper, we are not trying to find the root cause of each failure. Rather we are trying to illustrate mistakes in developing requirements, verifying requirements, validating requirements, verifying a system, and validating a system. Table I shows the failures we will discuss.

HMS Titanic had poor quality control in the manufacture of the wrought iron rivets. In the cold water of April 14, 1912, when the Titanic hit the iceberg, many rivets failed and whole sheets of the hull became unattached. Therefore, verification was bad, because they did not build the ship right. An insufficient number of lifeboats was a requirements development failure. However, the Titanic satisfied the needs of the ship owners and passengers (until it sank), so validation was OK [*Titanic,* 1997]. These conclusions are in Table II.

The **Tacoma Narrows Bridge** was a scaleup of an old design. But the strait where they built it had strong winds: The bridge became unstable in these crosswinds and it collapsed. The film of its collapse is available on the Web: It is well worth watching [Tacoma-1 and Tacoma-2]. The design engineers reused the requirements for an existing bridge, so these requirements were up to the standards of the day. The bridge was built well, so verification was OK. But it was the wrong bridge for that environment, a validation error. [Billah and Scanlan, 1991].

The **Edsel** automobile was a fancy Ford with a distinct vertical grille. The designers were proud of it. The requirements were good and they were verified. But the car didn't sell, because people didn't want it. Previous marketing research for the Thunderbird was successful, but for the Edsel, management ignored marketing. Management produced what management wanted, not what the customers wanted, and they produced the wrong car [Edsel].

In **Vietnam**, our top-level requirement was to contain Communism. This requirement was complete, correct, and feasible. However, we had no exit criteria, and individual bombing runs were being planned at a distance in Washington DC. Our military fought well and bravely, so we fought the war right. But it was the wrong

Table I. Some Famous Failures		
System Name	Year	Putative cause of failure
HMS Titanic	1912	Poor quality control
Tacoma Narrows Bridge	1940	Scaling up an old design
Edsel automobile	1958	Failure to discover customer needs
War in Vietnam	1967-72	No problem statement, Micromanagement
Apollo-13	1970	Poor configuration management
Concorde SST	1976-2003	It was not profitable
IBM PCjr	1983	Failure to discover customer needs
GE refrigerator	1986	Inadequate testing of new technology
Space Shuttle Challenger	1986	Bureaucratic mismanagement, failure to respond to engineers' technical concerns
Chernobyl Nuclear Power Plant	1986	Bad design, Bad risk management
New Coke	1988	Arrogance
A-12 airplane	1980s	Mismanagement, Failure to develop realistic requirements
Hubble Space Telescope	1990	Lack of total system test
SuperConducting SuperCollider	1995	Cost overruns, Failure to maintain public support
Ariane 5 missile	1996	Incorrect reuse of software, Faulty scaling up
UNPROFOR Bosnia Mission	1992-95	No cease fire agreement, Poor coordination
Lewis Spacecraft	1997	Design mistakes
Motorola Iridium System	1999	Misjudged competition, Mispredicted technology
Mars Climate Orbiter	1999	Use of different units
Mars Polar Lander	2000	Failure of middle management
Sept 11 attack on WTT	2001	Failure to anticipate terrorist threat
Space Shuttle Columbia	2002	NASA corporate culture, failure to heed lessons learned
Northeast power outage	2003	Lack of tree trimming

war. We (in Bahill's opinion) should not have been there: bad validation.

John F. Kennedy, in a commencement address at Duke University in 1961, stated the top-level requirements for the Apollo Program: (1) Put a man on the moon (2) and return him safely (3) by the end of the decade. These and their derived requirements were right. The Apollo Program served the needs of Americans: so, validation was OK. But on **Apollo 13**, for the thermostatic switches for the heaters of the oxygen tanks, they changed the operating voltage from 28 to 65 V, but they did not change the voltage specification or test the switches. This was a configuration management failure that should have been detected by verification. On the other hand, perhaps Apollo 13 was a tremendous success and not a failure. The lunar module, the astronauts, the backup systems and the backup crew were robust, so the mission was heroically saved. [*Apollo 13,* 1995].

The **Concorde** Supersonic Transport (SST) was designed and built in the 1960s and 1970s by Britain and France. It flew commercially from 1976 to 2003. The requirements for the airplane were fine and the airplane was built well. But we suggest that it fails validation: because the purpose of a commercial airplane is to make money, and the Concorde did not. [http://www.concordesst.com/]. The Concorde was a success only as a political statement, not as a business system. Once again, these conclusions are not black and white. Indeed one of the reviewers of this paper stated, The Concorde "established a basis of European technical self confidence that permitted Airbus to erode much of the US dominance in this field. Thus, it can reasonably be argued that the Concorde was a successful strategic program."

The **IBM PCjr** was a precursor to modern laptop computers, but it was a financial failure. The keyboard was too small for normal sized fingers. People did not like them and they did not buy them. Modern laptops have normal sized keyboards and PDAs have a stylus. It seems that there is an unwritten requirement that things designed for fingers should be big enough to accommodate fingers. They got the requirements wrong. They build a nice machine with good verification. And the success of present day laptops validates the concept [Chapman, Bahill, and Wymore, 1992: 13].

In 1986, **General Electric** Co. (GE) engineers said they could reduce the part count for their new **refrigerator** by one-third by replacing the reciprocating compressor with a rotary compressor. Furthermore, they said they could make it easier to machine, and thereby cut manufacturing costs, if they used powdered-metal instead of steel and cast iron for two parts. However, powdered-metal parts had failed in their air condition-

Table II Did they do these tasks right?				
Name	Year	RD	VER	VAL
Titanic	1912	No	No	Yes
Tacoma Narrows Bridge	1940	Yes	Yes	No
Edsel automobile	1958	Yes	Yes	No
War in Vietnam	1967-72	Yes	Yes	No
Apollo-13	1970	Yes	No	Yes
Concorde SST	1976-2003	Yes	Yes	No
IBM PCjr	1983	No	Yes	Yes
GE rotary compressor refrigerator	1986	Yes	No	Yes
Space Shuttle Challenger	1986	Yes	No	No
Chernobyl Nuclear Power Plant	1986	Yes	Yes	No
New Coke	1988	Yes	Yes	No
A-12 airplane	1980s	No	No	No
Hubble Space Telescope	1990	Yes	No	Yes
SuperConducting SuperCollider	1995	Yes	Yes	No
Ariane 5 missile	1996	Yes	No	No
UNPROFOR Bosnia Mission	1992-95	No	No	No
Lewis Spacecraft	1997	Yes	Yes	No
Motorola Iridium System	1999	Yes	Yes	No
Mars Climate Orbiter	1999	No	No	No
Mars Polar Lander	2000	Yes	No	Yes
September 11 attack on WTT	2001	No	Yes	Yes
Space Shuttle Columbia	2002	Yes	No	No
Northeast power outage	2003	No	Yes	Yes

ers a decade earlier [Chapman, Bahill, and Wymore, 1992: 19]

Six hundred compressors were "life tested" by running them continuously for 2 months under temperatures and pressures that were supposed to simulate 5 years' actual use. Not a single compressor failed, which was the good news that was passed up the management ladder. However, the technicians testing the compressors noticed that many of the motor windings were discolored from heat, bearing surfaces appeared worn, and the sealed lubricating oil seemed to be breaking down. This bad news was not passed up the management ladder!

By the end of 1986, GE had produced over 1 million of the new compressors. Everyone was ecstatic with the new refrigerators. However, in July of 1987 the first refrigerator failed; quickly thereafter came an avalanche of failures. The engineers could not fix the problem. In December of 1987, GE started buying foreign compressors for the refrigerators. Finally, in the summer of 1988 the engineers made their report. The two powdered-metal parts were wearing excessively, increasing friction, burning up the oil, and causing the compressors to fail. GE management decided to redesign the compressor without the powdered-metal parts. In 1989, they voluntarily replaced over 1 million defective compressors.

The designers who specified the powdered-metal parts made a mistake, but everyone makes mistakes. Systems Engineering is supposed to expose such problems early in the design cycle or at least in the testing phase. This was a verification failure.

The requirements for the **Space Shuttle Challenger** seem to be OK. But the design, manufacturing, testing, and operation were faulty. Therefore, verification was bad [Feynman, 1985; Tufte, 1997]. Bahill thinks that validation was also bad, because putting schoolteachers and Indian art objects in space does not profit the U.S. taxpayer. Low temperatures caused the o-rings to leak, which led to the explosion of Challenger. The air temperature was below the design expectations, and no shuttle had ever been launched in such cold weather. The political decision was to launch in an environment for which the system was not designed, a validation mistake.

The **Chernobyl Nuclear Power Plant** was built according to its design, but it was a bad design: Validation was wrong. Requirements and verification were marginally OK, although they had problems such as poor configuration management evidenced by undocumented crossouts in the operating manual. Human error contributed to the explosion. Coverup and denial for the first 36 hours contributed to the disaster. This is our greatest failure: It killed hundreds of thousands, perhaps millions, of people. Here are references for the U.S. Nuclear Regulatory Commission summary [Chernobyl-1], a general web site with lots of other links [Chernobyl-2], a BBC report [Chernobyl-3], and for

some photos of what Chernobyl looks like today [Chernobyl-4].

The Marketing Department at the **Coca Cola** Company did blind tasting between Coca Cola and Pepsi Cola. Most of the people preferred Pepsi. So, they changed Coke to make it taste like Pepsi. This made Coke more pleasing to the palate, but Coke's branding was a much stronger bond. People who liked Coke would not buy or drink New Coke, and they complained. After 4 months, the company brought back the original under the name of Classic Coke and finally they abandoned New Coke altogether. They did the marketing survey and found the taste that most people preferred, so the requirements development was OK. They manufactured a fine product, so verification was OK. But they did not make what their customers wanted, a validation mistake [http://www.snopes.com/cokelore/newcoke.asp].

The **A-12 Avenger** was to be a carrier-based stealth airplane that would replace the A-6 in the 1990s. This program is a classic example of program mismanagement. Major requirements were inconsistent and were therefore continually changing. Nothing was ever built, so we cannot comment on verification. Validation was bad, because they never figured out what they needed [Fenster, 1999; Stevenson, 2001].

The **Hubble Space Telescope** was built at a cost of around 1 billion dollars. The requirements were right. Looking at the marvelous images we have been getting, in retrospect we can say that this was the right system. But during its development the guidance, navigation, and control (GNC) system, which was on the cutting edge of technology, was running out of money. So they transferred money from Systems Engineering to GNC. As a result, they never did a total system test. When the Space Shuttle Challenger blew up, the launch of the Hubble was delayed for a few years, at a cost of around 1 billion dollars. In that time, no one ever looked through that telescope. It was never tested. They said that the individual components all worked, so surely the total system will work. After they launched it, they found that the telescope was myopic. Astronauts from a space shuttle had to install spectacles on it, at a cost of around 1 billion dollars. [Chapman, Bahill, and Wymore, 1992: 16]

The **SuperConducting SuperCollider** started out as an American Big Science project. Scientists were sure that this system was needed and would serve their needs. But it was a high political risk project. Poor management led to cost overruns and transformed it into a Texas Big Physics project; consequently, it lost political support. The physicists developed the requirements right and the engineers were building a very fine system. But the system was not what the American public, the bill payer, needed. [Moody et al., 1997: 99–100].

The French Ariane 4 missile was successful in launching satellites. However, the French thought that they could make more money if they made this missile larger. So they built the **Ariane 5**. It blew up on its first launch, destroying a billion dollars worth of satellites. The mistakes on the Ariane 5 missile were (1) reuse of software on a scaled-up design, (2) inadequate testing of this reused software, (3) allowing the accelerometer to run for 40 seconds after launch, (4) not flagging as an error the overflow of the 32-bit horizontal-velocity storage register, and (5) allowing a CPU to shutdown if the other CPU was already shutdown. The requirements for the Ariane 5 were similar to those of the Ariane 4: So it was easy to get the requirements right. They needed a missile with a larger payload, and that is what they got: So, that part of validation was OK. However, one danger of scaling up an old design for a bigger system it that you might get a bad design, which is a validation mistake. Their failure to adequately test the scaled-up software was a verification mistake [Kunzig, 1997].

The **United Nations Protection Force** (UNPROFOR) was the UN mission in Bosnia prior to NATO intervention. They had valid requirements (stopping fighting in former Yugoslavia is valid) but these requirements were incomplete because a peacekeeping force requires a cease-fire before keeping the peace. This expanded cease-fire requirement later paved the way for the success of NATO in the same mission. Moreover, the UNPROFOR failed to meet its incomplete requirements because they were a weak force with limited capabilities and poor coordination between countries. UNPROFOR had incomplete requirements, and was the wrong system at the wrong time. This was a partial requirements failure, and a failure of verification and validation [Andreatta, 1997].

The **Lewis Spacecraft** was an Earth-orbiting satellite that was supposed to measure changes in the Earth's land surfaces. But due to a faulty design, it only lasted 3 days in orbit. "The loss of the Lewis Spacecraft was the direct result of an implementation of a technically flawed Safe Mode in the Attitude Control System. This error was made fatal to the spacecraft by the reliance on that unproven Safe Mode by the on orbit operations team and by the failure to adequately monitor spacecraft health and safety during the critical initial mission phase" [*Lewis Spacecraft*, 1998].

Judging by the number of people walking and driving with cellular phones pressed to their ears, at the turn of the 21st century there was an overwhelming need for portable phones and the **Motorola Iridium System** captured the requirements and satisfied this need. The

Motorola phones had some problems such as being heavy and having a time delay, but overall they built a good system, so verification is OK. But their system was analog and digital technology subsequently proved to be far superior. They should have built a digital system. They built the wrong system [http://www.spaceref.com/news/viewnews.html?id=208].

On the **Mars Climate Orbiter**, the prime contractor, Lockheed Martin, used English units for the satellite thrusters while the operator, JPL, used SI units for the model of the thrusters. Therefore, there was a mismatch between the space-based satellite and the ground-based model. Because the solar arrays were asymmetric, the thrusters had to fire often, thereby accumulating error between the satellite and the model. This caused the calculated orbit altitude at Mars to be wrong. Therefore, instead of orbiting, it entered the atmosphere and burned up. But we do not know for sure, because (to save money) tracking data were not collected and fed back to earth. Giving the units of measurement is a fundamental part of requirements development. And they did not state the measurement units correctly: so this was a requirements-development mistake. There was a mismatch between the space-based satellite and the ground-based model: This is a validation error. They did not do the tests that would have revealed this mistake, which is a verification error.

On the **Mars Polar Lander**, "spurious signals were generated when the lander legs were deployed during descent. The spurious signals gave a false indication that the lander had landed, resulting in a premature shutdown of the lander engines and the destruction of the lander when it crashed into the Mars surface. … It is not uncommon for sensors … to produce spurious signals. … During the test of the lander system, the sensors were incorrectly wired due to a design error. As a result, the spurious signals were not identified by the system test, and the system test was not repeated with properly wired touchdown sensors. While the most probable direct cause of the failure is premature engine shutdown, it is important to note that the underlying cause is inadequate software design and systems test" [*Mars Program*, 2000].

The Mars Climate Orbiter and the Mars Polar Lander had half the budget for project management and systems engineering of the previously successful Pathfinder. These projects (including Pathfinder) were some of the first in NASA's revised "Faster, Better, Cheaper" philosophy of the early 1990s. It is important to note that despite early failures, this philosophy has yielded successes and is an accepted practice at NASA [2000].

Some failures are beyond the control of the designers and builders, like the failure of the **World Trade Towers** in New York after the September 11, 2001 terrorist attack. However, some people with 20/20 hindsight say that they (1) missed a fundamental requirement that each building should be able to withstand the crash of a fully loaded airliner and (2) that the FBI did or should have known the dates and details of these terrorist plans. This reinforces the point that many of our opinions are debatable. It also points out the importance of perspective and the problem statement. For example, what system do we want to discuss—the towers or the attack on the towers? For purposes of this paper, we will reflect on the buildings as a system, and assume "they" refers to the World Trade Center designers. Clearly, the buildings conformed to their original design and fulfilled their intended purpose—maximizing premium office space in lower Manhattan—so validation and verification were OK. However, the building's requirements proved incomplete, and failed to include antiterrorism provisions. This is a requirements failure.

NASA learned from the failure of the Space Shuttle Challenger; but they seemed to have forgotten the lessons they learned, and this allowed the failure of the **Space Shuttle Columbia**. At a high level, the Columbia Study Committee said that NASA had a culture that prohibited engineers from critiquing administrative decisions. The NASA culture produced arrogance toward outside advice. After the Challenger failure, they installed an 800 telephone number so workers could anonymously report safety concerns, but over the years that phone number disappeared. At a low level, the original design requirements for the Shuttle Orbiter "precluded foam-shedding by the External Tank." When earlier missions failed to meet this requirement but still survived reentry, NASA treated this as an "in-family [previously observed]" risk and ignored the requirement. But the system still failed to meet its requirements—a verification failure [*Columbia*, 2003; Deal, 2004]. We deem Columbia a system validation failure for the same reasons as the Challenger.

The **Northeast electric power blackout** in August 2003 left millions of people without electricity, in some cases for several days. Some of the causes of this failure were an Ohio electric utility's (1) not trimming trees near a high-voltage transmission line, (2) using software that did not do what it should have done, and (3) disregarding four voluntary standards. Verification and validation were fine, because the system was what the people needed and it worked fine for the 26 years since the last blackout. The big problem was a lack of industry-wide mandatory standards—a requirements failure [http://www.2003blackout.info/].

Table II presents a consensus about these failures. It uses the three CMMI process areas, RD, VER, and VAL. Table II uses the following code:

RD: Requirements Development, Did they get the requirements right?
VER: Requirements Verification, Did they build the system right?
VAL: System Validation, Did they build the right system?
"They" refers to the designers and the builders.

Engineers with detailed knowledge about any one of these failures often disagreed with the consensus of Table II, because they thought about the systems at a much more detailed level than was presented in this paper. Another source of disagreement was caused by the fuzziness in separating high-level requirements from system validation. For example, many people said, "The designers of the Tacoma Narrows Bridge should have been the world's first engineers to write a requirement for testing a bridge in strong crosswinds."

The failures described in Tables I and II are of different types and are from a variety of industries and disciplines. The following list divides them (and a few other famous failures) according to the characteristic that best describes the most pertinent aspect of the systems:

Hardware intensive: GE rotary compressor refrigerator, IBM PCjr, Titanic, Tacoma Narrows Bridge, and Grand Teton Dam
Software intensive: Ariane 5 missile, some telephone outages, computer virus vulnerability, and MS Outlook
Project: A-12, SuperConducting SuperCollider, Space Shuttle Challenger, Space Shuttle Columbia, War in Vietnam, Edsel automobile, Apollo-13, New Coke, UNPROFOR, Lewis Spacecraft, Mars Climate Orbiter, Mars Polar Lander, Three Mile Island, Hubble Space Telescope, Chernobyl Nuclear Power Plant, and Concorde SST
System: Motorola's Iridium system, Western Power Grid 1996, WTT Attacks, and Northeast Power Grid in 1977 and 2003.
Financial: Enron (2003), WorldCom (2003), and Tyco (2003).
More famous failures are discussed in Petroski [1992], Talbott [1993], Dorner [1996], Bar-Yam [2004], and Hessami [2004].

5. SYSTEM AND REQUIREMENTS CLASSIFICATION MODEL

As previously mentioned, because System Verification and Validation are often confused with Requirements Development, it is worth juxtaposing these concepts in a unified model. This model helps further clarify terminology and demonstrates how the concepts of System Verification, System Validation, and Requirements Development interact in the systems engineering design process.

Our model first divides the set of all systems into two subsets: (1) verified and validated systems and (2) unverified or unvalidated systems. The model next divides the set of all system requirements into four subsets (1) valid requirements, (2) incomplete, incorrect or inconsistent requirements, (3) no requirements, and (4) infeasible requirements.

We now explore the union of all possible systems with all possible requirement sets. The result, shown in Table III, is a ***System and Requirements Classification Model*** *(SRCM)*. This model (1) provides a means of categorizing systems in terms of design conformity and requirements satisfaction and (2) provides a way to study requirements not yet satisfied by any system. Each region of the model is discussed below, in an order that best facilitates their understanding.

Region A1: This region denotes the set of systems that have been verified and validated and have valid requirements. It represents systems that conform to their designs and fulfill a valid set of requirements. A properly forged 10 mm wrench designed and used to tighten a 10 mm hex nut is an example of a system in this region. Most commercial systems fall into this category.

Region B1: Region B1 is composed of unverified or unvalidated systems that have valid requirements. These systems have perfectly legitimate designs that, if properly implemented, satisfy a valid set of requirements. Most systems will pass through this region during development. However, some fielded system will be in this region because of poor design realization. Fielded systems in this region are often categorized as error prone or "having bugs." A 10 mm wrench that breaks when tightening a 10 mm hex nut because of poor metal content is an example of a system in this region. These systems are potentially dangerous, be-

Table III. The SRCM Model

Verified and Validated Systems	Unverified or Unvalidated Systems	No System	
A1	B1	C1	Valid Requirements
A2	B2	C2	Incomplete, Incorrect or Inconsistent Requirements
A3	B3	C3	No Requirements
		C4	Infeasible Requirements

cause presumptions about legitimate designs can hide flaws in implementation.

Region A2: This region denotes verified and validated systems that satisfy an incomplete, incorrect, or inconsistent set of requirements. These systems fail because the assigned requirements do not adequately satisfy the stakeholder needs. Adequate satisfaction results when a system meets the needs of a majority of the principal stakeholders (as defined by the chief decision maker). A system in this region would be a properly forged 10 mm wrench that is used to tighten a 10 mm hex bolt, but the bolt fails to tighten because of a previously unnoticed backing/lock nut. In this case, a new system design featuring two tools is required. Another example would be a user's manual that incorrectly instructs a user to apply a properly forged 11 mm wrench to tighten a 10 mm hex nut. The wrench is fulfilling its design (it would have worked for 11mm), adheres to its design (properly forged), but is the wrong tool for the job. The IBM PCjr was a useful and properly functioning computer that was not quite what consumers really wanted. The 2003 Northeast Blackout, with its lack of industry-wide standards, also fits into this region.

Region A3: This region represents systems that are verified (adhere to their designs) and validated, but whose intended designs do not match any significant requirement. A significant requirement is one that is shared by the majority of principal stakeholders. These systems are often described as offering solutions in search of problems. A strange looking (and properly forged) wrench designed to tighten a non-existent nut is an example of a system in this region. Yes, this wrench might eventually satisfy the needs of a customer in a completely foreign context, such as use as a paperweight. However, it does not meet a requirement within the context of its original design (a tool for tightening a nonexistent nut). The weak glue created by 3M stayed in the A3 region for a long time until someone invented Post-it® notes. Their scientists studied the glue carefully, but 3M certainly did not have a requirement for weak glue. In the 1940s IBM, Kodak, General Electric, and RCA were offered the patents for what eventually became the Xerox photocopy machine, but they declined saying that there was no requirement to replace carbon paper.

The types B2 and B3 are unverified equivalents of types A2 and A3, respectively. Not only do these system designs either address nonexisting, incomplete, or incorrect requirements, but the systems also fail to adhere to their designs or fail to satisfy stakeholder needs in

Table IV. SRCM Type for the Famous Failures

Name	Year	RD	VER	VAL	SRCM Type
Most commercial systems		Yes	Yes	Yes	A1
Perpetual motion machine		No	NB*	Yes	C4
Titanic	1912	No	No	Yes	B2
Tacoma Narrows Bridge	1940	Yes	Yes	No	B1
Edsel automobile	1958	Yes	Yes	No	B1
War in Vietnam	1967-72	Yes	Yes	No	B1
Apollo-13	1970	Yes	No	Yes	B1
Concorde SST	1976-2003	Yes	Yes	No	B1
IBM PCjr	1983	No	Yes	Yes	A2
GE rotary compressor refrigerator	1986	Yes	No	Yes	B1
Space Shuttle Challenger	1986	Yes	No	No	B1
Chernobyl Nuclear Power Plant	1986	Yes	Yes	No	B1
New Coke	1988	Yes	Yes	No	B1
A-12 airplane	1980s	No	NB*	No	C2
Hubble Space Telescope	1990	Yes	No	Yes	B1
SuperConducting SuperCollider	1995	Yes	Yes	No	B1
Ariane 5 missile	1996	Yes	No	No	B1
UNPROFOR Bosnia Mission	1992-95	No	No	No	B2
Lewis Spacecraft	1997	Yes	Yes	No	B1
Motorola Iridium System	1999	Yes	Yes	No	B1
Mars Climate Orbiter	1999	No	No	No	B2
Mars Polar Lander	2000	Yes	No	Yes	B1
September 11 attack on WTT	2001	Yes	Yes	Yes	A1
Space Shuttle Columbia	2002	Yes	No	No	B1
Northeast power outage	2003	No	Yes	Yes	A2

*NB means system was not built.

the process. Exemplifying famous failures are listed in Table IV.

Our model also features several other regions of importance. Region C1 represents the set of all valid requirements that have no satisfying system designs. This region represents potential and realizable systems that will satisfy the stakeholder's needs. An affordable and efficient electric car—arguably within our technological grasp—represents one such system. In essence, the art and science of requirements development is the process of pairing requirements from this region to systems in region A1. Entrepreneurship thrives in this region.

Another important region, C4, denotes infeasible requirements. Naturally, there are no overlapping systems here, because technology prevents such systems from being built. A perpetual motion machine represents one such system. However, this category is important because research and development efforts are concerned with expanding the verified/unverified system boundary into this region. We believe requirements will logically and sometimes quietly move from this region into C1 and C2 as technology advances.

Region C2 represent requirements that are incomplete/incorrect and do not have an assigned design. These regions represent systems that could become troublesome inhabitants of the B2 region in the future.

Although system design is not a serial process, because there are many iterative loops, the ideal system design path starts at C3 (or perhaps even C4), goes up the column and then across the row to A1. Other paths are also possible. For example, a previously unnoticed feasible requirement may cause a system to quickly move from C1 directly across to A1. A prototype design process might see a system move from C3 up to C2, then back and forth several times from C2 to A2 until finally reaching the top row (and eventually A1).

Table IV summarizes some of our famous failures and categorizes each failure according to one of the model's ten regions. Two system generalizations were added to demonstrate membership in each important category.

Most of the failures of this report were chosen randomly. They were chosen before this paper was written and before the SCRM model was formulated. The exceptions were the systems added in Table IV that were not in Table II. The systems of this paper do not present an even distribution of types of failure or types of system. Furthermore, the distribution was not intended to reflect real life. Data from the Standish Group [1994] could be used to infer what real world distributions might look like. As an example of our discrepancy from the real world, consider that of our two dozen examples only one, the A-12 airplane, was canceled before any airplanes were built. Whereas the Standish Group Report [1994] said that for software projects that were seriously started, two-thirds were canceled before completion. Figure 1 shows the number of systems of Table IV that fit into each region of Table III.

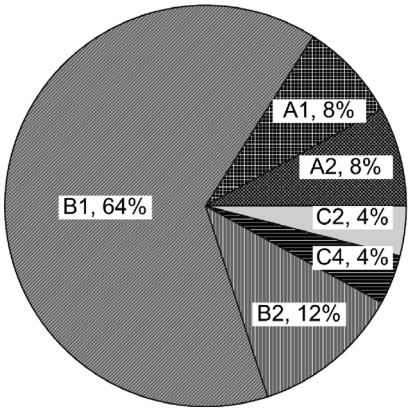


Figure 1. Number of systems from Table IV that are in each cell of the SRCM model.

6. WHAT COULD HAVE BEEN DONE BETTER?

We studied these famous failures and tried to answer the question, "What could they have done better?" Our answers are in Table V. Of course, like all generalizations about complex systems, our answers are not precise. But, nonetheless, they may be helpful. The conclusions in Table V are based on documents with much more detail than was presented in this paper.

7. LESSONS LEARNED

Is it important to develop requirements, verify requirements, validate requirements, verify the system, and validate the system? Yes. This paper has shown examples where failure to do these tasks has led to system failures. Is doing these tasks a necessary and sufficient condition for system success? No. Many systems succeed just by luck; and success depends on doing more than just these five tasks. Is it important to understand the difference between these five tasks? Yes. The CMMI is a collection of industry best practices, and it says that differentiating between these tasks is important. If you can distinguish between these five tasks, you will have a better chance of collecting data to prove that you do these five tasks. This paper has also shown some unique metrics that could be used to prove compliance.

However, there can be controversy about our consensus. Is getting the top-level requirement wrong, a

Table V. Failure Analysis Generalizations	
What could they have done better?	Famous Failure
Gain a better understanding of customer wants, desires and needs	IBM PCjr Edsel Coca Cola UNPROFOR Bosnia Mission
State the top-level problem	Vietnam A-12 airplane
Total system test	Hubble Space Telescope
Disseminate test results more widely	Hubble Space Telescope General Electric refrigerator
Engineering design, testing and verification	HMS Titanic Lewis Spacecraft Northeast power outage Chernobyl Nuclear Power Plant
Configuration management	Apollo 13
Quality control	HMS Titanic
Anticipate unknown problems when scaling up old designs	Tacoma Narrows Bridge Ariane 5 missile
Forecast technology	Motorola Iridium System General Electric refrigerator
Get politics out of business and scientific decisions	Concorde Supersonic Transport Space Shuttle Challenger SuperConducting SuperCollider
Get sufficient money	Chernobyl Nuclear Power Plant Mars Climate Orbiter Mars Polar Lander
Establish good corporate culture	Space Shuttles Challenger and Columbia

system-validation problem or a requirements-development problem? This issue provided the most contention in discussions about these famous failures. It prompted questions such as, "Should they have written a requirement that the Tacoma Narrows Bridge be stable in cross winds? Should they have written a requirement that the Challenger not be launched in cold weather? Should the Russian designers have told their Communist Party bosses that there should be a requirement for a safe design?" For the space shuttles, the top-level requirement was to use a recyclable space vehicle to put people and cargo into orbit. Was this a mistake? If so, what type?

Can we learn other lessons from this paper that will help engineers avoid future failures? Probably not. Such lessons would have to be based on complete detailed failure analyses for each system. Such analyses are usually about 100 pages long.

8. CONCLUSION

In this paper, we have sought to elucidate the frequently confused concepts of requirements development, requirements verification, requirements validation, system verification, and systems validation. After a brief terminology review, we inundated the reader with a casual review of two dozen famous failures. These examples were not offered as detailed failure analyses, but as recognized specimens that demonstrate how shortcomings in requirements development, verification and validation can cause failure either individually, collectively, or in conjunction with other faults. To further distinguish these concepts, we included the system and requirements classification model and several summary views of the failures—failures by discipline, failure generalization, and lessons learned. We hope our approach will promote understanding of terminology and improve understanding and compliance in these five critical systems engineering tasks.

ACKNOWLEDGMENTS

This paper was instigated by a question by Harvey Taipale of Lockheed Martin in Eagan MN in 1997. This paper was supported by Rob Culver of BAE Systems in San Diego and by AFOSR/MURI F4962003-1-0377.

REFERENCES

F. Andreatta, The Bosnian War and the New World Order: Failure and success of international intervention, EU-ISS Occasional Paper 1, October 1997, http://aei.pitt.edu/archive/00000667/.

Apollo 13, Imagine Entertainment and Universal Pictures, Hollywood, 1995.

A.T. Bahill and F.F. Dean, "Discovering system requirements," Handbook of systems engineering and management, A.P. Sage and W.B. Rouse (Editors), Wiley, New York, 1999, pp. 175–220.

Y. Bar-Yam, When systems engineering fails—toward complex systems engineering, International Conference on Systems, Man, and Cybernetics, 2 (2003), 2021–2028.

Y. Billah and B. Scanlan, Resonance, Tacoma Narrows bridge failure, and undergraduate physics textbooks, Am J Phys 59(2) (1991), 118–124, see also http://www.ketchum.org/wind.html.

W.L. Chapman, A.T. Bahill, and W.A. Wymore, Engineering modeling and design, CRC Press, Boca Raton FL, 1992.

Chernobyl-1, http://www.nrc.gov/reading-rm/doc-collections/fact-sheets/fschernobyl.html.

Chernobyl-2, http://www.infoukes.com/history/chornobyl/zuzak/page-07.html.

Chernobyl-3, http://www.chernobyl.co.uk/.

Chernobyl-4, http://www.angelfire.com/extreme4/kiddofspeed/chapter27.html.

M.B. Chrissis, M. Konrad, and S. Shrum, CMMI: Guidelines for process integration and product improvement, Pearson Education, Boston, 2003.

Columbia Accident Investigation Board Report, NASA, Washington, DC, August 2003, pp. 121–123.

R. Davis and B.G. Buchanan, "Meta-level knowledge," Rule-based expert systems, The MYCIN Experiments of the Stanford Heuristic Programming Project, B.G. Buchanan and E. Shortliffe (Editors), Addison-Wesley, Reading, MA, 1984, pp. 507–530.

D.W. Deal, Beyond the widget: Columbia accident lessons affirmed, Air Space Power J XVIII(2), AFRP10-1, pp. 31-50, 2004.

D. Dorner, The logic of failure: Recognizing and avoiding errors in complex situations, Peruses Books, Cambridge, 1996.

Edsel, http://www.failuremag.com/arch_history_edsel.html.

H.L. Fenster, The A-12 legacy, It wasn't an airplane—it was a train wreck, Navy Inst Proc, February 1999.

R.P. Feynman, Surely you are joking, Mr. Feynman, Norton, New York, 1985.

A.G. Hessami, A systems framework for safety and security: the holistic paradigm, Syst Eng 7(2) (2004), 99–112.

R. Kunzig, Europe's dream, Discover 18 (May 1997), 96–103.

Lewis Spacecraft Mission Failure Investigation Board, Final Report, 12, NASA, Washington, DC, February 1998.

Mars Program Independent Assessment Team Summary Report, NASA, Washington, DC, March 14, 2000

J.A. Moody, W.L. Chapman, F.D. Van Voorhees, and A.T. Bahill, Metrics and case studies for evaluating engineering designs, Prentice Hall PTR, Upper Saddle River, NJ, 1997.

NASA Faster Better Cheaper Task Final Report, 2, NASA, Washington, DC, March 2000.

H. Petroski, To engineer is human: The role of failure in successful design, Random House, New York, 1992.

Standish Group International, The CHAOS Report, 1994.

J.P. Stevenson, The $5 billion misunderstanding: The collapse of the Navy's A-12 Stealth Bomber Program, Naval Institute Press, 2001, Annapolis, MD.

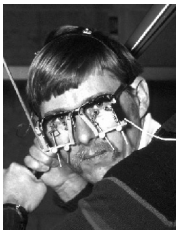
Tacoma1, http://www.enm.bris.ac.uk/research/nonlinear/tacoma/tacoma.html#file.

Tacoma2, http://washington.pacificnorthwestmovies.com/TacomaNarrowsBridgeCollapse.

M. Talbott, Why systems fail (viewed from hindsight), Proc Int Conf Syst Eng (INCOSE), 1993, pp. 721–728.

Titanic, a Lightstorm Entertainment Production, 20th Century Fox and Paramount, Hollywood, 1997.

E.R. Tufte, Visual explanations: Images and quantities, evidence and narrative, Graphics Press, Cheshire, CT, 1997.



Terry Bahill is a Professor of Systems Engineering at the University of Arizona in Tucson. He received his Ph.D. in electrical engineering and computer science from the University of California, Berkeley, in 1975. Bahill has worked with BAE SYSTEMS in San Diego, Hughes Missile Systems in Tucson, Sandia Laboratories in Albuquerque, Lockheed Martin Tactical Defense Systems in Eagan, MN, Boeing Integrated Defense Systems in Kent WA, Idaho National Engineering and Environmental Laboratory in Idaho Falls, and Raytheon Missile Systems in Tucson. For these companies he presented seminars on Systems Engineering, worked on system development teams and helped them describe their Systems Engineering Process. He holds a U.S. patent for the Bat Chooser, a system that computes the Ideal Bat Weight for individual baseball and softball batters. He is Editor of the CRC Press Series on Systems Engineering. He is a Fellow of the Institute of Electrical and Electronics Engineers (IEEE) and of the International Council on Systems Engineering (INCOSE). He is the Founding Chair Emeritus of the INCOSE Fellows Selection Committee. This picture of him is in the Baseball Hall of Fame's exhibition *Baseball as America*.

Steven J. Henderson received an M.S. in Systems Engineering from the University of Arizona in 2003. He is a Captain in the U.S. Army, and is currently serving as an Instructor of Systems Engineering at the U.S. Military Academy at West Point, New York. He graduated from West Point in 1994 with a B.S. in Computer Science. He is currently pursuing a Ph.D. in Systems and Industrial Engineering from the University of Arizona. He is a member of the Phi Kappa Phi honor society, the Institute for Operations Research and Management Sciences (INFORMS), and the American Society of Engineering Educators (ASEE). He was a NASA Faculty Fellow in the summer of 2004.