Designing Software Systems for Sustainability

By
Francis David Kenny Jr.

A MASTER OF ENGINEERING REPORT

Submitted to the College of Engineering
Texas Tech University in
Partial Fulfillment
of the Requirements for
the Degree of
Master of Engineering
Approved

Dr. J. Borrelli

_____

Dr. A. Ertas

_____

Dr. T. T. Maxwell

_____

Dr. M. M. Tanik

_____

October 12, 2002

Acknowledgements

It is impossible to acknowledge all the influences that have made this paper possible. Space and limitations on the author's memory, rather than by a lack of appreciation, dictate any omissions. My family, most immediately Nancy, Amanda, Christopher, and Sarah, have always supported me in my education, even when I was at my most frustrated. I must thank Liliana for disproving the theory that no human endeavor can result in perfection and simultaneously reminding me that I have a long way to go. Doctors Ertas, Tanik, and Maxwell, with the rest of the Raytheon - Texas Tech Faculty, have all provided expert instruction, support, and advice. My colleagues and predecessors in the Raytheon Texas Tech program, Steve Nelson, Jo Alamares, Robert Price, Tom Kollman, and others have all provided their support and encouragement. My former co-workers on the ASP program Brian Shrock, Chris Womack, Glenn Kneese, Julius Rahmandar, Joan Umbricht, Becky Keller, Daniel Blaisdell, Ronda Dillard, and especially John Lively variously system provided knowledge, polarized examples, inspiration, references, motivation, guidance, and leadership in the team's difficult path toward process improvement.

My classmates deserve special thanks for their support. Brenda Terry of Raytheon at McKinney also provided invaluable support and assistance.

My formal and practical maintenance management education began in the U. S. Air Force. My best wishes, respect, and gratitude go to those of my former comrades in arms who influenced my training, opinions, and career. *Illegitimus nil carborundum.*

Designing Software Systems for Sustainability

Table of Contents

List of Figures

List of Tables

Abstract

Software development organizations are oriented to delivering the product to the customer. All available resources and organizational processes are concentrated on the definition, design, implementation, integration, and delivery of the product, often ignoring maintenance. Perhaps motivated by business reasons, or by the poor image held by maintenance activities, software maintenance is often an ad hoc activity. Unplanned, disorganized maintenance activities are more expensive for the user and developer.

Software is subject to error and the need for modification after delivery. Software designed to be sustainable must take advantage of modern languages and software management practice, and should have a system-oriented maintenance infrastructure.

Definitions, Terms, Acronyms, and Abbreviations

ASP	Algorithmic Signal Processor
CORBA	Common Object Request Broker Architecture. A system by which programs make use of stored, well-defined, and standardized system objects over a network (as opposed to local duplication of resources).
COTS	Commercial, Off-The-Shelf. This term usually refers to commercially built, general-purpose processing platforms and other hardware.
CRT	Cathode Ray Tube
DR	Discrepancy Report. ASP program terminology for field reports of software operations problems. Pre-delivery software problems or "bugs" were sometimes also referred to as DRs.
Freeware	Freeware is free software, usually distributed through the internet.
FTP	File Transfer Protocol. A simple method allowing copying of entire software files between networked computer systems.
GUI	Graphical User Interface
IMP	Raytheon Integrated Master Plan
IMS	Raytheon Integrated Master Schedule
IPDS	Raytheon Integrated Production Design System
LCD	Liquid Crystal Display
LED	Light Emitting Diode
Open Source	Members of and contributors to the Open Source Foundation
RCS	Revision Control System. An open source SCM tool, similar to SCCS, with better controls for multiple, simultaneous, edits of the same code module.
RMSS	Reliability, Maintainability, Sustainability engineering discipline, part of the IPDS.
SCM	Software Change Management. Alternately, may be System Change Management or Source Code Management. In the usual usage and for the purposes of this paper, applies to software source and compiled code version control, usually implemented through a specialized database application.
Shareware	Software that is marketed and downloaded through the internet. Fees may be voluntary, or when paid may allow for increased functionality.
Shrink-wrapped	Usually "shrink-wrapped software," refers to commercially available software that is often found in retail outlets and packaged in cellophane-wrapped cartons. Desktop systems use most shrink-wrapped software packages.
SSCS	Source Code Control System -- A common, simple, file-system based SCM database tool.
Y2K	Abbreviation popularly given to the Year 2000, most often in the context of predicted software system process failure at the turn of the century due to incorrect programming practices.
ASP	Algorithmic Signal Processor
CORBA	Common Object Request Broker Architecture. A system by which programs make use of stored, well-defined, and standardized system objects over a network (as opposed to local duplication of resources).
COTS	Commercial, Off-The-Shelf. This term usually refers to commercially built, general-purpose processing platforms and other hardware.
CRT	Cathode Ray Tube
DR	Discrepancy Report. ASP program terminology for field reports of software operations problems. Pre-delivery software problems or "bugs" were sometimes also referred to as DRs.
Freeware	Freeware is free software, usually distributed through the internet.
FTP	File Transfer Protocol. A simple method allowing copying of entire software files between networked computer systems.
GUI	Graphical User Interface
LCD	Liquid Crystal Display
LED	Light Emitting Diode
Open Source	Members of and contributors to the Open Source Foundation

RCS	Revision Control System. An open source SCM tool, similar to SCCS, with better controls for multiple, simultaneous, edits of the same code module.
SCM	Software Change Management. Alternately, may be System Change Management or Source Code Management. In the usual usage and for the purposes of this paper, applies to software source and compiled code version control, usually implemented through a specialized database application.
Shareware	Software that is marketed and downloaded through the internet. Fees may be voluntary, or when paid may allow for increased functionality.
Shrink-wrapped	Usually "shrink-wrapped software," refers to commercially available software that is often found in retail outlets and packaged in cellophane-wrapped cartons. Desktop systems use most shrink-wrapped software packages.
SSCS	Source Code Control System -- A common, simple, file-system based SCM database tool.
Y2K	Abbreviation popularly given to the Year 2000, most often in the context of predicted software system process failure at the turn of the century due to incorrect programming practices.

Chapter 1
Introduction

Software development processes are the subject of much interest and ongoing research. Most software management theory is well developed and described, but practice continues to vary from the ideal.

This report was inspired by the case of a software version control and maintenance management system re-designed and implemented in the maintenance phase of a long-term software project. Although development took place over a two to three-year period, the working philosophy of the developers was similar to that of rapid prototyping. Many process and quality shortcuts were taken for the sake of the delivery schedule. The urgency of the delivery and integration phases, with design changes caused the abandonment of the software change management plan. Under pressure, software developers circumvented software code approval processes, and the schedule limited the time allotted to pre-delivery. The same design changes caused a redesign of the system that was never fully documented or re-planned. Rather than replacing the discrepancy reporting system, the development team abandoned this essential maintenance tool. These development shortcuts created difficulties in the maintenance phase of the project life cycle.

Users expect long service life from software systems. Consider that a complex software application (or suite of applications) that provide the customer with an essential data processing service may require a substantial amount of time to create and deploy, but may be in service for five, ten, or more years. However, a long service life requires that the software undergo maintenance. Indeed, a perceived potential flaw in legacy applications, some of which had been in use for decades, caused the Y2K crisis of recent memory. To prevent the occurrence of massive system failures at the beginning of the millennium, in-house and contract programmers made expensive, sometimes Herculean efforts to locate, repair, or mitigate the effects the faulty code. The results of the many separate emergency software maintenance efforts were that software failures caused no major interruptions to *essential service*s – to the apparent disappointment of some doomsayers. In fact, the services dependent on the potentially vulnerable software worked so well after the turn of the century that many expressed doubt that there had been an actual risk.

The Y2K effort is an example of emergency maintenance performed to avoid an anticipated pitfall. In the case study project, concentration on the speed of development and deployment, with little thought to the operational life of the project adversely affected maintenance efficiency. Many of the mistakes made in the project, although

classic in software development circles, were customer-driven and controlled through the budgeting process (this is also a "classic mistake" [McConnell, 1983]). Project management, representing both technical and business needs, were aware of the pitfalls, but were unable to avoid them or compensate for their effects.

A recent bulletin from Raytheon's company Six Sigma program announced a success in correcting similar problems in another project, using many of the same methods employed in the case study, proving that the experiences with the case study project are not unique [Raytheon Mission Systems Six Sigma quarterly, Volume, 1 Issue 1; August 2002]. The problems with post delivery software support are also not unique to the company. despite the warnings suggestions for avoiding the pitfalls provided by authorities like Royce [1970], Brooks [1975], Boehm [1981] (in two cases, with U. S. Air Force case history as source material), almost 30 years later Daich [1996] describes the continuing need for an improved software development validation process (also as a result of an Air Force study).

Moreover, much of the literature available focuses on the software application itself, ignoring the program interaction with the user, the system hardware, system configuration, networks, or other "external" factors. Yet to provide for system-level sustainability the maintenance management system must support the entire application environment. A NASA case study [Ramamoorthy, 1996] describes a software control system that *evolved* to provide the basis for historical performance, failure analysis, failure correction, failure probability analysis, and process improvement. A controlled software maintenance process, designed and implemented as an integral part of the initial software application design, would provide a software system an infrastructure for support throughout the software lifecycle.

Through centuries of development and evolution, mechanical processes, methods, and techniques are well developed and reasonably efficient. Automated systems use computerized processes to provide even greater efficiencies over their mechanical predecessors, enabling new capabilities. However, computer systems of hardware and software are necessarily more complex than mechanical processes, and therefore more prone to error. Using Boehm's definition:

> *Software engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation. [Boehm, 1981]*

Further, we can extend this definition to state that software engineering techniques create computerized applications that provide an *essential service* to the user. Users will choose a reliable mechanical or manual system over an inefficient and error-prone computerized system. The concept of an *essential* service is situational and relative – the computerized service may be entertainment or a life-critical real time system – but it is a service *useful to man* for which the user feels a distinct need.

Chapter 2
Background

2.1 Literature Search

Academic, government, and commercial researchers have studied software development, software project management, software cost estimation, and maintenance management extensively. In spite of the research, software projects continue to be difficult to plan, implement, and control.

In his classic work *The Mythical Man-Month*, Fred Brooks [Brooks, 1995] described the failure of a software management process, and drew conclusions from his analysis of the case. The original edition, published in 1975, provided background and other material for much of the research in software development and management theory. This updated version contains further perspectives after 20 years, including the assertion that the problems outlined in his original case study still exist in the software industry.

Ramamoorthy [1996] describes an ideal software support system that evolved in the development of the software that controls the NASA Space Shuttle. The NASA/IBM/Loral production, test, and verification model serves as an example and template for the complete Software System Infrastructure.

Royce [1970] first described the waterfall model of the software lifecycle. Later, Boehm [1981] incorporated Royce's reiterative "do it twice" cycles into the model, first describing Operations and Maintenance as a separate stage in the cycle. Boehm's ideas later [1998] evolved into the spiral model of constant development that underlies most modern production methods.

McConnell [1993 and 1996] provides a roadmap for how to manage code, the programming team, and programming processes to deploy quality software products. McBreen [2002] (inspired in part by McConnell) describes a method for applying the model of craftsmanship to small-project software processes, developer training, and ethics. While eschewing written, standardized processes and the concept of Software Engineering, McConnell does advocate a systematic method of refining software requirements through an intimate customer-developer relationship. Another study by Ramamoorthy [2000] reinforces the concept of Software Engineering as a service with a slightly different comparison to craftsmanship and cottage industry as information industry service providers. Dr. Ramamoorthy emphasizes the need to apply design principles to Software Engineering services to meet customer needs.

Trewn and Yang [2000] describe design reliability in complex systems, defining the levels of failure and the interdependence between components.

Alamares [2001] describes the need for a documented software testing process within Raytheon's integrated system of design and production. She also comments on software complexity and on the connection between test process and the development of software performance metrics.

Analyzed together, these references support the need for a software support infrastructure. The support infrastructure created for maintenance of other engineering products can supply a model for an integrated system that will sustain the software product to extend its useful life span and to decrease the costs of post-delivery software maintenance.

Chapter 3
Case Study

3.1 Project Description and History

This report will refer to the software project using the acronym ASP, for Algorithmic Signal Processor. ASP is a port to COTS (commercial, off-the-shelf) hardware of a specialized, government-sponsored signal-processing program. The company previously developed applications similar to ASP using specialized, custom-built hardware fully integrated with custom software. The original ASP implemented the same signal processing algorithms and controls in software using COTS hardware in a stand-alone application similar to the embedded application. Later iterations of the project implemented the same algorithms and controls in an integrated control environment with other similar applications. Another contractor developed the integrated application, which introduced interface dependencies on ASP software for display, control, and data transmission. At the time of this writing, there are two different variations of ASP installed at four separate customer locations. A third variant has recently been decommissioned.

From March 2000 through January 2002, the author's work officially consisted of simple functional testing of software changes to ASP on the factory test beds, tracking of user-reported "bugs" through Discrepancy Reports (DRs), the generation of software upgrade packages, and the maintenance of some system documentation. In practice, however, the author became the SCM manager for ASP, ensuring that the maintenance team delivered software changes to the correct software baselines. Another worker, who had the responsibility of administering and maintaining the SCM system software, was only committed to ASP support for a very limited time per week – severely limiting the effectiveness of the SCM application.

3.2 Methods of Software Control

During the development of ASP's predecessor, the programming team used an open source program called RCS to track software changes. However, due to customer concerns about open source software, ASP switched to an off-the-shelf, text-based system called SCCS that was included in the standard SUN operating system as part of the development platform.

The developer chose not to spend much time developing a shell for SCCS. A shell, or group of interface programs, would have allowed the development group to access the software baseline in a similar fashion to that of

RCS – the "legacy" SCM software. Instead, the planned maturity path used three different software baselines called WORK, BUILD, and BASE. SCM system designers intended the WORK baseline to be consistent with the last successful software compile. WORK was also the modifiable version that programmers would use for reference during development. The BUILD baseline was identical to the last complete WORK baseline, but included all working executables. The programming team used the BUILD baseline for testing. The version called BASE was to be the ready-for-delivery baseline. As WORK was compiled successfully, it was copied over BUILD, and as testing was completed all changes approved, the BUILD baseline was copied over the BASE version (see figure 1).

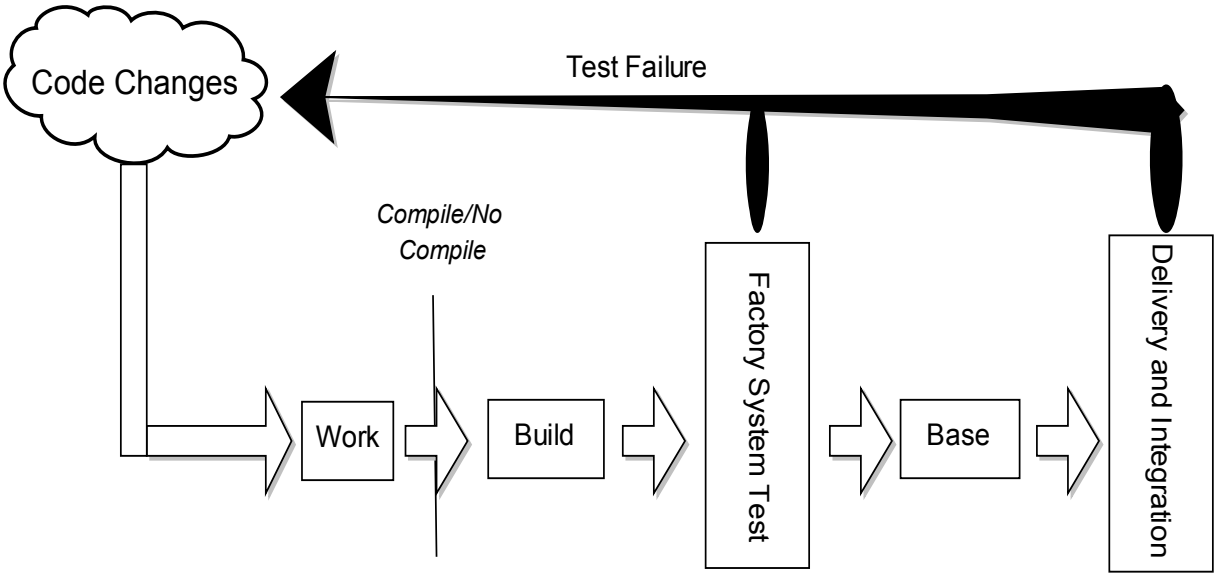


Figure 1 Original ASP Code Flow

During development project requirements changed constantly. As engineers incorporated the requirements for the application variations into ASP design, developers changed the basic SCM plan, creating a separate baseline for each projected delivery configuration. Changes took place concurrently with site deliveries and integration, taking the software system from the deployment to the maintenance phase of the software development lifecycle. The adaptations made to meet the delivery schedule further confused the relationship between the baselines. There was no clear path for incorporation of new baseline changes, and several customized baselines were supposedly available simultaneously, even though each was often at a different stage of development. System configuration for testing and deployment became even more complex (see figure 2). The lack of SCCS shell tools limited the ability to track an individual change into all pertinent baselines, and control of the disparate baselines was limited.

After system delivery, the development code approval process was also restricted. Software code Configuration Item (CI) lead engineers were no longer required to approve changes to the various baselines before introduction to the SCM system. Instead of code approval process, the maintenance programmers used an uncontrolled code-and-fix system. Because of the lack of a factory system test-bed that resembled the field configuration, integrators accomplished the real system test using systems in the actual operational environment. Several deliveries of faulty code reduced the user confidence in the system and the maintenance team.

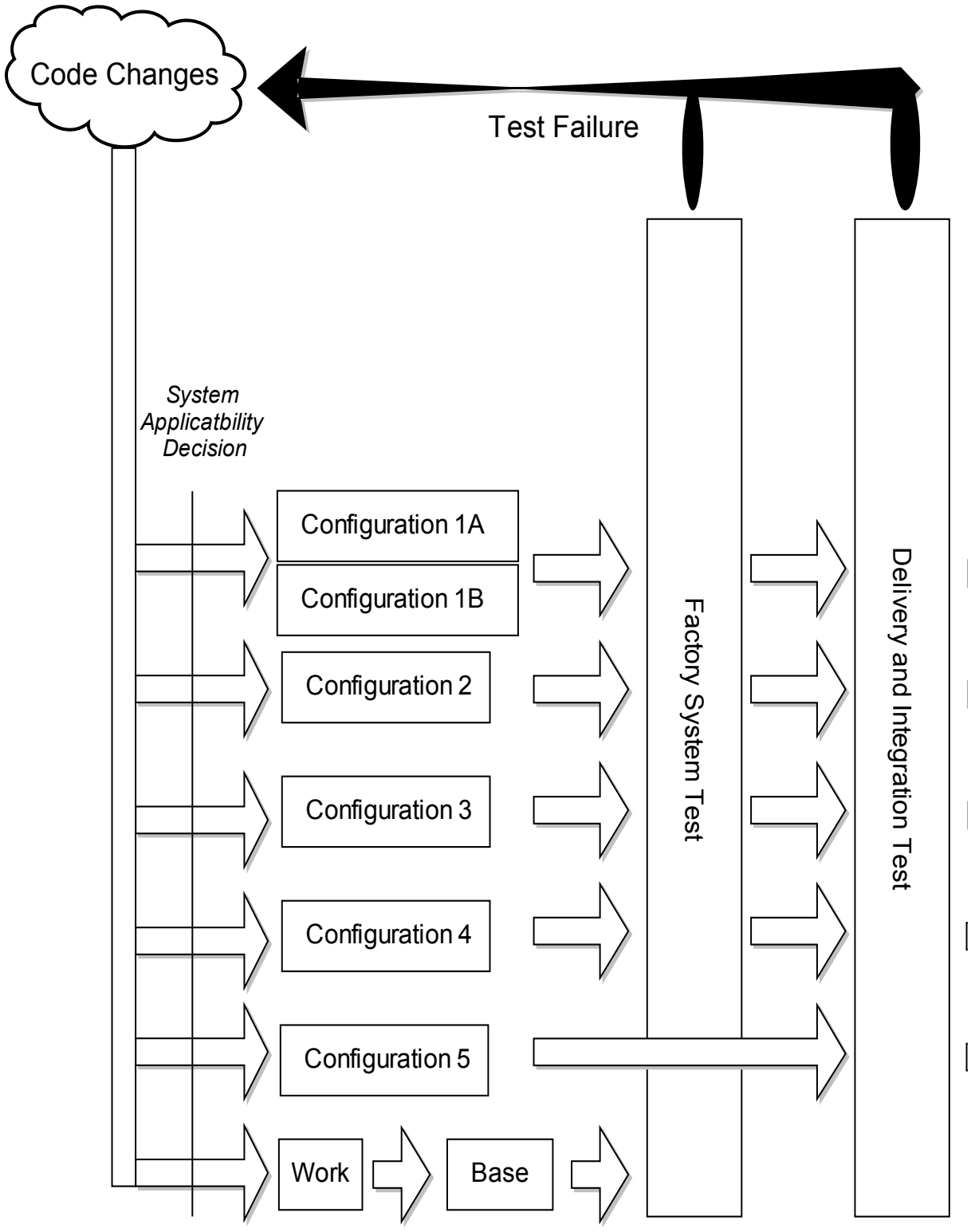


Figure 2 Ad Hoc ASP SCM Code Maturity Flow

The software design did not support the abstraction of site hardware configuration. The SCCS software tool could have supported several branches in the original SCM scheme, but the new scheme bypassed many of the tool's features. No clear path for software development existed, and the system applied very little actual control to the software product. The relationship between the actual problems, problem reports, software changes, and individual baselines were unclear and difficult to understand. Maintainers delivered software baselines to specific sites untested, and/or without the intended software changes. Original documentation of the system was limited, and documentation of changes and fixes was often incorrect or non-existent. Baseline upgrade was difficult, and required careful and detailed analysis and control which were often impossible under an unpredictable fix delivery cycle.

System engineers assigned to maintain the software developed complex delivery scripts and multiple configuration files, which were also under SCM control, to handle the possible combinations of deployed hardware and software. Minor changes to site hardware and systems necessitated complex and laborious script and configuration changes, as well as error prone software check-in to various baselines.

The imposition of limitations on personnel and time in the late stages of development exacerbated the problems caused by the lack of a truly representative SCM model and support system. Budgeting limited SCM personnel support to only 20 hrs per week at best, limiting the involvement of knowledgeable SCM technical expertise to consultation on the design of the original system, the support of software builds, and file system maintenance. SCM experts were not involved in the change to the new maturity model, and their ability to maintain and stay current with the system was limited by their available time.

The lack of a user software problem reporting system also affected software maintenance. During development and early deployment / integration, the factory used a text-based discrepancy report (DR) database system. The software was not Y2K compliant. Since the scheduled project sell-off was to occur just at the end of the millennium, rather than replacing the DR system, database maintainers stopped updating the database and dumped the data records to text files. System users had no consistent method for reporting problems. DRs filtered back to the system maintainers through the customer representative and prime contractor, with little direct contact. Maintainers received duplicate, unverified, and/or inconsistent reports of system failures, many not attributed to a specific installation or system and without supporting data. Maintainers did not have a consistent picture of field performance.

Once delivery had occurred and the customer had accepted the software, the programming team moved on to the follow-on project. While the schedule called for further delivery of system improvements in conjunction with other contractors' upgrades to their applications and hardware, ASP development essentially ceased. The internal software approval process, thus far only loosely applied to facilitate the rapid delivery schedule, was now dropped completely.

3.3 Redesign of ASP Maintenance Processes

The maintenance team redesigned the software SCM system with the following points in mind:

- Simplify the software change process
- Provide better tracking and documentation of software changes
- Change the software maturity model to reflect the actual process

Maintainers implemented changes to the system in several parts. Simplification of the delivered product system from five separate software baselines into two separate product lines created an integrated environment product and a standalone product (see figure 3). The original baseline model BUILD, WORK, and BASE source code was archived and removed from the file system, as were the now-redundant configuration-specific baselines. The new product baselines receive new software check-ins from developers, but the software build process captures a complete snapshot of the entire baseline, time-stamping the resulting software build.

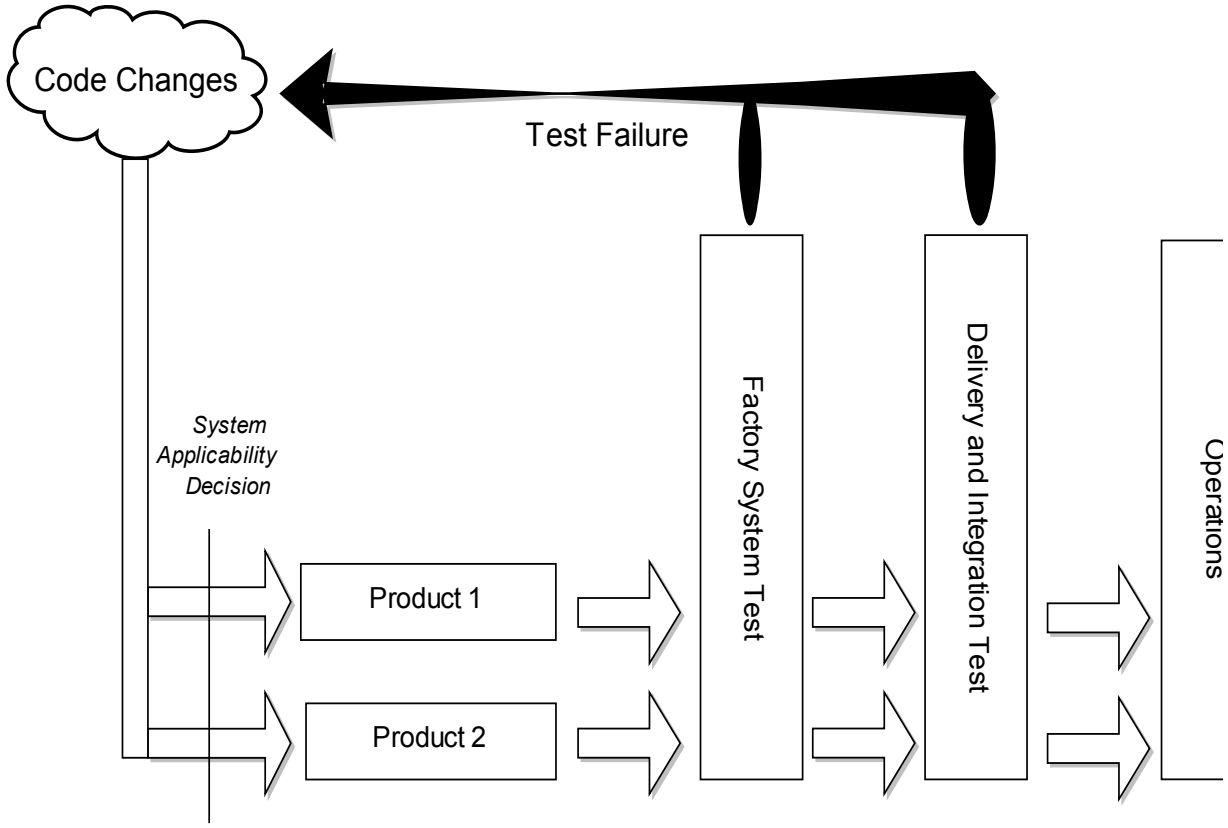


Figure 3 Redesigned SCM Code Maturity Flow

In conjunction with the prime contractor, maintenance engineers also created an integrated, web-based software bug database and reporting system using off-the-shelf tools already in place for the follow-on project. All user-created DRs as well as developer created DRs are stored in the system. As part of the changed SCM systems, developers can only checkout and deliver changes to a software source file when there is a DR for the change, and when reviewers assign the DR to the specific developer. All other developers must use copies of the source code. Developers are responsible for merging their changes to those of other developers.

3.4 Comparison

The redesign of ASP has had several positive results. Once a verified a software change becomes available, it is applicable as a patch or upgrade to all the similar deployed systems. The system changes result in simplified documentation and delivery preparation: a single FTP delivery package is applicable and available to all

the affected sites. The change allows simplified documentation, requiring a single memo for all applicable deliveries, rather than creation of a customized version of a system change document for each different installation.

In addition to the time saving at delivery, the change saves more effort and time due to the streamlined check-in processes. A useful software change requires check-in to, at most, two baselines, verses six in the previous configuration, which also reduces the overhead load of regression test prior to delivery. In addition, the implementation of the software check-in linked with the DR system allows maintainers to understand easily which changes are included with which software baseline -- a relationship that before had been a difficult to establish.

The time-stamped and archived baselines of the new system give the maintenance team the ability to get snapshots of compiled code easily. This feature allows checks of baseline performance against previous versions of the software -- a feature that was missing previously. Before the change, unless maintainers delivered code changes to the customer, the SCM software overwrote the superseded compiled code upon every build. After the change, the SCM administrator may now choose to keep undelivered versions of ASP, allowing incremental performance comparisons.

An additional benefit of time-stamped versions is a function of the way the ASP software logs errors. The error log includes the complete directory path to the software source code that contained the error. Because the directory path name now contains the timestamp, error messages written to the log contain embedded version information: an important factor in troubleshooting. Previously, an error log from the field was only identifiable by the installation site -- no ASP version information was immediately available.

System users began to benefit from the incremental changes immediately. Implementation of a fix sometimes now takes place before a given site encounters the problem -- incorporation of the DR repair from another site meant that the fix was already included in a system patch or complete baseline upgrade. DR status and prioritization became more available to the user because of the integrated DR reporting system. The new system delivers software changes more quickly, and software baselines are less dependent on site configuration, because of the simplified baseline model.

Unfortunately, the redesign of ASP SCM is still missing some much-needed features. Design and requirements information are still not readily available. A more complete system would have automated much of the documentation processes. SCM still does not provide a simple way to capture performance data and test results.

A well-designed SCM system, implemented early in the development cycle, could allow the capture of much information that is still now lost.

The reason frequently given for the acknowledged problems with ASP was the "rapid" deployment environment: developers considered good software development control practices and SCM too restrictive to fast development and on-schedule delivery. However, the limited resources assigned to software management with the poorly organized maintenance undercut the customer and user's perception of quality, required many hours and much attention by non-SCM personnel to understand and maintain, and unnecessarily complicated software maintenance.

3.5 Requirement for Support System

In ASP, the customer paid for maintenance at a "level of effort" rate, meaning that within the maintenance budget, the customer reimbursed the developer for the hours spent maintaining the software, plus a fixed profit. The company could have negotiated to receive more income for the increased hours charged to maintenance when customer budget was available. In this scenario, the company would have made greater profit by assigning more programming time and by allowing more time for testing and documentation. Instead, the maintenance team's struggle with the SCM and bug report systems consumed much of the maintenance budget, rather than by effective troubleshooting, coding, and repair.

Poorly implemented SCM affects software maintenance in fixed cost scenarios in a similar way. Maintenance activities reduce the developer's potential profit when maintainers spend most of the time managing SCM, reverse engineering uncontrolled software, or modifying software-testing tools.

Shrink-wrapped software applications also require good SCM. Additionally many shareware, freeware, and open source projects use an SCM product to control system maintenance. If developers do not maintain a good system of SCM and control, maintainers expend much of the allotted resources on software management functions. A good SCM implementation prevents redundant development and allows better control of delivered product.

Often Systems Engineers and project managers bemoan the perceived disconnect between the customer and the system user. In most software projects, the customer defines the requirements with the developer and makes the purchase. The system users are often a different group of people, with potentially quite different needs than the customer. While the customer may approve and pay for the initial software delivery, eventually the software developer must try to meet the user's requirements. In turn, the user's opinions of software/system quality influence

the customer's response to the developer. System users will not endorse the purchase of products and support from the developer if they are unimpressed with system quality. The developer-maintainer's response to the user's needs in the maintenance process directly affect the user's perception of quality. Good SCM as a part of a good software support infrastructure allows maintainers to implement quality maintenance.

3.6 Infrastructure for Sustainability

Software maintenance and version control should be closely linked with documentation, training materials, troubleshooting, test, and performance measurement, all the major parts of system support infrastructure. In this context, documentation includes deliverable and internal design and process documents as well as software source-code documentation. Deliverable documentation may be user manuals or on-line help, design and interface specifications, and performance data.

Software maintainers are seldom the same individuals who develop software. Innovative and capable software designers and implementers are usually in great demand for those talents, and migrate to projects where they can perform those functions after a software application is developed, delivered, and integrated. Software maintenance is a less glamorous task than development. The software maintainer is a developer with a specialized skill set: she/he must be able to interface with the system users, understand the operation of entire system, have the ability to reverse-engineer and to debug legacy code, and be able to develop test procedures that test software fixes on the component and integrated product levels. Providing software maintainers with the tools to perform these functions is a necessary part of the software system infrastructure.

The infrastructure should compensate for the lack of staff continuity by making historical development information – software meta-data – available to maintainers. Software maintainers need access to the original design requirements from the customer. In the ASP system, system engineers categorized a user-requested change as either a maintenance issue or a new development requiring additional funding – sometimes somewhat arbitrarily. The intent of the “new development” classification was to obtain additional funding for any feature not part of the original system requirements. However, the only requirement documents available to the maintainers were those originally delivered as part of the design process. The design documents were paper-based and had not been maintained or upgraded as the original project changed. Actual, as-built design and requirements documentation was not available.

Because the company wanted to maintain its good working relationship with the customer, the customer arbitrated almost all maintenance verses development classification conflicts. Often in ASP, this meant that easy changes to the applications were not made, resulting in users not getting the functional changes they needed and the company loss of the opportunities to obtain income by generating new software functionality. In other cases maintainers were required to develop new functionality at no greater cost to the customer-- in effect, re-engineering parts of the product for only the pre-arranged cost of maintenance.

It is axiomatic in systems engineering that the reason to develop from set customer requirements lets the designer know when the work is finished. However, it is perhaps less obvious that the principle also applies to the post-delivery lifecycle of the product. The presence and availability of requirement documentation would have saved the ASP project development costs and could potentially have provided more income through an awareness of new business opportunities.

3.7 Tools and Support Infrastructure

Traditional "hardware" engineering relies on infrastructure for support and sustainability. Automotive and aircraft manufacture, architecture and civil engineering, and mining are all engineering fields that depend on standardized, in-place, and sometimes overlapping tools, skill sets, and functions to support post-deployment maintenance. Software and Systems engineering should be no different than the more traditional engineering disciplines. Just as an automobile manufacturer counts on the existence of competent mechanics, gas stations, and parts stores, a software product design should integrate with an existing support infrastructure for post-deployment maintenance. If an appropriate maintenance service is not available using an existing infrastructure, the system design must also provide this service.

3.8 System Focus

The development of a maintenance system for ASP also showed the need for a system viewpoint. While ASP was a single family of applications, every installation was different and customized. Each location with ASP used it slightly differently, and local operators, system administrators, and maintenance technicians had different operating procedures, system awareness, work methods, and organizational politics. From the factory maintenance perspective, each installed system had unique problems created only in the context of the convergence between the application, its system environment, local management, and the users. The maintenance system had to support not

only the hardware, but had to consider system configuration (and the consequences of systemic problems) and the human factors that contributed to both the problems and the repairs.

In ASP, the composition of the maintenance staff was also a factor in the effectiveness of maintenance efforts. Several engineers assigned to the project were inexperienced with the application, and received ad hoc training, often becoming self-educated in the use of the ASP application and the tools available to manage software source code. While the system needed much programming effort in many areas, the budget allowed only limited, disjointed, and loosely organized software programming and troubleshooting. Only when a programmer, who was highly skilled in debugging and troubleshooting, was dedicated to the project full-time were maintainers able to identify and change most of the specific software errors. Only when system engineers implemented the DR database was there an ability to organize, prioritize, and focus the maintenance effort on the real "bad actors" in the system. Only after implementation of the new SCM process was the software source code fully under managed control. Only when management committed to the change was any effective effort made to fix the maintenance system. The system model for maintenance must therefore include the people who are involved in the work, and must encompass personnel organization, training, and management.

Chapter 4
The Software and System Lifecycle Models

Software maintenance may be underemphasized because the terminology has changed. The original Waterfall model [Royce, 1970], listed eight software life stages or levels, with retirement as an ultimate condition (see figure 4). Each level of the original waterfall modes had two purposes or names, one for a recursive re-entry into the preceding stages, and each stage in the sequence required either retirement or reentry into a preceding stage. The eighth level, "Operations and Maintenance" was included with "Revalidation." Later, [Boehm, 1981] refined the model with nine sub goals, describing maintenance phase as

> "…a *fully functioning update of the hardware-software system. This subgoal is repeated for each update."*

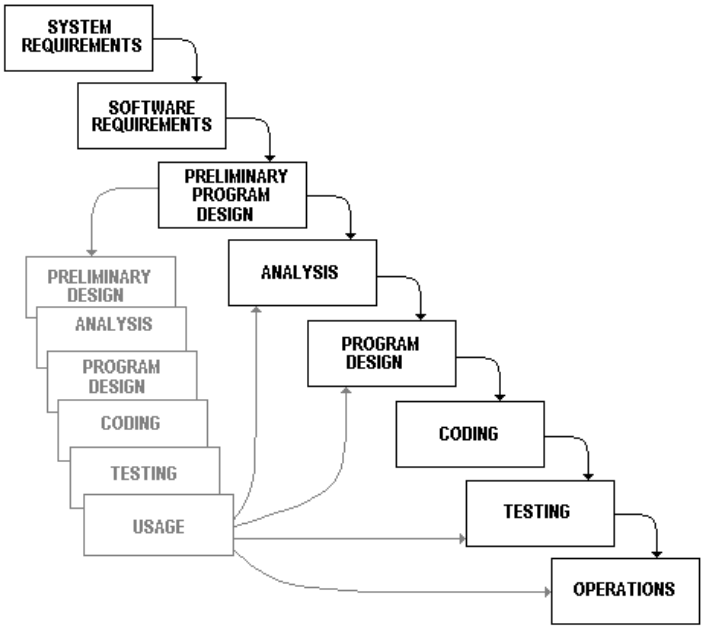


Figure 4 Royce's Original Waterfall Model [Royce, 1970]
Source: http://www.informatik.uni-bremen.de/gdpa/def/def_w/WATERFALL.htm

Boehm [1981] refined the waterfall model, showing that each stage is the beginning of an iterative and incremental cycle that includes all of the stages, including operations and maintenance. Still later, based on his change to the waterfall model, [Boehm, 1998 and 2000] describes and builds on a complex spiral model of software lifecycle (figure 5), turning maintenance into the continuous, iterative development of new products as changes to the preceding delivered product. In the continuing spiral model, Boehm recognizes the complexity of software maintenance coupled with the concept of continuous development, but falls away from the terminology of

maintenance, service, and operations. Rather than maintenance, Boehm's efforts refer to the continuous *enhancement* of software [Boehm, 1998].

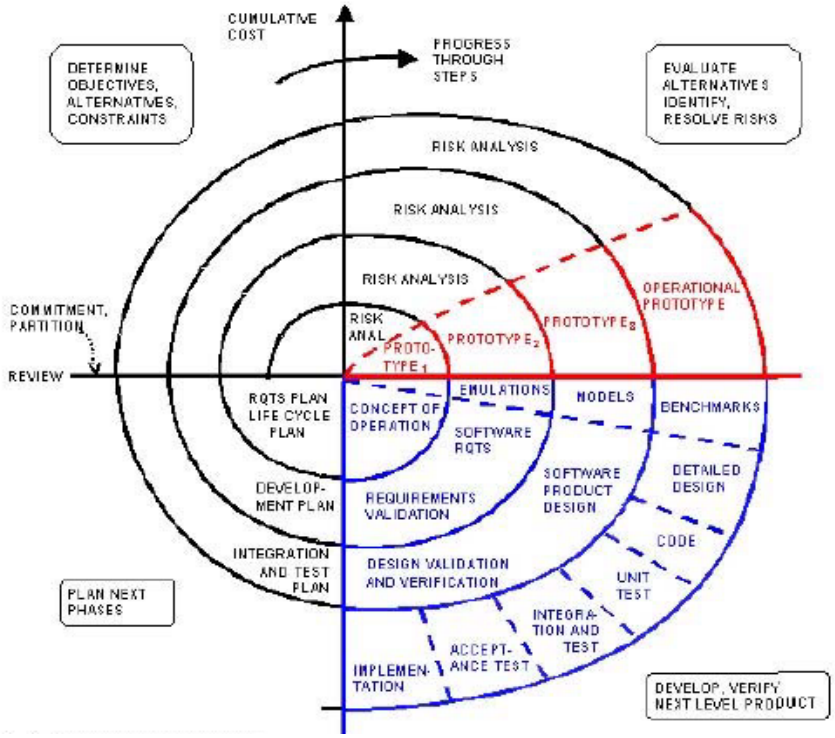


Figure 5 Boehm's Spiral Lifecycle Model [Boehm, 2001]

More recent variants of the software lifecycle seem to underemphasize or ignore maintenance completely, most notably (and disturbingly) from sources specializing in software project management. The Sequential Product Lifecycle model [Conger] has nine levels, including maintenance and retirement as separate distinct phases outside of the five-stage subsection labeled Software Development Lifecycle. The software development part of the model uses Analysis, Conceptual Design, Design, Implementation, Test, and Installation and Checkout, sidelining maintenance to an afterthought (but identifying two distinct design phases).

Similarly the System Development Lifecycle [McLeod and Smith] project management cycle lists 17 activities or levels. Like Boehm's reentrant waterfall, the stages of this model emphasize re-execution of the definition, design, and execution phases and do not include maintenance as a named activity. Another software development lifecycle model [Goldsmith] (figure 6), resembles Royce' original model, but lists six separate levels, none of which is maintenance. Instead, the Goldsmith lifecycle stops at delivery.

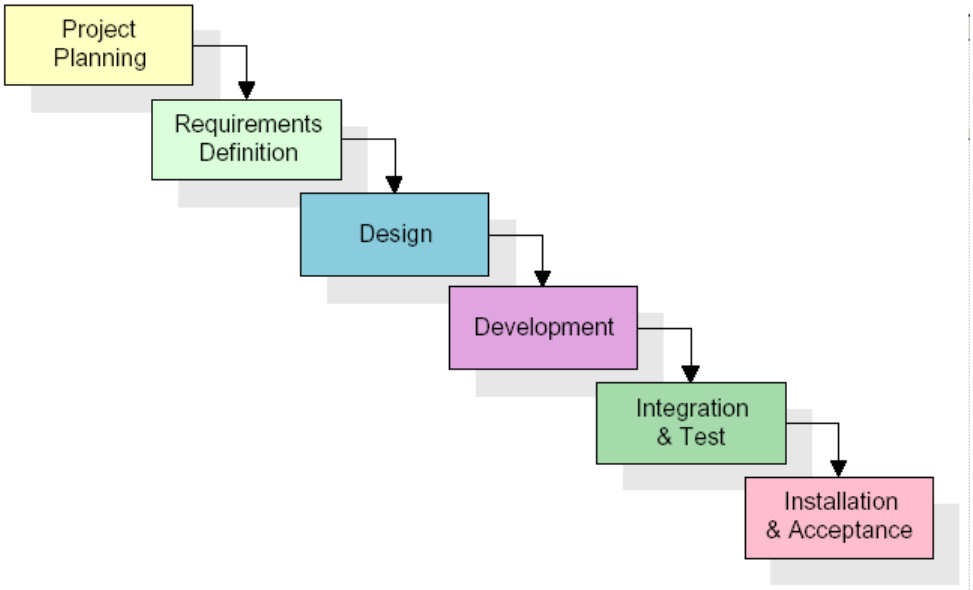


Figure 6 The Software Development Lifecycle [Goldsmith]

Software system maintenance is often ignored or marginalized, as it was in ASP, perhaps because the term is so unexciting, but more likely because the business model for development and delivery of software has changed. Maintenance is a complex, expensive service [Schneiderwind, 1987] which is sometimes difficult for a business manager to justify without close analysis. The concept of continuous development of a software product, perhaps attractive because developers can maintain a source of developer income, has replaced that of software maintenance – despite the fact that the activities are nearly identical. The rare software systems that apparently work well upon delivery are optimized before operations take place, or maintenance is performed "invisibly" concurrently with operations – so efficient that most people intimate with the software are unaware of any activity. Real-time, life-critical systems require extensive development, test, and designed-in fault tolerance [Boehm, 1981 and Ramamoorthy, 1996].

If the Conger, Goldsmith, and McLeod/Smith examples are typical of lifecycle models used to train project managers, it is unsurprising that system architects ignore maintenance in the early business development of a project. Maintenance is difficult on a product when the original product process is unknown (untraceable), change is not properly documented, changes are unstable, changes cause unpredictable "ripple" effects, and because of the "*Myopic view that maintenance is strictly a post-delivery activity*" [Schneiderwind, 1987].

At first glance, Raytheon's IPDS appears to be a standard, waterfall-like lifecycle model based on Royce. Upon closer examination, IPDS expands the model to encompass iterative development stages (figure 7).

Raytheon intends IPDS to be a *system* lifecycle template, applicable to any company product. The focus of IPDS is on the company's perspective of the whole product, incorporating business with engineering goals. Operations and support play an important role in the IPDS model, but the IPDS view of reiterative development of the product (*product evolution*) is a separate system than product supportability, sustainability, and repair.

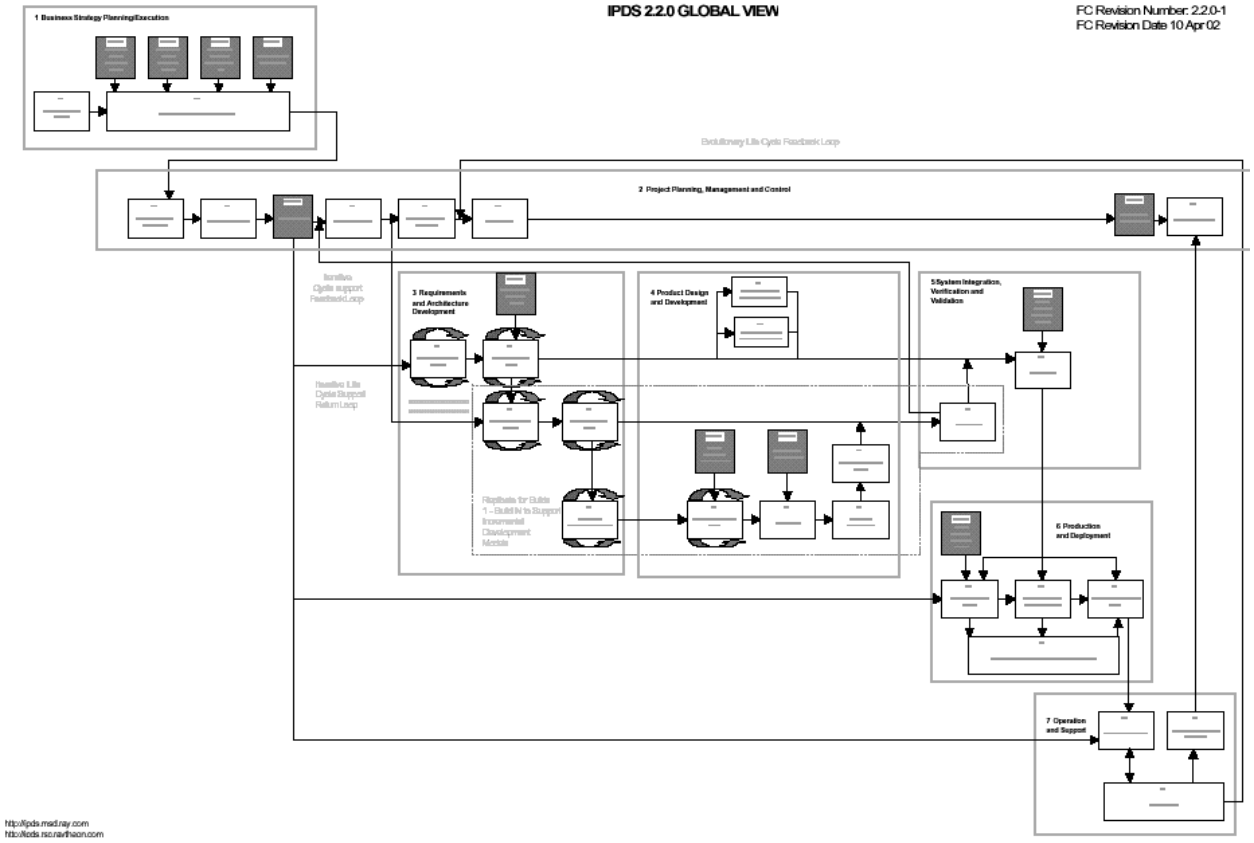


Figure 7 IPDS Architecture, Level 2

ASP would not necessarily have benefited from the use of IPDS as a lifecycle model. Recall that the ASP program did not continue to use IPDS or any similar available project lifecycle scheme after requirements changes forced a system redesign and schedule changes. In ASP, with system support very much driven by the prime contractor and the customer, planning and process were considered secondary to (rather than integral with) demonstrable delivery performance and development costs. McConnell [1996] identifies these attitudes in several of his enumerated "classic (software development) mistakes." Yet even IPDS does not emphasize software systems maintenance. While IPDS stage 7 specifies a maintenance and operations phase, the preceding phases give little support to software systems reliability, sustainability, and maintainability. Instead, the software design portion of the IPDS lifecycle addresses software lifecycle issues, and like other lifecycle models, leaves the software operating

environment to other disciplines. IPDS hardware systems include a reliability, maintainability, sustainability (RMSS) module and engineering discipline separate from software. RMSS engineering efforts concentrate on supply chain and spares manufacture and acquisition. ASP, and other company products that deliver integrated hardware and software, must evolve completely customized system maintenance processes without integrated guidance from IPDS.

Chapter 5
Need for Maintenance

The problem with the waterfall lifecycle models is that the definition of maintenance (where it is included at all) is too simple, although its inclusion with operations (a' la Royce) is significant. As pointed out by Schneiderwind [1987] and Boehm [1981] the traditional waterfall model does not show that the "Maintenance and Operations" block is really the beginning of the next iteration of the cycle of development. Even more significant is the exclusion of the overall software-operating environment from the maintenance and operations schema. Software operates on hardware and is operated by people to perform an *essential service* for an entity (individual or group), and the system design is incomplete outside the system context.

Boehm [1981] defines *software* maintenance as "*the process of modifying existing operational software while leaving its primary functions intact.*" He gives examples of the difference between maintenance and development/data update activities:

Table 1 Post Delivery Software Activities

Maintenance	Not Maintenance
Redesign and redevelopment of small portions of an existing software product	Major redesign and redevelopment (more than 50 % new code) of a new software product performing substantially the same functions
Design and development of small interfacing software packages which require some redesign of the existing product	Design and development of a sizable (more than 20% of the source instructions comprising the existing product) interfacing software package which requires relatively little redesign of the existing product
Modification of the software product's code, documentation, or data base structure	Data processing system operations, data entry, and modification of values in the database

An interpretation of Boehm's classification of maintenance vs. non-maintenance software activities

Further, Boehm classifies maintenance as software update or software repair, the first requiring change to specifications. Per Boehm, repair is corrective maintenance of failures, adaptive maintenance to changes in the processing or data environment, or perfective maintenance for enhancing performance or maintainability. However, the viewpoint of most software development organizations is necessarily software-centric. By deliberately focusing on software, most software lifecycle models avoid addressing software as a dependent part of a computing system. Observation shows that software operation depends on hardware, the user, dynamic configuration, and often, other software. One can only view software quality in the context of the entire software system.

With these classifications and through simple observation, we can make these general statements about maintenance of the software system:

- Systems require re-configuration.
- Storage devices require backup, replacement, and purging.
- Software sometimes requires “tweaking”-- tuning to work efficiently within the system.
- Failed systems require recovery.

The maintenance of any engineering product has four core purposes:

- Extends the useful lifetime of the product
- Prevents failure through improvements to system components
- Restores a product to usefulness after a failure
- Improves performance through modification, adaptation, or reconfiguration

Tuning, or minor changes to individual subsystems may improve performance. Examples of improvement through adaptation may include the adoption of higher-performance hardware or change to a newer supporting technology (e.g. IP version 6 over version 4) through abstraction in the software design implementation. Reconfiguration of existing components and subsystems (e.g. network change, reassignment of distributed processing nodes, increased memory, or task prioritization) may also improve system performance.

> *“Successful life-cycle software engineering requires continuing resolution of a variety of important but conflicting goals”*

On may apply Boehm’s [1981] comment on the software lifecycle to maintenance as a microcosm or subset of the entire development system. If viewed strictly as a post-delivery activity, maintenance is continued development of the product (figure 8). The business model derived from this viewpoint appears different from the lifecycle and model of other kinds of engineering and manufacturing products. However, there are many parallels between software systems maintenance and the post delivery lifecycle models of other engineering products.

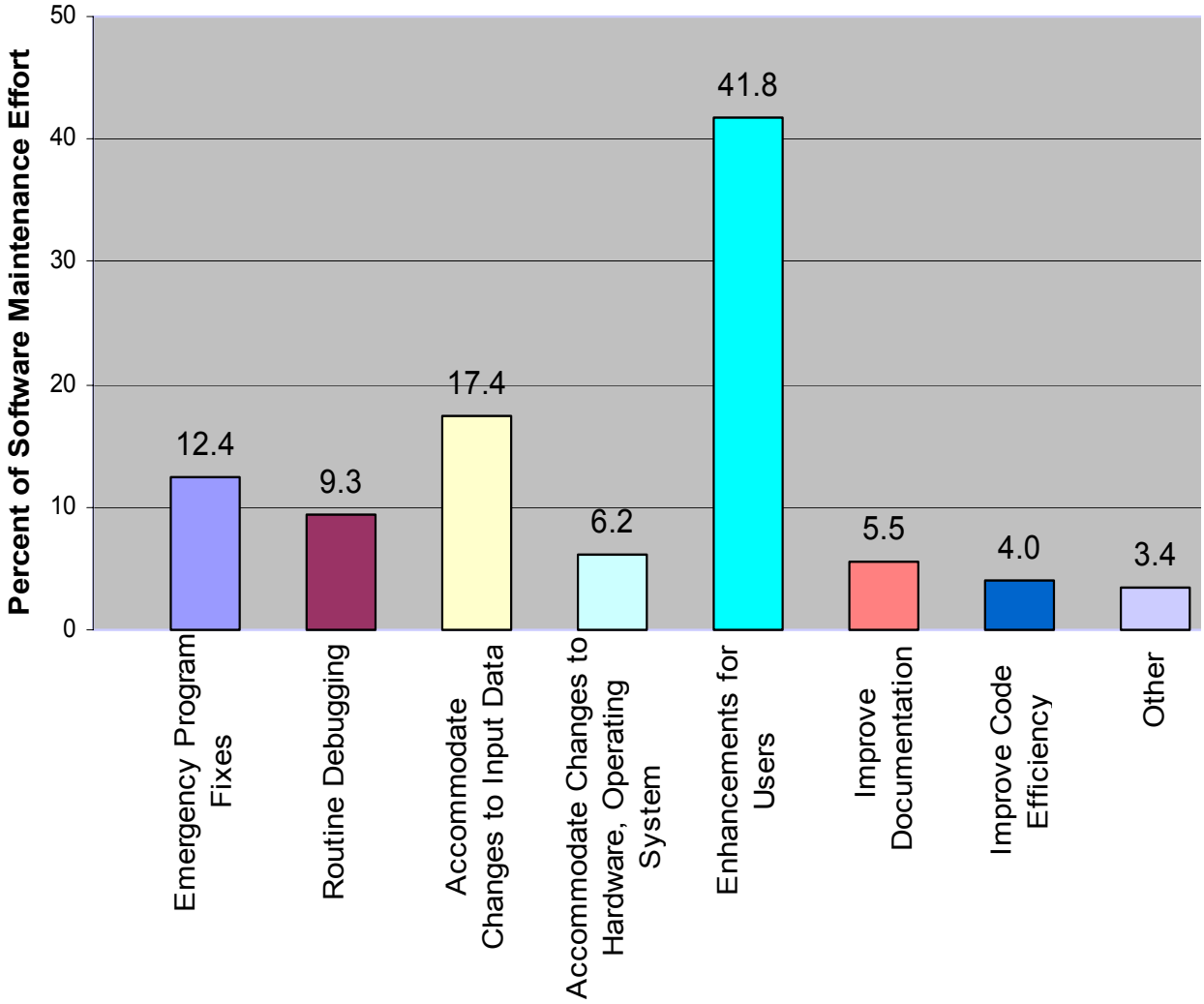


Figure 8 Distribution of Software Maintenance Effort
Data from [Boehm, 1981]

5.1 Maintenance Costs

The decision to fix or replace the application is subject to managerial analysis and in the same manner as the original decision to purchase the software. The time, effort, and assets (in the form of critical data and lack of access to the *essential service*) lost due to ignoring the problem should be weighed against the cost of maintenance verses the cost of replacement of all or part of the system.

Custom or low-volume software technical applications are expensive to develop and to deploy, just as are specialized products of any other engineering discipline. Having purchased such an application and created the supporting system, a system user would be justifiably reluctant to replace the entire system at once. The initial cost of new applications dictates a policy of continuous and incremental maintenance through upgrades and changes.

Contrary wise, developers sometimes market new applications as cheap alternatives to buggy legacy systems. However, after a complete analysis, the new implementation may require systemic changes that make them infeasible: user training, hardware and network infrastructure improvements, data conversion, and planned obsolescence may make the change too expensive.

Maintenance is a large part (more than 50% of the software cost [Boehm, 1981]) of the total cost of ownership of the product, and possibly the cause of much of the cost overruns for users and developers. Unplanned, post-delivery maintenance cause increased cost to developers while the system is still under warranty or maintenance contract. Unexpected maintenance costs, through downtime and reduction or elimination of the essential data service, affect customer's overall cost-of-ownership and the customer's opinion of system. Users who must have changes made to legacy software systems are likely to encounter high re-engineering costs. A new commercial product, developed to compete with an existing product, must take into consideration the functionality, operating environment, and market of the competing application. Development of the next level of functionality or performance, even if that level is attained through complete replacement of the software product, the replacement can be classified as a form of system maintenance to continue the *essential service*.

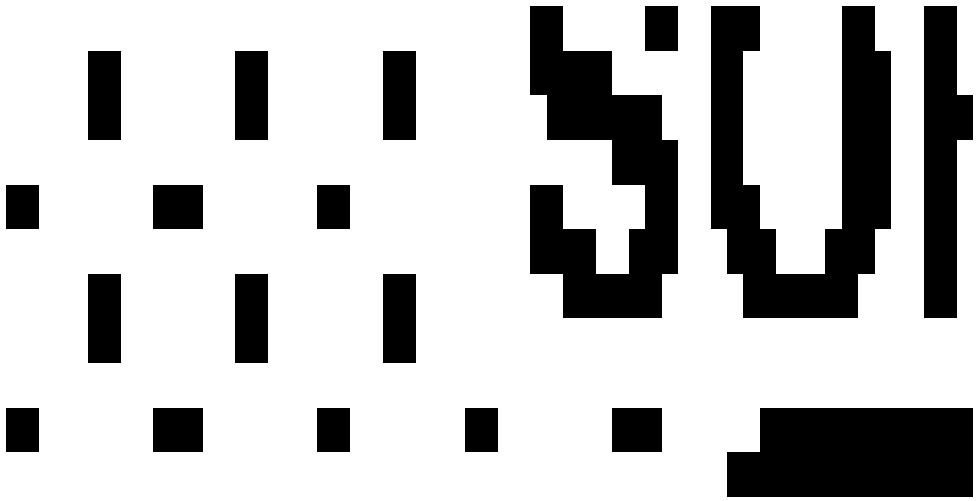


Figure 9 Hardware/Software Cost Trends
[Boehm, 1981], reproduced from [Yue]

Boehm's data (fig 8) show that only one eighth of maintenance activities are dedicated to emergency fixes, and the figure is less than 25 percent when emergency fixes are combined with routine debugging (interpreted as diagnosis and troubleshooting activities that do not directly deliver a fix). Maintainers dedicate the remaining portion of maintenance actions (over 75%) to activities driven by post delivery requirements changes. Other data from Boehm [1981] show that requirements change approximately 25% over the development portion of the lifecycle, and that software maintenance is responsible for 50% of the cost of the software product.

The ability to predict the inevitability of product change should drive the software design, and therefore the design of the software support system. McConnell [1996] identifies *Designing for Change* as a software "best practice." Essential to the 5-part system are "identifying areas likely to change" and "develop a change." As a side benefit, McConnell cites examples that show that programs designed for change "*continue to yield benefits long after their initial construction."* Quoting a 1998 study by Capers Jones, McConnell points out that failure to plan for and manage change reduces a product service life and makes the product *"catastrophically expensive to maintain within 3 to 5 years."* Returning to Boehm's statistics, we can conclude that approximately 40% of the cost of software product change occurs in the post-delivery phase of the project.

5.2 Maintenance Finance

Royce's waterfall concept of software maintenance originated from the need for maintenance of other systems, and is complete as far as it goes. The accepted concept of the lifecycle of a standard manufactured product

is a cycle of defining the need, selecting a product, purchase and delivery, maintenance and operation of the product, and retirement or replacement. Business uses the principle of product and inventory depreciation, amortization, and product life to manage business costs. Software, as a business investment, seems to defy this expense model.

Modern, accepted accounting practices dictate that depreciation is applied not only the initial cost of a business asset, but to the setup and capital expense costs of the product over its life span [Finkler, 1992]. If accountants do not expense maintenance of a software product as a capital cost, but instead as an operating expense, the cost of maintenance buried in user operating expenses. The recursive waterfall or Boehm spiral model allows one to view and manage each major software development as a separate item, even though the new software may be an advanced development of the same application. This may be attractive from an accounting perspective, allowing operating costs to reduce the tax expense, but the practice may hide the real cost of ownership of the software product. Software's apparent differences from other products make this cost model possible.

Software differs from most other engineered products in many ways. Consider that software does not wear out – the antithesis of depreciation. The Y2K panic showed that there is much legacy software supplying *essential service*s throughout the world. Legacy hardware systems, still operating economically to perform a data processing service, support legacy applications. System maintainers have "ported" legacy applications to new hardware platforms and/or have implemented applications within a "shell" to support new features. Supposedly antiquated MS-DOS, CPM, IBM, and UNIX applications are still supported by many desktop systems or are used on hardware that is much more capable than the original system – the application that would otherwise become too slow is "saved" by Moore's law, increasing system throughput by adaptation to newer hardware. By virtue of market strategy, Microsoft OS products are somewhat backwards compatible [Viewz, 2002], allowing users to gain new capabilities without replacing the entire system – hopefully without suspending the *essential service* supplied by the system.

Software maintenance costs are predictable using cost estimation techniques like the Boehm COCOMO [Boehm, 1981] model. Boehm also indicates a relationship between software reliability designed into the product and reduced maintenance cost over the lifecycle of the product. The customer expectation of software serviceability over the life span must be a consideration when selecting a lifecycle model. Detailed software system design must implement an efficient software maintenance system within the lifecycle model for maximum return.

5.3 Maintenance Modes

5.3.1 Failure Modes

The hardware world uses an obvious definition for the term failure – a hardware product fails when it ceases to function. Users of systems consider even a partial loss of overall functionality is a complex mechanical system a failure, and continued operation with limited functionality is a "failure mode." Software users rarely consider application functional deterioration a failure, once the software is past the "infant mortality" phase of early deployment, integration, and test. To describe the system repair mode of software, one must agree upon a definition of a software system failure.

The software system may become completely unfit for use through catastrophic failure to operate or loss of critical data. The system fails to perform the *essential service* for the user. From the system viewpoint, any component of the system may cause a complete failure, not only the software. While system failures may indicate the necessity of replacement of the software, more often these failures indicate that the system requires maintenance short of total overhaul.

Less severe than complete failure, the software system may simply be difficult to use. Developers may design or implement the user interface may be poorly (although this should affect the original purchase decision) or it may become insufficient when actual usage diverges from the initial system model and the designer's concept of user operations. Storage, hardware, configuration, or network problems can also cause this level of failure.

Trewn and Yang [2000] define reliability in terms of functionality:

"Functional reliability is defined as the likelihood of successfully providing necessary functions that a system or component is intended to deliver. The concept of failure is extended to include:

- *hard failure (complete failure of function)*
- *soft failure (performance degradation in delivering functions)*

Outside of hardware failures, software may enter an untested condition, causing a catastrophic failure. In this case, the software developer failed to anticipate the condition that actual operation has exposed – a failure which may be excusable. McConnell [1996] calls software the most complex system implemented by human effort. Software's complexity makes it difficult to test. Software testing processes, even in mature organizations, sometimes remain underdeveloped [Alamares, 2001]. Testing prior to delivery is costly in both time and effort, and cannot possibly discover and correct all possible failure modes. Developer simulation and modeling can only

attempt to approximate actual use and the best and most complete testing of any product occurs during actual use. Prior to product release, Auto, aircraft, machinery, and other manufacturers subject their products to a variety of normal use environments, extreme usage testing, and often test the product to destruction so that the final released product is the best possible within the limitations of the production process. The manufacturer's philosophy of extensive testing rarely applies to software development.

Software system failures are more likely to occur in actual usage than in pre-delivery and system integration testing. Software system users while more likely to encounter an error, are necessarily less well equipped than the system developers to identify, diagnose, and correct system failures. Failure in the post delivery stage of use dictates the need for a maintenance infrastructure to support post-delivery repairs.

5.3.2 **Amelioration**

System repairs take many forms, and are obviously dependent on the nature and severity of the system failure. In the case of either a hard or soft failure, the process of repair is some combination of:

- Restoration of the system to pre-failure level of operation (restart or re-boot)
- Determination of the state of the system at the time of the failure (to aid in recovery, to gather data for diagnosis)
- Isolation of the cause of the failure (diagnosis)
- Correction of the faulty component (Code change, repair)
- Avoidance of the failure circumstances (Data checking, operator actions)

These steps may all take place in a complete repair of the system, although not necessarily in the order presented. Occasionally the repair is not complete, for example, an incomplete diagnosis or a pending delivery of a software repair may mean that the system goes through repeated restoration.

Failure recovery is often the simplest form of maintenance and for the user is a matter of re-starting part or all of the system and the attempted recovery of lost data. In other cases, immediate restoration of the system may wait until correction of the faulty software or hardware component. As with other stages of repair, the severity of the system failure determines the type of restoration required: Consider that most PC users know how to re-boot Microsoft operating systems through the "control + alt + delete" key sequence as a last "soft" resort in response to a non-responsive application. Software and hardware reboots are also common in restoration of other operating systems. At another extreme, hardware storage failure can cause the application itself to be corrupted, or can cause

some form of data loss. The storage media or associated hardware might require replacement, and the application may require re-installation.

In a real software failure, the software itself requires troubleshooting and repair. A software repair may be any combination of software source code modification, system reconfiguration, or an adaptation of user behavior.

Source code modifications seek to isolate and prevent or compensate for the specific conditions that caused the failure. Complete diagnostics of the failure condition is necessary for a successful modification. Code modification also usually requires recompilation and delivery of new executable programs or modules, and has direct implications in software source code control and documentation. Software code changes are themselves subject to error and create a high risk of introducing new errors into the system, either through software. System errors sometimes occur through the incorrect application of a software code change, requiring that the change delivery process also be subject to error checking and oversight.

System reconfigurations may take many take many forms. The maintainer may prescribe hardware changes, such as the addition of memory, network connection modules, or the replacement of components, again based on the diagnosis of the failure. Configuration changes may also be "soft," in that startup scripts, configuration files, command options, or system settings are changed. Because system pre-delivery testing is necessarily limited in scope, failures that require configuration changes are even harder to predict than code errors prior to implementation.

The system change most often created by a software failure is adaptation of user behavior. Like the old vaudeville routine, the first reaction to a failure is often "don't do that" -- to avoid repetition of the actions that brought on the failure in the first place. If the action is routine and required to provide the *essential service*, the user may have to change the sequence of actions performed.

In an un-maintained system, the users will continue to encounter an error rather to give up even limited availability of the *essential service*. Behavior change is often the only recourse for users of legacy or unsupported software systems. In this case, user behavior changes in avoidance of the circumstances of the failure may be the only cure. While it is technically unsatisfactory, user behavior change may be the only solution to the system failure. A well-managed system should use an established process to document and distribute the change to all users. Legacy software, such as the applications at the center of the Y2K panic, may be more prone to this type of failure and repair.

Software repairs may also require a complex delivery system. At the simplest extreme, home computer users may find simple instructions on-line or in automated help applications for reconfiguring software (or instructions on changes to user behavior). Open source system users post questions and problems to centralized databases on the internet. In response to user problems, open source developers use the internet to post changes to a central repository. Users download changes via FTP, and sometimes recompile the change on the local hardware system. Updates may be automatic: Netscape, the America On Line (AOL) user desktop, Adobe Acrobat, and Microsoft applications and operating systems will often automatically download updates, patches, and changes through the internet. In addition to internet-based downloads, many commercial developers offer software repair delivery using compact disk or diskette. ASP uses FTP over private network to deliver changes through an intermediary.

CORBA, a methodology and strategy for the use of distributed modules across a network, allows the program that uses a distributed object to specify the object by the type of service needed. Maintainers create object system software updates through changes to individual modules. The requesting program receives services specified in the request, from the most current module in the network.

ASP and automated delivery systems use additional software for installation of new software. Boehm refers to the management of software using other software, or software used to support the main user application as "scaffolding" [Boehm, 1981]. Scaffolding software is also a potentially complex software subsystem, and has its own lifecycle. Developers should create scaffolding and support software in parallel with the object application, and control it using an SCM system. Delivery software and processes, along with other system scaffolding, make up one component of a software system infrastructure.

Chapter 6
Comparison Technologies

Other complex engineering artifacts require maintenance. The original engineers were those who could create and operate complex systems and keep them from breaking down. Original engineers built roads, harbors, buildings, irrigation, and other systems to support agriculture and commerce – and these systems have all required periodic maintenance. Maintenance engineering has become an important subset of many engineering disciplines. Shipboard steam engineers appeared after Fulton's adaptation of Watt's invention to boats and became a second, specialized, technical, group with a separate chain of command along side the ship's command officers (who specialized in command, commerce, and navigation). Steam engineering, when applied to the maintenance and efficient operation of railroad systems (and combined with early information technology in the form of the telegraph) initiated the definition and study of management of technical services for profit.

Maintenance and sustainability processes, terminology, and thought originated with traditional engineering products. While there are many differences between "hardware" and software products, software system engineers may be able to exploit some similarities in a software system infrastructure design. In exploring the structure of a support system, comparison to easily recognized, common processes may be useful.

The transportation industry offers two such common examples of maintenance and service infrastructures that have developed or evolved to support complex systems. Aircraft and heavy-load trucks provide an *essential service* directly to the owning organization, and indirectly to the customers who those organizations service in turn. The two subsets of the transportation industry are also consumers of maintenance services, and benefit from an extensive, integrated service infrastructure.

6.1 Trucking

Fleet maintenance for autos and trucks require complex, structured support systems. Over-the-road drivers deliver goods throughout North America, logging hundreds of thousands of miles per year. Operators may own their own vehicles, or a transportation company may own a large fleet of vehicles. Large companies operate depots, located close to highway hubs, for loading and distribution of truck cargo.

Large diesel trucks require relatively smooth roads to operate. In the U. S. the Interstate system provides high-volume access between major cities. State and local roads provide access from the interstate highways from

producers and suppliers to the customers. States and the federal government share financial responsibility for repair of existing highways and the establishment of new routes. Highway infrastructure is a large part of the state and federal budgets, and is necessarily under legislative oversight. Local and state governments obtain some revenue for road maintenance through taxation of the trucking industry and on motor fuel.

Large truck operators require special licensing. Operators must complete approved training, usually with a nominal amount of on-the-road experience, and pass written and/or practical examination, specific requirements administered and set by state agencies (that are also subject to legislative review). Operators keep logs of hours driven and the composition of the load.

Truck systems are subject to frequent inspections by the operator and by regulatory agencies. Road safety and equipment condition, loaded weight, and driver logs are sometimes inspected at state police "weigh stations" located on major highways, usually just inside state borders. The state that licenses the tractor or trailer requires periodic (usually annual) inspections of the vehicle.

Tractor-trailer trucks are an inherently modular system, in that a tractor may connect to one or more trailers, but each part is separately maintained. Trailers may be general-purpose, or specialized for the size and shape of a specific type of load. Trailers are subject to a separate maintenance schedule, and a subset of the transportation maintenance infrastructure is dedicated to just the trailer. Governments license heavy load trailers separately from the tractors used to pull them.

Diesel truck tractors and trailers require fuel, lubricants, and replaceable single-use parts such as filters, seals, and tires. Diesels operate relatively efficiently at idle and complex refrigeration and other support systems powered by the engine must operate continuously, so diesel engines often operate for extended periods. Diesel engine use is measured in hours of operation rather than distance traveled. Unlike automobiles, where large parts usually last the life span of the vehicle, operators may replace large diesel truck subsystems such as engine, transmission, and differentials with newer parts. Often, replacement parts come from different manufacturers than the original supplier. Large parts come in standard sizes and use standard measurements for performance comparison. Many truck manufacturers create cab/chassis combinations and assemble parts customized to a specific usage or to customer specifications.

Maintenance of trucks has become specialized. Periodic replacement of lubricants and fuel is readily available. A subset of maintenance specialists service and replace difficult-to-handle tires and wheels, sometimes

concurrently with other services. Engine, transmission, and axles, optimized for fuel efficiency and longevity, require specialized skills for service. Specialists often maintain only trailers, although as with most tractors, non-specialized mechanics perform most trailer maintenance. A few specialists using special equipment handle major overhaul of trucks, including bodywork, replacement of major parts, and upgrades. The manufacturer often provides rebuild and overhaul services.

Operators perform most first line maintenance, determining the need by scheduled maintenance interval, through inspection, or through subsystem failure. Mechanics with general skills fulfill most simple maintenance needs, and specialized, highly trained mechanics perform more complex system repairs and overhauls. Major overhauls and upgrades to systems require a combination of highly skilled mechanics, specialized support equipment, and engineering support often provided by the manufacturer of the entire vehicle or the system component.

Truck manufacturers frequently provide operator manuals, maintenance manuals, and specialized maintenance training. Commercial or manufacturer-supported schools certify skilled mechanics. Manufacturers compile records of component and system testing prior to delivery, and use these with field-test results to develop and update maintenance schedules, operation instructions, technical bulletins, and in some cases manufacturer recalls. Manufacturers also use historical records to support and develop modifications of parts and components for increased service life and/or for reduced cost for both fielded models and for development of new product lines.

6.2 Aviation

Since the Wright brothers, the military, private enthusiasts, and commercial aviation have developed a highly specialized infrastructure for the safe operation of aircraft. Some commercial and military aircraft have been flying for over 50 years depending on high quality repairs, preventative parts replacement, and major subsystem upgrade. Small, regional airlines use older transport aircraft to transport passengers and freight. The military makes extensive use of older transport and combat aircraft, most notably, the U. S. Air Force still uses the B-52 bomber and the KC-135 (based on the Boeing 707) air tanker, all of which were originally manufactured in the 1950s and 60s. Private owners operate restored and meticulously maintained aircraft dating back to World War I. The U.S. forest service uses retired and surplus military and civilian aircraft for reconnaissance, fire suppressant delivery, and transport to detect and fight forest fires.

Like those of automobiles and trucks, aircraft manufacturers develop maintenance manuals and service guidelines in conjunction with development and deployment of their systems. Operations managers, planners, servicing agencies, and mechanics use maintenance information to keep the product in operation for the maximum time. Where the original manufacturer is no longer in business or maintenance information is not available, private individuals and third-party service providers develop replacement parts, maintenance support information,

When sub-components and sub-systems have worn out or become outdated, operators may upgrade systems such as power plant, navigation, and communications using modern replacements. Mechanics replace other components with new or rebuilt parts.

Training for and documentation of aircraft maintenance are extensive. In the United States, the Federal Aviation Administration (FAA) certifies pilots, technical flight crew, and mechanics. Each maintainer and operator requires extensive training from a certified facility and/or certified instructor. Most maintenance work performed by qualified technicians requires inspection by another qualified (and usually more experienced) and licensed technician. Maintainers and operators keep extensive and detailed logs of the work done to and hours flown by each aircraft. Operators and maintainers also keep work logs. Logbooks are "backed up" or duplicated at the maintenance facilities, while the main copy stays with the subject. Logs are subject to inspection, audit, review, and a controlled change and correction process.

Manufacturers like Boeing and Raytheon keep provide engineering services for the deployed aircraft fleets – by regulation, by contract and warranty, and through a commitment to customer satisfaction. Detailed records are duplicated at the engineering facility or manufacturing plant. User problem reports are subject to trends analysis, and the results influence the manufacture, design, and availability of spare and replacement parts. The manufacturer tests all parts and systems extensively post delivery, and test results become part of the overall database supporting all similar aircraft sub-systems. The designers and developers of new products and manufacturing processes improve their design and implementations using the results of maintenance database analyses. Analyses that show increased risk to the operator/customer are the basis for technical bulletins, increased inspection rates, detailed maintenance instructions, and as a marketing tool.

More so than tractor-trailer accidents, aircraft accidents come under heavy public scrutiny and government oversight. Aircraft owners, operators, manufacturers, and maintainers are required to cooperate with FAA accident

investigators, supplying information from recorded data. In turn, the FAA provides the results of these investigations with the motive of preventing further accidents.

Aviation infrastructure, like that of the trucking industry, requires extensive support by local, state, and federal government agencies in cooperation with private companies. Taxation supports airports and secondary transportation routes. Air routes and flight exclusion areas are coordinated with local authorities and with the FAA. Contractors to local governments and large commercial airlines provide ground transportation, servicing, and other support services. Recent congressional mandates have recently established the federalized airport security agency, and are likely to establish further regulations to enhance flight safety and security in the near future.

6.3 Differences

The transportation industry models, illustrated above, represent an evolving, comprehensive, and market driven approach to maintenance integrated with the object system. In software maintenance the equivalent planned maintenance approach often does not exist. Software users rarely own software source code, which usually remains the intellectual property of the original developer. Rather than many independent suppliers of maintenance services, the primary maintainer of commercially developed software is the developer. The open source movement provides a notable exception, in which uses a public development of software by privately donated contributions, with customization and maintenance provided by secondarily contracted or in-house providers.

While one does not have to change the oil in a software program, maintainers should have performance specifications and design documentation at hand to deliver software changes to software users effectively and efficiently. The high-level system design should abstract hardware or operating system version changes to provide a simple upgrade path for deployed systems. These principles apply to open source software maintenance, as well as developer-maintained applications.

Software does not wear out, but more often becomes economically unfit for use through advances in hardware and changes to interfacing software. New hardware capability is constantly under development, and frequently capabilities increase and are made available at lower cost, as observed by Gordon Moore. Software which does make use of increases in memory and storage capacities, access speeds, network interconnectivity, user interface capabilities, or the next new technology will replace software that fails to take advantage of new features. As previously stated [McConnell, 1996] software should be designed to adapt to new technology, for both longevity

and for low maintenance costs. Testing should model anticipated platform changes and adaptations, and capture current performance levels with current hardware, to make the evolution of the product feasible.

We may observe another difference between software systems and life-critical aircraft systems, in that the practitioners of aircraft maintenance must be officially trained and licensed. Even software professionals who work on aircraft systems do not generally require professional engineering or mechanic licensing. Quality assurance and extensive pre-release testing are prerequisite to creation of high reliability software systems. Even though some states (like Texas) have started professional licensing programs for software engineers, there is no regulated mandatory licensing of software engineers.

Although there are many more potential comparisons, we must examine a final pertinent difference here. Brooks [1987] points out that "*software systems are more complex for their size than any other human construct*." Software's complexity makes it difficult to manage efficiently, design well, or implement properly. Brooks' prediction is that software systems, running on already complex computer hardware, will not easily achieve greater efficiencies in development – even in spite of the increase in efficiency in computer hardware [Brooks, 1987]. Managing and creating software systems will be more difficult than those of the transportation examples. Success stories and case studies prove that managed software system development is possible, but software system designers must correct for the scale and complexity of other systems when considering comparisons.

6.4 Similarities

When viewed from the system perspective, the similarities between software systems maintenance and maintenance of other systems become more apparent. Once again, Moore's law guarantees hardware obsolescence, although obsolescence does not occur at the same 18-month rate. Despite software's abstract nature, software systems are tangible entities composed of software applications, supporting processors, networks, display and input devices, documentation, procedures, and operators.

Another similarity is that maintainers perform much maintenance on all the example systems for the purpose of upgrades. Aircraft require more efficient and more capable communications and navigation gear. Aircraft and over-the-road vehicles require increased power plant efficiency. By use of standardized and specified interfaces (mechanical, electrical, hydraulic, and electronic), many upgrades to a given chassis or airframe are possible over the lifetime of the system. The desire to gain the same synergies and efficiencies in software maintenance provides motivation for movement toward software components, modularity, and defined structures.

Another software system feature similarity is that hardware wears out, through increased user demands on capacity and through the simple wear caused by the heating and cooling cycle characteristic of system operation. Hardware replacement and software upgrade are not necessarily concurrent, as was illustrated in the Y2K example. One decision point for a user move to advanced hardware might be *forward compatibility* of the extant software system. Again referring to Boehm's cost ratios (fig 10), software investment can be as much as 80% of the total system cost. It makes economic sense for the user to invest in software that will remain useful when users must later change the hardware subsystem.

New capabilities for improved performance apply to all the systems. Over the last 30 years, computers user interfaces have gradually been modified from punch cards, to keyboard, to using CRT (and more recently, LED and LCD) graphical interfaces and pointing devices. In order to implement these new capabilities, initially older applications were "wrapped" in other applications. Maintainers have modified legacy applications (especially databases) to enable their use over the internet. Eventually, tool and theory advances have made building web-based, GUI enabled applications the standard.

6.5 Missing Infrastructure

In the transportation system examples cited above, a maintenance infrastructure has evolved to perform system upgrades, routine repairs, and large subsystem diagnostics and repair. These services dovetail with other commercial, government, and private support services. Independent and manufacturer-sponsored maintenance engineering and technical service providers are part of the infrastructure that allows operators to continue to provide *essential service* to their customers. Manufacturers, maintenance service providers, and system users capitalize on the use of standard components and specifications, system documentation, and service accessibility and competition.

Software systems have few of these advantages. Some manufacturers (e.g. SUN, Microsoft, and IBM) do provide service for selected products, but to encourage users to upgrade to newer software, manufacturers may not support many of the older releases. There is not yet a standard for interchangeable software components. Hardware is somewhat interchangeable, in that operating systems and applications have had to become more universally operable. Only a few independent service providers support a small amount of commercial software, so users often must return to the original manufacturer for service.

The customer for the ASP project did budget for some maintenance by the manufacturer. However, the ASP project did not contain a systematic method for receiving failure reports from the field, for staffing, or for

creating or delivering software repairs. The manufacturer provided maintenance services, but inefficiently, reducing the potential availability of the *essential service* required by the customer.

Chapter 7
System Testing and Analysis

In other engineering systems, components of subsystems are subject to testing and examination under simulated and real working conditions. Software systems undergo a similar usage cycle as other types of systems, but statistical analysis of software systems testing and failure under working conditions is under-emphasized in most applications.

In ASP system integrators limited performance measurement and test to the minimum required show the customer that the software met specifications. Test tools and records were specific to the integration environment; maintainers could not generalize the results of performance measurements to apply them to other configurations. Most test results were not available as historical data. This lack of data prevented scientific analysis of the software system performance and reliability. An appropriate system infrastructure could have provided ASP maintainers with valuable testing and system analysis tools, allowing for more efficient use of programming time and a focus on the problem areas with the most impact on system performance.

7.1 Software Test Considerations

System developers often view software testing as a separate, and usually onerous, part of software development. In fact, however, testing and test design takes place throughout the development cycle. By setting a system requirement, a systems engineer is beginning to define the test by which the implementation will demonstrate a functional capability. In the design phase, the engineer defines a method by which the final system will comply with the requirement, and begins to limit the potential ways in which the software can function. In the development phase, the software engineer will make specific choices that implement the design, and will try out portions of the code to ensure that it meets low level, defined requirements at a modular level. Post delivery, users continue to test the system until its final retirement when the software can no longer perform economically.

Pre-delivery verification and validation testing has two main purposes: demonstration of functional fitness for use and the detection and elimination of errors. Determination of fitness for use begins with the initial requirements and design, and continues through the maintenance phase until the system is retired or replaced. The development team may anticipate errors as an inherent risk in the system's overall environment or because of design choices. To the extent that it models actual usage, testing can detect and eliminate anticipated errors before delivery

of the product. Only actual usage of the system will allow maintainers and users to detect a varying quantity and the severity of unanticipated errors. Maintainers must address these errors after delivery -- in the maintenance phase of the software lifecycle.

Test development requires analysis of the system requirements at every level. At the highest level, the issue is fitness for use as defined by the customer requirements and the standards of the developer. In an ideal case, system engineers write the test in abstract form simultaneously with the process that determines initial customer requirements. As system engineers decompose high-level requirements into more detailed low-level requirements, each new level of functionality implies and requires a new level of testing.

Test development also requires analysis of the test environment – the software system. If it is not possible to test the project using the "real" environment of the target system, then developers must design and implement simulations and hardware test bed platforms with the software. This creates many levels of complexity, since developers must also design and implement simulators, emulators, test networks, and test hardware that create a reasonable facsimile of the real, final system. If the test environment does not resemble the real world faithfully, any testing conducted will have limited usefulness at best. Tests, test environment, and test systems can generate implicit requirements and assumptions that directly affect the target system. Tests conducted on target hardware, in the target environment, with real data, may still not simulate the final system loading, nor properly replicate the real user environment. Testers must make a proper analysis of all the factors in the final use of the product to demonstrate all the functional limitations and capabilities of the system fully.

A limitation on testability is engendered by analysis of the total system, and is implicit in the design requirements, for instance, real time software may not be executed using a real time OS, but on a time-sharing, desktop system. The hardware, the number of tasks executing, user privileges, and system configuration at the time of the test will influence system performance. The accuracy of the system clock and the system's disk read/write rates will determine how well the software works. Post-test analysis should identify differences in the test performance over the anticipated performance in a real-time environment.

Testing adds effort, time, and engineering cost to the software project. Even if a developer deploys a system with minimal testing, and maintainers test and change the system in actual use, the software system generates high costs in user down time, lost data, and potentially in reduction of the developer's reputation. Testers must minimize the test effort to reduce developer and user costs, but must maximize testing to eliminate critical errors.

The decision to conduct a given type or level of test must be balanced over the need to deliver software without the type of error that the test is designed to detect and eliminate, or the need to demonstrate functionality.

In the end, the final software system test is real use with real data by real users. No amount of pre-delivery modeling and emulation can predict actual usage in all conditions over the lifetime of the product. Because the ability to model the real world is limited, extensive testing has a diminishing contribution to the final value of the system. Software that is critical to prevent or is likely to cause damage, injury, or loss of life (e.g. medical systems, aircraft controls, the space shuttle flight control system [Ramamoorthy, 1996]) must undergo more rigorous testing than systems with a less important purpose.

After design validation, further testing requires two additional phases: development-level testing for errors and error handling, and system level test to characterize performance.

Development testing takes place during all phases of coding and compiling the software, in conjunction and concurrent with writing the program. The developer looks for errors by executing application, in part or as a whole, with both correct and deliberately out-of-specification input data. A coder must closely examine and further refine the detailed design of the project to deliver the final functionality. The system designer may include some error handling and data validation into the system at the start; maintainers may add other anticipatory and recovery functions because of post-test analysis.

Developers often add "instrumentation" to the code under test. The simplest form of instrumentation is the "#ifdef debug" C-language preprocessor instruction in conjunction with data logging using a "printf" statement. Almost all instrumentation influences system performance, affecting processor cycles, memory footprint, and system IO. System logging may affect the inherent timing requirement in real-time systems. Instrumentation is useful in the development phase of testing, but like internal test drivers, developers must remove or disable logging before system characterization and delivery.

At the development level, the programmer checks for proper code execution using data that fits the parameters of the "real-world" data, as defined in requirement analysis and design. For complex programs, the programmer will develop test stubs and drivers to run these tests. The programmer will also determine robustness in the program by generating out-of-tolerance input data, and creating error handling and data validation systems to compensate, anticipate, and recover from data corruption. Testers may use special applications to create both "good" and "bad" data to be stored in memory buffers or files.

7.2 System-Level Testing

System testing checks the product fitness-for-use and for high-level functionality. In the latter stages of development, a system approaches a level where it is almost ready for delivery. All coding is finished and the software includes most design points before system level test commences. System level testing assumes that the development level error handling and data validation checks are operational, and that a fully integrated system is practical.

While new errors and repairs may still occur during system testing, the purpose of system level testing is to determine the performance of the system as a whole. In a real-time system, characterization includes evaluation of requirements for timing and for robustness under normal and extreme usage, but not necessarily error handling.

Whether or not one considers a system as real-time, a data processing application will have a real or implicit throughput requirement. Testing should characterize the system at the "normal" and extreme levels of real use, as determined by analysis. Depending on required reliability, the system response in data environments beyond the performance specification may also be tested. While "torture testing" is of limited use in software development, but can be useful for hardware performance comparison.

Changing test system operating parameters creates changes to system performance. In a time-sharing system, system administrators can change the operating system user privileges or task priorities. If the operating system allows, the test engineer can change the overall system load, data logging, read-write buffering, or other factors to compare throughput and response times under controlled, varying conditions. If applicable, test designers could also change hardware platforms, data storage devices, I/O bandwidth, and network connectivity to derive a source of comparison. Analyzing the test data generated from these variations will create a basis for predicting the performance on various target system configurations.

Besides performance characterization, system level testing begins to answer the original requirements and high-level specifications. System level testing must begin to establish the product fitness-for-use as defined by the customer. The tester must demonstrate system functionality, often for the customer, before delivery. The degree to which the system actually works is the final validation of the design and the developers understanding of the customer's requirements.

As stated earlier, the system test-bed may not resemble the final application -- hardware, OS version, networking, and other factors may influence the actual field implementation. Modeling and characterization tests performed at system level may also predict the effects of configuration changes on the implementation.

7.3 Performance Measurement Tools

Every software engineer on a project develops test tools for verifying the functionality of his or her portion of the code. Stubs, drivers, and test data are useful in all phases of development for comparison and debugging. Often, these tools are not available to system integrators, who must devise their own methods of testing the code to the same requirement. Maintainers must repeat the tool development process if the tools are not available.

Maintainers may use test software for post-delivery for diagnostics during the maintenance phase of the software lifecycle to determine the appropriateness and fitness of software upgrades and repairs. Test software should be maintained current with the design and implementation of the target application, keeping it compatible with the delivered system code.

The long-term use of test applications may require the use of common source code modules and the delivery of test software and stored test data into an SCM system. An SCM infrastructure that includes all system software should allow for the maintenance of the software tools for test and performance measurement. Boehm [1981] refers to these tools as system “scaffolding.”

In development, an SCM infrastructure should support testing of code to quantify the degree to which the code meets the customer's requirements. In maintenance, the SCM infrastructure should supply the same functionality, but with an emphasis on user requirements and on minimizing rework. Manufacturers apply the term rework to any non-production-oriented work on the product to fix a perceived quality shortcoming. Repeating tests unnecessarily, excessive down or re-cycle time because of bugs, repeating development efforts to establish performance parameters, or unresolved functionality requirements are typical examples of software rework.

In the space shuttle case study [Ramamoorthy, 1996], a good SCM structure, integrated with the product design, can provide much of the infrastructural needs of both the developer and the software maintainer, can reduce overall projects costs, and will enable an increase in perceived product quality by both the customer and the software user.

7.4 Need for Statistics and Measurement

System designers should plan for metrics and instrumentation of the software system to meet the twin goals of being easy to gather and to make the most of the information content. The selection of appropriate metrics for system performance should begin in the requirements stage. The system design should consider customer and developer standards for reliability, and include specific methods for collecting metrics data.

Perhaps software systems reliability statistics are neglected at least partially because many designs do not include comprehensive definitions for system failure. Another cause for lack of reliable statistics may be the difficulty in gathering such information when system operators do not have access to a trouble reporting system. A third cause may be that the operators' observations of system behavior may be inaccurate or misleading in determining the actual cause of a problem.

The software system design should include a means for gathering failure statistics that augments user observation and discrepancy reporting. In some circumstances, the software could gather its own data, or a companion application could provide performance feedback to the developer. Automated reporting is the paradigm used by many newer PC desktop systems, where the operating system reports software failure information back to the manufacturer via the Internet. The system implementation may enhance the automated development of trouble statistics by creating a method of gathering of useful system state information at the time of the failure. Failure prediction, based on observed performance while the system is in actual operation, would provide a valuable tool in management of software maintenance.

In the transportation examples from preceding chapters, manufacturers and maintenance service providers benefit from the use of statistical analysis of performance data. System, sub-system, and component reliability statistics are based on factory test data, which is compared in-turn to data gathered on systems in actual use. These data form a basis for maintenance planning and the availability of maintenance resources.

The COCOMO model [Boehm, 1981] provides just one method for software engineers to predict failures using measurements of software complexity with variations on instruction or line-of-code counts. Experience with the organization, the language, and the software development tools used form the basis for these predictions. Performance statistics gathered for deployed systems would help to refine the accuracy of failure prediction, allowing more targeted quality assurance and maintenance activity on probable “bad actor” software and system modules.

Failure statistics would also enable the measurement of the effectiveness of the maintenance organization. Engineers use the Mean Time to Repair (MTTR) as a measure of design reliability in concert with the efficiency of the maintenance of the system. In ASP, the raw mean time to repair – as measured from the first report of failure to confirmation by the user that the software repair actually fixed the problem – was sometimes measured in years. Yet the system had no efficient means for testing software performance or measuring actual effort required to fix a problem. The maintenance support system did not allow for measurement of real maintenance efficiency.

There are systems that measure software reliability effectively. A software test or operational failure may require a change to the software. NASA maintains the space shuttle's flight control system software using an SCM system that captures useful change information, and helps to create performance statistics and accurate failure predictions [Ramamoorthy, 1996]. Developers, testers, maintainers, and quality assurance use the system to record the software change and metadata about the change with

> *"data detailing scenarios for possible failures and the probability of their occurrence, user response procedures, the severity of the failures, the explicit software version and specific lines of code involved, the reasons for no previous detection, how long the fault had existed, and the repair or resolution."*

NASA now regards this system as a necessity, and the information contained as the "*minimum set of data that must be retained regarding every fault that is recognized anywhere in the lifecycle, including faults found by inspections before software is actually built.*"

In turn, these data are used to develop statistical analysis and project estimation techniques [Ramamoorthy, 1996]. The shuttle system infrastructure supports performance measurement.

Chapter 8
Requirements For A Software Systems Maintenance Infrastructure

8.1 System Focus

A software system support infrastructure should address the complete environment in which the software operates. The lack of configuration control over the system created or exacerbated many of the maintenance problems in the ASP project. Lack of specific information on the network and hardware configurations made large deliveries necessary: to ensure delivery of the proper software to a specific user, engineers sent the entire software baseline via FTP. Deliveries of software repairs depended on on-line factory phone support, the skill of first-line maintenance engineers, and laboriously maintained delivery scripts. Specific configuration details often caused system problems. Extended downtime, apparently caused by software baseline changes, eroded the customer's confidence in the software product -- even when the software change was not the direct cause of the problem. The lack of confidence caused the user organizations to delay the deliveries of software changes.

In ASP, system operators and first-line maintainers often needed information available only at the factory. In some cases, factory maintainers disseminated special installation instructions to local maintenance engineers who had only limited training and experience with the system. The on-site maintainers were unable to use the special instructions because of a lack of training. Original system maintenance manuals became outdated when the systems changed.

The software system support infrastructure must consider the hardware and network configurations of the delivered systems, the operators and first line maintainers, the developer maintenance process, personnel training, documentation, problem reports, and the change delivery process. In addition, the developer has a requirement to maintain the existing software development infrastructure of source code control, requirement database, system test, and quality assurance until the developer no longer supports the software. Even after a product is “retired” by the developer, continued use of the product will require user or third-party maintenance with many of these same features.

The user may also need visibility into the entire system beyond the need for maintenance documentation. Even after a product is “retired” by the developer, continued use of the product will require user or third-party maintenance with many of these same features. In the case of ASP, the development contract called for the delivery

of the system source code with each compiled baseline. If the company later chooses not to support the software, or the customer chooses another contractor to maintain the software, to maintain the software the next maintenance service provider will need this software source code.

8.2 System lifecycle

Boehm [1981] and McConnell [1996], along with the Raytheon IPDS, endorse the choice of an appropriate software lifecycle for each project or product. The product lifecycle will be a primary design input to the software support infrastructure. The lifecycle and project scope will determine the probable size of the development and maintenance efforts. In contracted development, the customer may require a specified level of support over a given product lifecycle. The maintenance effort therefore becomes an essential part of the overall system design.

8.3 Software System Change Management

The struggle with the maintenance of ASP generated some interest in Software Change Management methods and tools in an effort to find the correct fix. In the ASP project, there was only limited SCM support and initially no customer DR support. While maintainers were able to assemble a working solution to the problem, the system still lacked some functionality.

Further analysis led the team to the discovery that the problem went beyond SCM and bug reports, and beyond just the maintenance phase of the Software Development lifecycle. The company's Integrated Product Development System (IPDS), although it provides solutions in the hardware manufacturing and aircraft fields, lacks a systemic process outline detailing complete life-cycle support for software development. While there are processes specified for software development support in IPDS, there is a lack of connectivity between critical areas and a lack of a common infrastructure. Maintainers needed a new model for software development support infrastructure.

As shown previously, software applications have similarities to most other types of products developed in industry. The differences between software and other products are important to the infrastructure model. Developers often create software in a trial-and-error manner at the modular level, and the modules themselves do not often work very well together at first try. Rather than a precision manufacturing process, large software projects more resemble the custom tailoring of a suit. The developer changes each portion incrementally until the whole application works as intended, or at least operates within the design parameters. The development process therefore

requires frequent changes to software source code. Moreover, software change in even a small project using only a few developers is a complex process.

Software engineers, especially those working on large projects, discovered early the value of a software change management system. SCM software tools were an obvious response to the problems of version control, software configuration, and developmental change, using software to control software is a natural match. The software system model should apply the same paradigm to the lifecycle of the entire software environment.

8.4 ASP SCM Problem

Most software change management tools only manage the target application source code. In the quest for more reliable software development practices for large projects software requirements, test tools, test records and documentation also require change and version control. System requirements change frequently, engendering changes to software test scenarios and testing tools. Hardware specifications may change in the development of a large system, also dictating software changes. Software and requirement changes are likely to create the need for changes to system documentation -- whether the developer delivers that documentation to the customer or maintains it internally. In the maintenance phase of a deployed software system, performance problems and application bugs discovered by the users are also agents of change to the software system, and must be trace-able from their origin to all the permutations of changes to all the delivered products.

Software development organizations often place the burden of documenting the justification for a change to the software on the programmer. In order to maintain the code efficiently, programmers insert comments into the source code so that later programmers can understand the reasons for the specific method used to solve a particular programming task or problem. Often, however, the best programmers are the worst at writing comments to code. Neither software productivity measurement tools nor the development environments are conducive to writing good comments or documentation: organizations often judge productivity by the number of Lines of Code (LOC) generated and driven by a production schedule. Software comments are not necessarily available to the test or documentation groups, who may not even become aware of a fundamental change to the system.

Configuration management and document control are mature fields in mechanical, electrical, aerospace engineering disciplines. Much of industry uses efficient requirement and product deficiency tracking processes, and these are utterly necessary in software maintenance. Development organizations use software-based support tools to address all the different facets of development, deployment, and maintenance processes of complex systems, but in

only a few cases are the processes and tools integrated into a coherent system. What is needed is a system-level process to map not only software requirements and changes to specific source code, but also a repository of the tests of that source code that demonstrate that the actual functionality satisfies the requirement. A systemic approach to software infrastructure support would also correlate user-discovered problems to software changes, and software changes to documentation changes.

Several technologies approach the solution to the software-specific part of the problem. Extreme programming traces requirements (stories) to tests, then to the software source code, then to design documentation in a re-entrant cycle that involves the customer in each iteration. Eiffel, one of the first object-oriented languages, also introduced the idea of design-by-contract, where software source code is self-documenting by virtue of the requirement decomposition system into pseudo code that is compatible with the programming language. Advocates state that CASE tools address the entire software lifecycle, but these are often only programming or design tools that fail to address the detail work of SCM, software changes, or testing. Rational Corporation provides software tools to tie requirements and bugs to software, but does not provide a tie to documentation or testing. It is clear that programming process, language choices, and support tools can only address facets of the support infrastructure problem, and cannot provide a complete solution to a high system-level problem.

Raytheon's IPDS provides a process for software product development from initial inception to retirement, but the model segregated most of the sub-processes (configuration management for example) at the arbitrary boundary between software and hardware engineering. Hardware has its own sub process for configuration management while software engineering maintains a different set of checklists and general requirements for a configuration management system. The hardware design and manufacturing disciplines dominate test and reliability engineering in IPDS, therefore IPDS designs, processes, and checklists often do not fit the software process model. The documentation control process also differs between software and hardware. Methods between the disciplines seem to differ enough to make the tools used to accomplish the tasks in the separate venues incompatible, yet the basic principles and goals are the same. A cradle-to-grave system development process should use a comprehensive tracking and management software tool to accomplish all the common support functions. The tools developed to provide the infrastructure should support and model the in-place processes, regardless of the product, and should be a requisite component of a product development infrastructure.

8.5 Support System Model

Processes and integrated software tools provide infrastructure support. IPDS models the overall development process, so software systems architects can use it as a reference. As part of the software design process, software architects should specify comprehensive support system requirements; however, the software support infrastructure should meet general requirements and functionality.

The final solution to the problem must include the functionality if current software configuration management systems, current document management systems, current systems and engineering processes, current bug-tracking and current test processes. If integrated with the final product, the system of software and processes will add the functionality of performance measurement at modular component and higher levels and traceability for the entire product lifecycle, providing the organization the ability to maintain the software product properly.

8.6 The Software Development Infrastructure

If designers use either the spiral or recursive waterfall models of the software lifecycle, then software maintenance must begin and end with software development. The target software application is the focus and context of the support system. To describe the requirements of a software system maintenance infrastructure, we can begin by examining the infrastructure for software development.

8.7 Requirements Database

The stated and inferred customer needs for the application must be stored, changed, and updated throughout the life of the project. Initial statements of customer needs are necessarily vague, and sometimes use the customer's own jargon; which may be different than that of the programming team. Customer requirements should evolve from general, loosely stated needs into engineering specifics. The developer also has requirements for the system, usually in terms of marketing strategy, costs, performance, and other factors. Developer requirements are refined into product functional requirements. Software engineers further decompose both sets of requirements into a combined set of specific functionality.

The most basic developer requirement in the commercial world is the maximum cost of developing and delivering the application while maintaining profitability. Non-commercial developers, such as those in the open source, government, or in-house application programming communities also have functional requirements for the target application.

Requirements continue to evolve throughout the development process. As the developers implement the application, each requirement may need revision to meet actual cost targets, to improve the developer's understanding of the customer's need, to clarify the customer's actual from perceived needs, or to provide for a path of future development. Occasionally, the final application cannot meet a customer or developer need, and engineers must modify the requirement derived from the need statement to reflect a compromise. Alternately, an unrealistic or infeasible requirement is a candidate for removal or deletion.

Refined requirements tell the development team how to accomplish each part of the development task. It used to be axiomatic that the requirements were used as a measurement of completeness, letting the developer "know when [the product was] done." In the continuous development model, where software system continue to evolve, the concept of being "done" may no longer be applicable. However, requirements do form a basis of comparison for measurement of the project's progress.

Keeping track of the project requirements is essential throughout the life of the project even past the retirement of the application – transcending the lifecycle. Later, others may use historical records of a defunct project as teaching material, in conjunction with other historical records as "lessons learned," as data for performance measurement and estimation-prediction, and as a potential source of future project ideas and opportunities.

Any complex software project requires an adaptable, easily maintained requirements database, coupled with the organizational processes to populate and maintain it. Usable databases, in turn, require a serious effort to design, create, service, and use. A good database requires a database administrator, a database manager, and its own programming and data entry teams.

COTS requirement database tools must be adapted to fit the organization's processes. The developer potentially becomes a customer: management must make the build, buy, or reuse decision for the database. The developer organization will use the database throughout the target product's lifetime.

The requirements database is the first example of a complex development support subsystem. The database will also have a lifecycle, from needs and requirements, through cascading and recursive development, delivery, maintenance, and eventually, retirement from active use. The data (separate from the database) become part of the historical record of the entire project.

8.8 Design Tools

Design methods differ between organizations. Design tools should fit the organizations processes. Established software design methods wax and wane in popularity, and new methods are appearing constantly. Organizations also evolve and change methods. Design methods and tools will change with the organization.

As with the development of requirements, design decisions are significant to the project in a historical sense, beyond their immediate usefulness to the application design. Design information should become historical information, organized in a specialized database.

Customer requirements are the main input into the design process. The system designer chooses the basic approach to the technical solution. The high-level design is recorded, and like the initial system requirements, evolves into the specific design and the final implementation.

8.9 Coding Tools

The use of a specific software programming language often dictates other later design decisions. In the maintenance and update phase of development there is often little choice of language. The choice of language in the design phase may drive the ability to support modification of code in maintenance.

Structured Programming and Object Orientation are technologies and coding strategies designed to enable the ease of maintenance and reuse of software source code. Either strategy is considered industry standard. Unfortunately, the mere use of tools capable of object-oriented implementation does not necessarily create easily maintained object-oriented source code. The designer choosing the language should consider the later maintainability of the code.

As seen in the transportation examples, standard components make much of the maintenance process less expensive and encourage standardization of design. Fastening hardware, a simple and common example, uses standard material, strength, shape, and service life specifications, allowing designers to choose the mechanical hardware most suited for the design. If appropriate off-the-shelf components are not available, the mechanical engineer can describe a customized part using standard terminology. Component architecture has been slow in coming to software. Both structured programming and object orientation technologies promised ease of design and reuse of modular components, such as is the case with hardware. However, the slow adaptation of standard descriptive terminology for performance and character of software modules has prevented real software components

from becoming an industry reality. Groups of programmers have been able to establish localized software component reuse through enforcement of programming standards and quality assurance. Like hardware components, software component specifications should include test performance data.

8.10 Extended Source Code Management

The SCM system could be more than a repository of source code, or even a management tool for accessing such a system. A typical good repository performs source code versioning, includes and enforces source code change process, keeps source code inspection/check-in approval system records, provides an audit trail (user mapped to check-in/change), links to the development system configuration for testing, provides a software build tool support, and run-time (patch) delivery support/ release control.

The typical SCM/support infrastructure should also contain software and processes to support:

- Integrated Design Requirements
- Design Details
  - Interface Definition (External and Internal)
  - Software And Hardware Implementation Specifications And Limitations
  - Run-Time Configuration Specification
  - Build/Development System Configuration
    - Compiler/Linker Options
    - Compiler/Linker Tool And Version Information
    - "Standard" Library Versions
    - Make Files (*or language equivalent*)
    - Versioning
  - Integration Information
- Testing Tools (*Both software and procedures. Should show how the software meets specific requirements)*
  - Component Level Tests (as a Programmer Deliverable) White Box Test Stubs, Scripts, Tools for Testing Smallest Components of Project Code
  - Higher Level Test Structures, Scripts, Procedures at a Modular and Program Level
- Testing Records
  - Lint (*or language equivalent*) Testing of Source Pre-Compile
  - Build Errors/Warnings From Compile/Link Phase
  - Pass/Fail White Box and Black Box Testing
  - Customer Acceptance -- Upon Demonstration
  - Software Performance Records
- Code and Document Inspection Records

- Other System Documentation (*automated and paper based user and maintenance manuals, vendor supplied documentation*)
- Defect Reporting System (*software is created from requirements, but changed by discrepancy*)
  - Defect Report Repository
  - Correlation to Change
  - Correlation to Requirement
  - Correlation to Test Results
  - Metrics Generation (MTTR, MTBF)
- Change/Merge Control Software and Process
- Field Integration Details
  - Results of field test and usage
  - "Soft" Configuration Information (*system admin specifics, required operating system and network settings)*
  - Hardware Configuration Details
- Delivery Details
  - Automated Delivery Software
  - Delivery "Meta-data" (*software changes included in a given delivery or version release, method of delivery, installation instructions)*
  - Delivery Software
  - User Configuration Variations

Many of the software design requirements generated by such a support system resemble those of a transaction database. Database designs include levels of access, a variety of views of the interconnected data, and a need for administration and management. Like any good data base design, a good software control infrastructure should be platform independent (no specific repository tool), data independent, and should include human-followed procedures as well as machine software. System designers should tailor the system to the project at hand, and initial support system design should allow for expansion and evolution.

The basic premise of the proposed SCM model is that a large software project is the result of a cooperative effort between experts of many disciplines. The SCM infrastructure tool would provide support for requirements, design, development, test, integration, maintenance, and quality assurance. A software development infrastructure should therefore provide support for the entire development system over the lifespan of the product. SCM is a necessary activity in any software development activity. The SCM tool provides for storage of many versions of individual files, assigning version numbers and identifying versions by date. The shuttle case study [Ramamoorthy, 1996] showed expansion of the existing code control system to include quality assurance, test data, and some user documentation along side test code and simulation data. Boehm [1981] and McConnell [1996] point out that

requirements and design both change over the life of the project. ASP user documentation and delivery meta-data required update and recording, and previous change data were not available to maintainers. The SCM existing system seems a natural fit to store all these data, creating a single integrated database that includes all data produced during the project lifecycle.

The proposed system differs slightly from the NASA/IBM/LORAL system [Ramamoorthy, 1996] in that it includes hardware configuration modeling (the shuttle system uses a single computer type), manuals, user trouble reporting, delivery software, and test procedures. The implementations of the SCM infrastructure tool may vary; for example, testing of a given software system, if it is not as life-dependent system as the shuttle application, may be less extensive than that of the shuttle system. A software designer should choose the basic infrastructure functionalities that support the system, allowing for future changes and adaptations.

Chapter 9
Conclusion

Ramamoorthy [2000] makes the comparison of software engineering to the service industry using the following distinguishing characteristics of service functions:

- Human (customer) needs driven
- Knowledge intensive, high mental effort
- Automation intensive to reduce manual effort
- Human interaction intensive
- Information technology intensive
- Team based

By analyzing the software maintenance subset of software engineering in terms of these characteristics, it is apparent that post-delivery development activities are also a service. In some contexts the English words for *service* and *maintenance* have the same meaning. The maintenance organization should efficiently provide the software system maintenance service to the software user.

Software development organizations are oriented to delivering the product to the customer. Along the road from initial program concept to delivery, all available resources and organizational processes are concentrated on the definition, design, implementation, integration, and delivery of the product.

The software community may underemphasize maintenance because the product seems different from all the other kinds of engineering products that require maintenance. Popular jokes aside, aircraft and automobiles cannot be repaired with a simple “reboot,” but require a complex service and maintenance infrastructure to adequately provide transportation services. Roads, bridges, and buildings require maintenance to provide their transportation and shelter services to their respective users. Computer hardware is subject to failure and component or assembly replacement. Like software, the life spans of most "hard” engineering products are less in development and creation than the total span in maintenance and use. Sustainability – the property that allows these products to remain functional – is designed into these products to extend the service life of the product and to minimize the total cost of ownership.

Software seems different than other engineering products to its users and creators. Software does not wear out and does not require fuel, lubricants, or replacement of parts. A software application may “live” as long as it

continues to provide an *essential service* economically, is reliable, and is supported by available hardware. Users will replace software that does not continue to perform its function – software that is no longer providing an essential function to or on behalf of the users is superceded by software that performs. Easily repaired and upgraded software is likely to have a greater life span than that which is expensive, difficult, or impossible to maintain.

Software is subject to error and the need for modification after delivery. Software is a portion of a complex system, and is likely to be delivered without a comprehensive test of the complete system. Software systems are almost sure to be defective on delivery, although the defect may not be readily apparent.

Software that is intended to be sustainable must not only take advantage of modern languages and software management practice – sustainable software should be designed to be maintainable and part of an overall system served by a deliberately designed system maintenance infrastructure. In almost every other field of engineering, it is axiomatic to plan for the sustainability of the product. By employing a support infrastructure that provides for product maintenance after delivery, the products themselves serendipitously often become more robust. This process of sustainable development would also benefit software.

Boehm [1981} makes the distinction between software production and software lifecycle productivity, stating that by lowering development cost through reduction of system reliability increases annual maintenance costs by the same amount. Lifecycle productivity is increased by high system reliability, in turn created through use of "*better structured and documented software, program support library procedures, and reliability-oriented aids such as diagnostics, environmental simulators, and test data management systems."* Boehm advocates use of these tools and techniques in spite of increased development costs in order to reduce maintenance costs.

Appendix 1
UML Design for Maintenance Support Infrastructure

Data Flow Diagrams (DFDs, presented in the following appendix) excel at the application of the analysis and design portion of the process in a data-dependent project (e.g. database), but UML offers a much more "natural" way of thinking about and representing real system processes and actors. UML can not only describe and analyze the process, but to decompose the high-level processes down to its smallest component parts. The diagrams represent only a few of the subsystems that are part of the infrastructure support system.

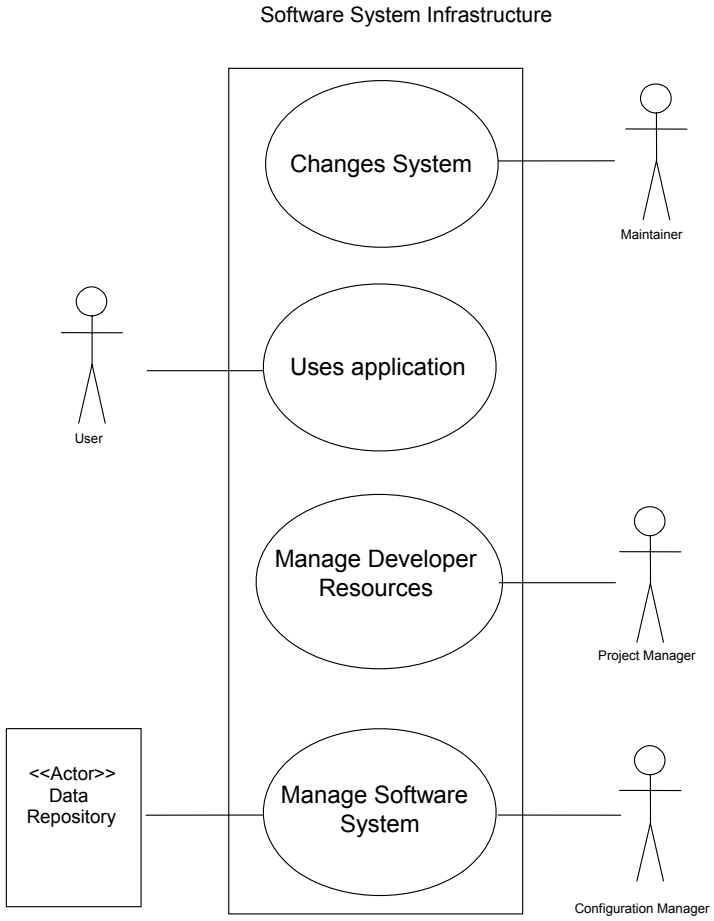


Figure 10 High Level Use Case Diagram (Maintenance Mode)

Figure 10 shows a high-level use case diagram of the proposed software system support infrastructure as it might appear in the maintenance phase of a project. Figure 12 shows the same system in the development stage. This figure is used to illustrate the evolution of the system. Time “snapshots” of the system at different stages in the

lifecycle would display different usage than figures 10 and 11, but some activities, and the infrastructure itself, would remain essentially the same.

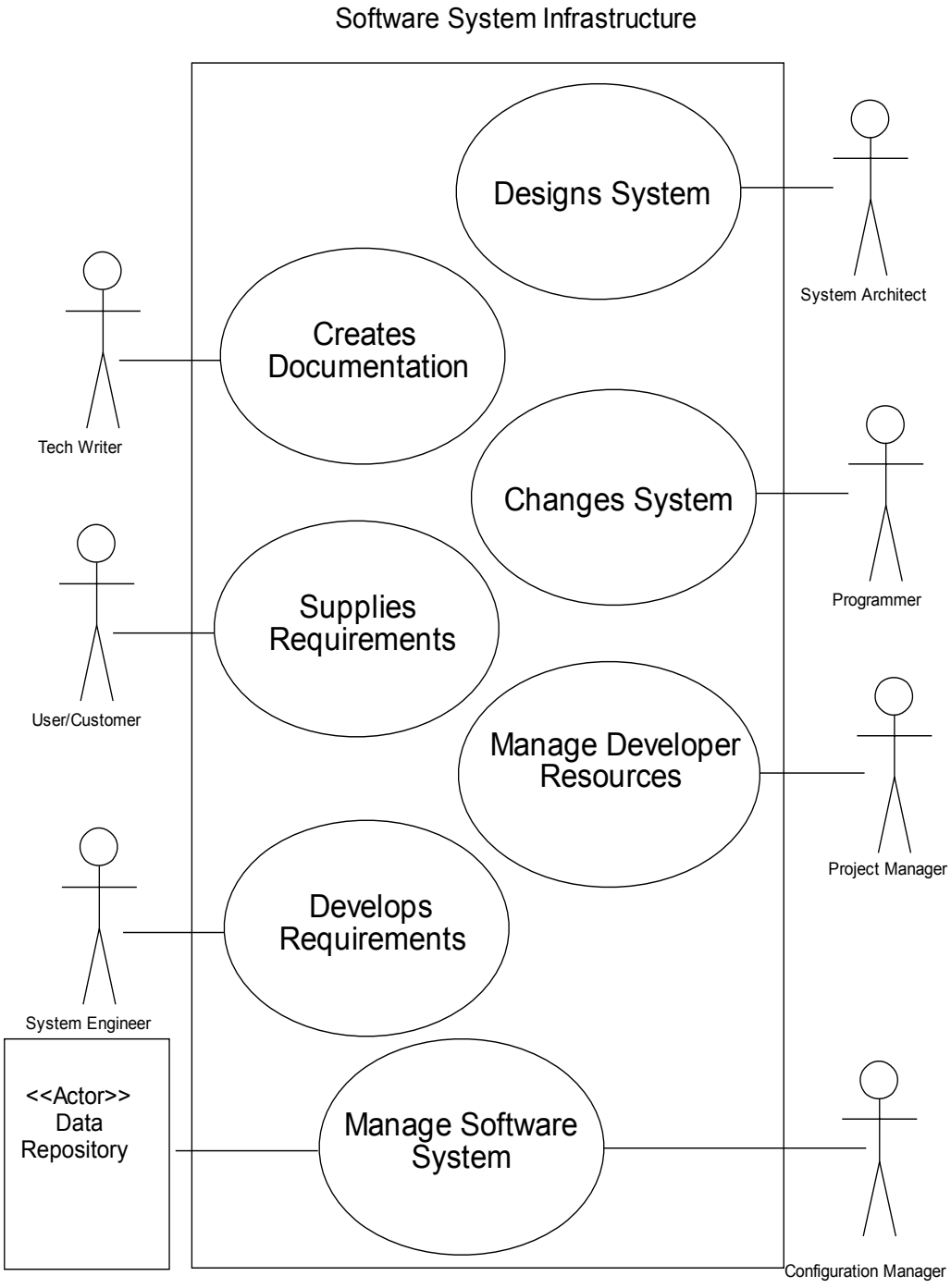


Figure 11 High Level Use Case Diagram (Development Mode)

Every interaction in the system requires development of the interface. Figure 12 shows the typical maintenance phase interaction of the user with the maintainer and the support infrastructure system for just one type of transaction. In this case, the user's report generates a code change, requiring several interactions with the maintainer. In this case, a single UML actor figure represents the maintainer -- who may actually be more than one individual. Other interactions at this level might include documentation changes, requirements changes, and user configuration changes.

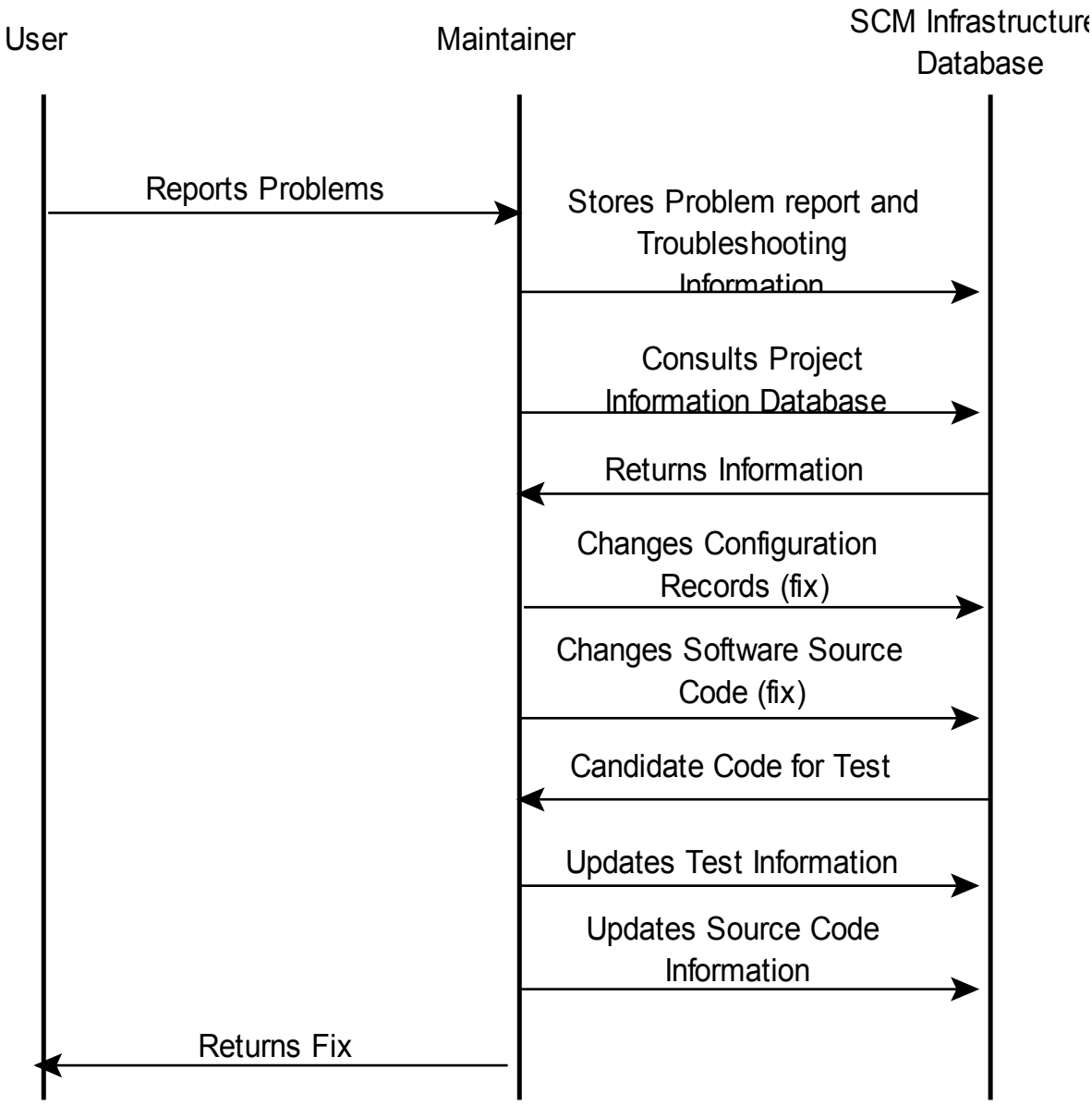
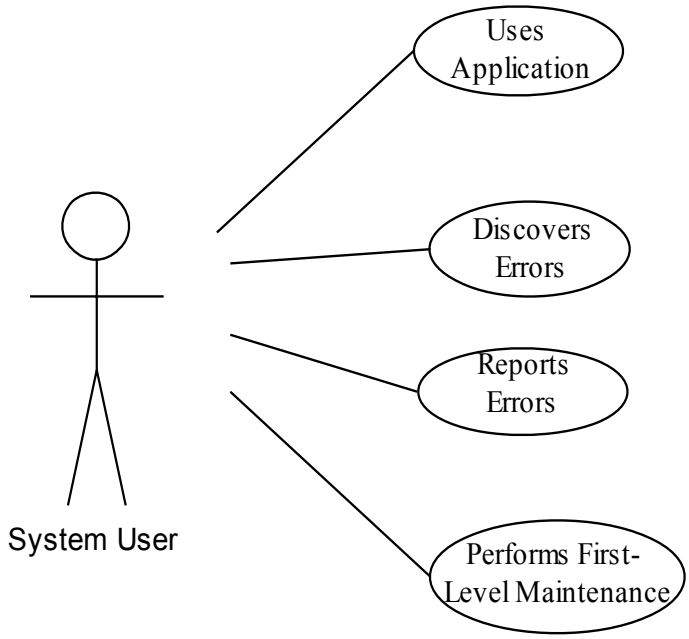


Figure 12 UML Sequence Diagram

Figure 13 shows two detailed use cases from the maintenance phase. At this level, use case diagrams show details from the perspective of the actors.



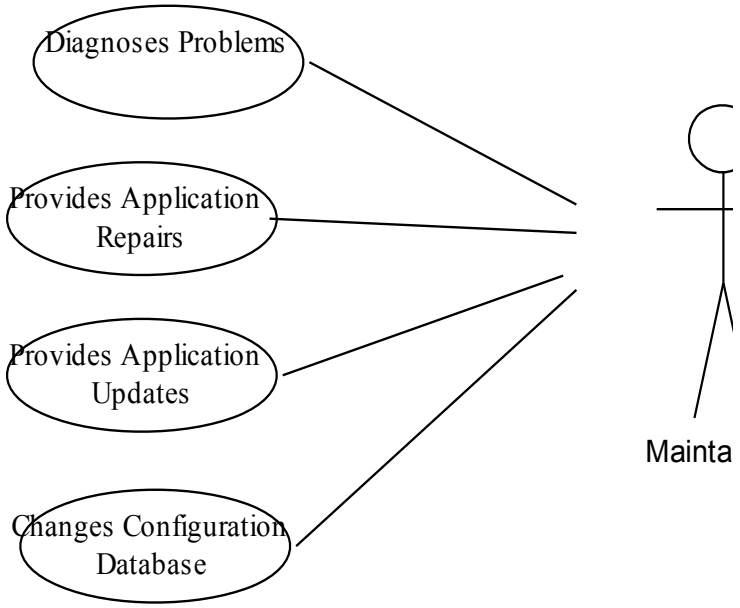


Figure 13 UML Detailed Use Case Diagrams (User and Maintainer)

Appendix 2
Systems Support Model Using Data Flow Diagram

Data Flow Diagrams are not especially conducive to determining relationships in a fully developed process model. Analysis of the modeled system must take place outside of the diagramming of the model. The system must first be described (in this case by the author's experience and by the Raytheon IPDS), then decomposed into inputs, outputs and processes.

Table 2 Data Flow Inputs, Processing, and Outputs

Inputs:
- Target Software System Requirements

Processing:
- Storage and retrieval of all artifacts
- Versioning of all process artifacts
- Code compilation support (build)
- Test Plans and Procedures (white and black box testing)
- Automated test (White box testing of code components, the software to do this task, the results of the tests)
- Design documentation at all levels.
- Source Code for the target application
- Audit for software changes, code generation, and delivery
- Source Code Changes trace-ability (who did it, why, what end-item received the change, how well did it do)
- Requirement Changes: reasons and affects on other portions of the project
- Metrics collection on all phases of development and deployment, test, repair, failure, change.
- Internal inspections of requirements, design, source code, test.
- SOFTWARE CHANGES during development and post-deployment
- User manual, installation and configuration instructions, customization, configuration details
- Business metrics
- Validation and Verification in the form of customer and/or quality champion acceptance testing

Outputs:
- Application Software
- Internal Documentation
- External Documentation

DFD Drawings

DFD:0 Context Diagram
Software Support Infrastructure

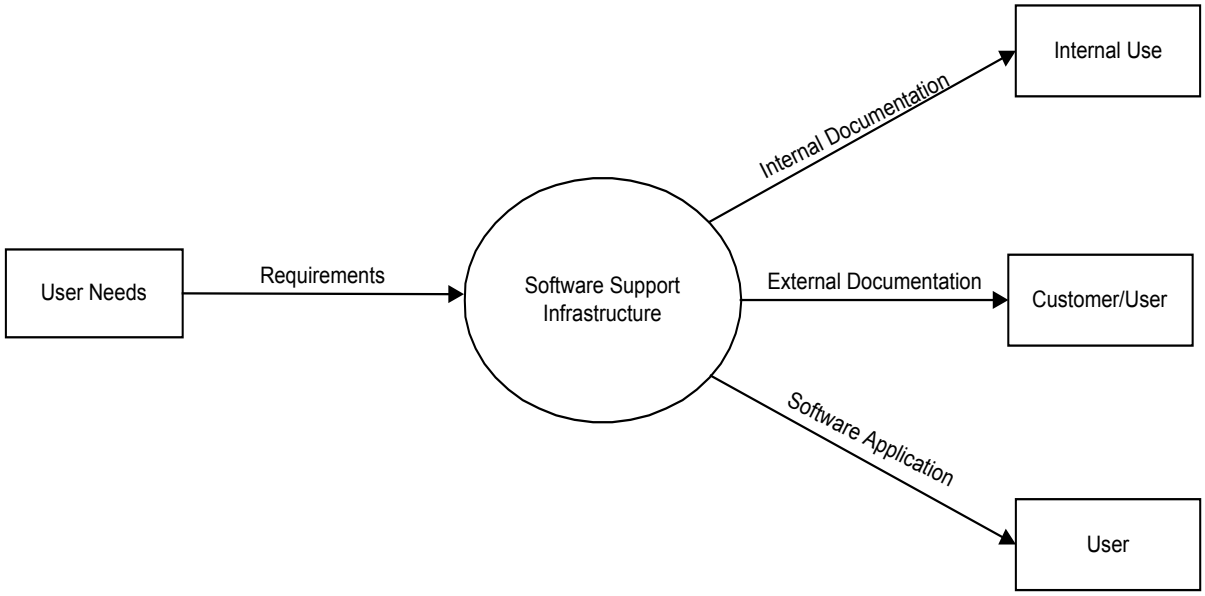


Figure 14 DFD: 0 Context Diagram for Software Support Infrastructure

At the highest level, the diagram shows the fundamental input and output of the entire system, essentially reflecting the requirement table, above.

DFD:1 Infrastructure Data Flow

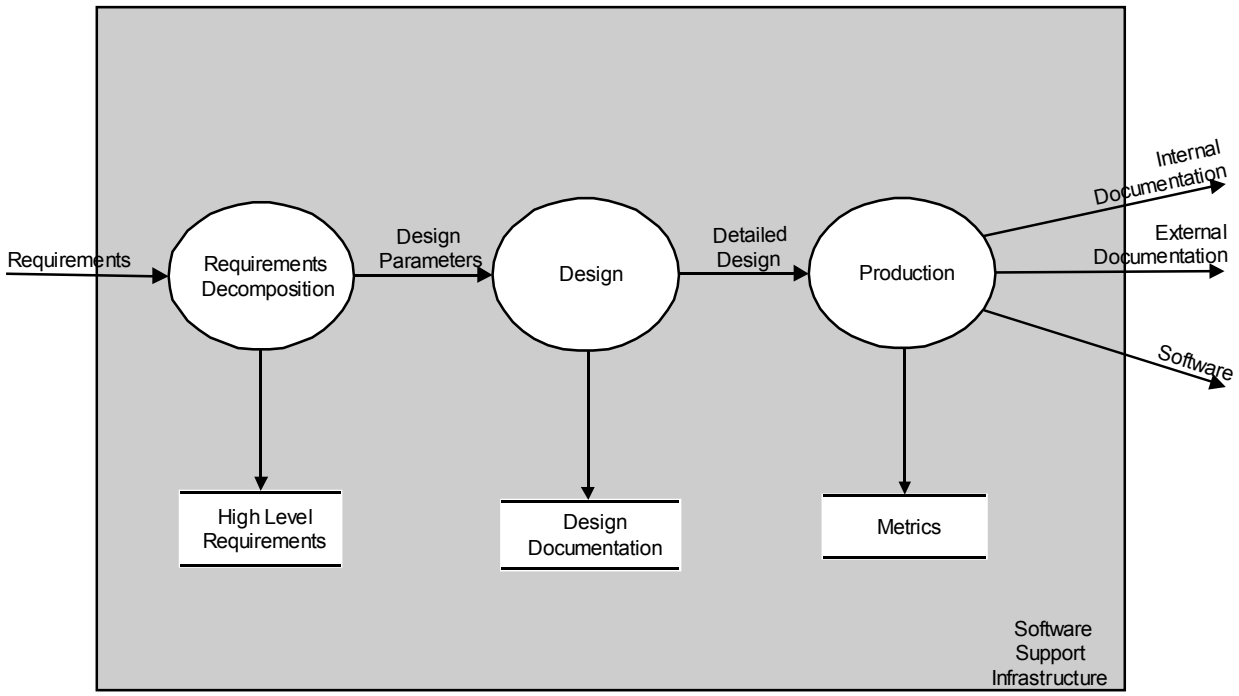


Figure 15 DFD 1: Infrastructure Data Flow

At the infrastructure level, the system appears slightly more complex. The simplified drawing does not show the business development or maintenance phases of the lifecycle and is "scoped" to just requirements development, design, and coding portions of the overall process. These are consistent with the corresponding portions of the high-level IPDS.

DFD:2.1 Requirements Decomposition
Data Flow

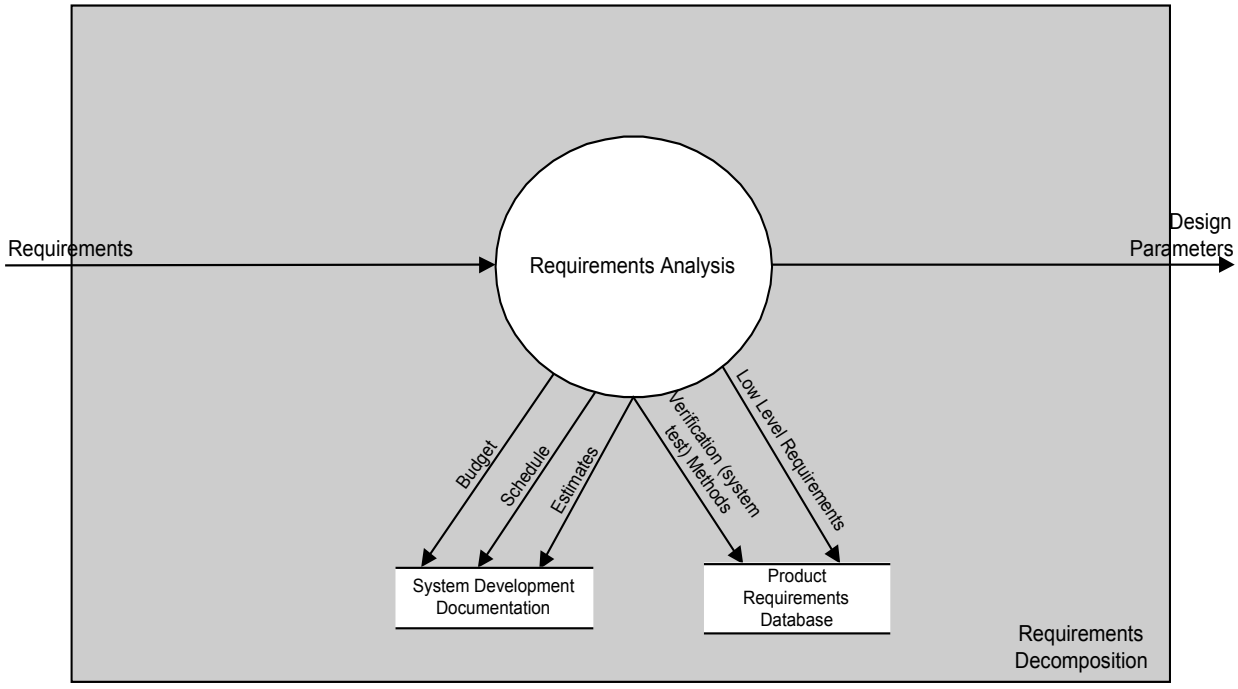


Figure 16 DFD 2.1: Requirements Decomposition Data Flow

This portion of the diagram illustrates a simplification of the decomposition of user needs to specific, testable design requirements and the specification of some high level functions of the final product. Data capture and early development is essential at this stage. While most of this block is represented in the IPDS model, the development of test procedures in parallel with requirements is not a current practice in many parts of the organization.

DFD:2.2 Design Data Flow

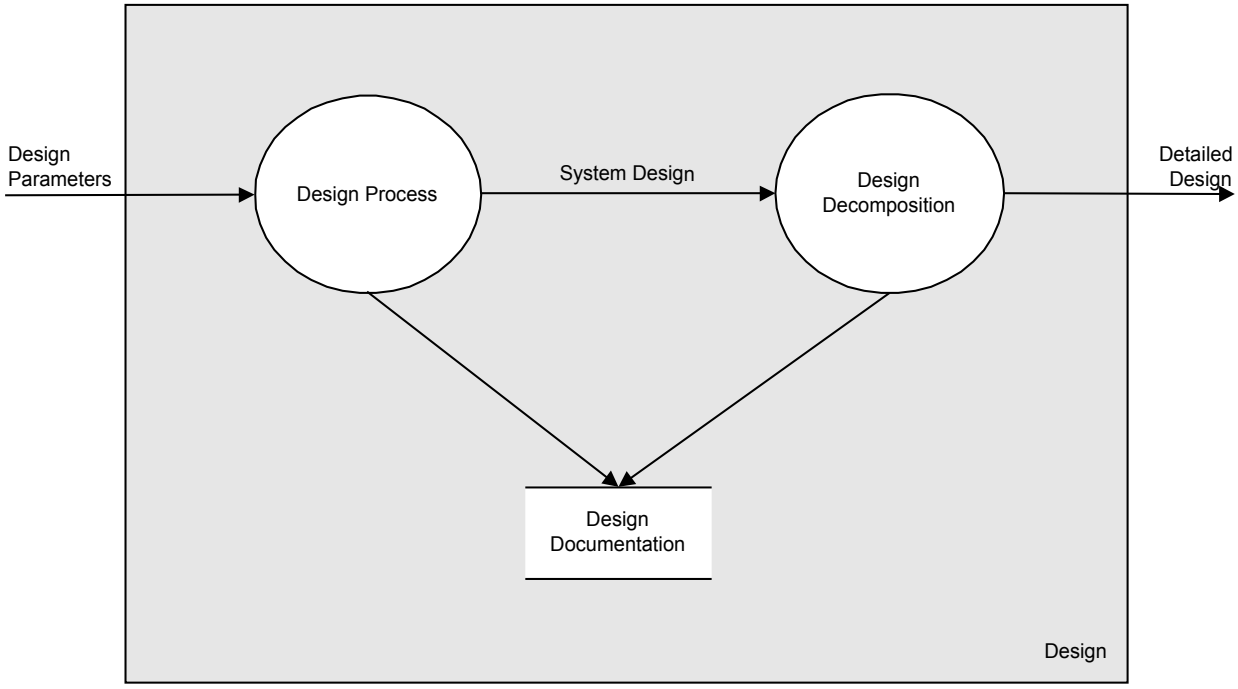


Figure 17 DFD 2.2: Design Data Flow

The high-level design is developed from high-level design parameters, and decomposed into more specific design requirements to be implemented at lower levels. This should be a recursive process with the requirements development phase, and is one area where UML might offer a better representation.

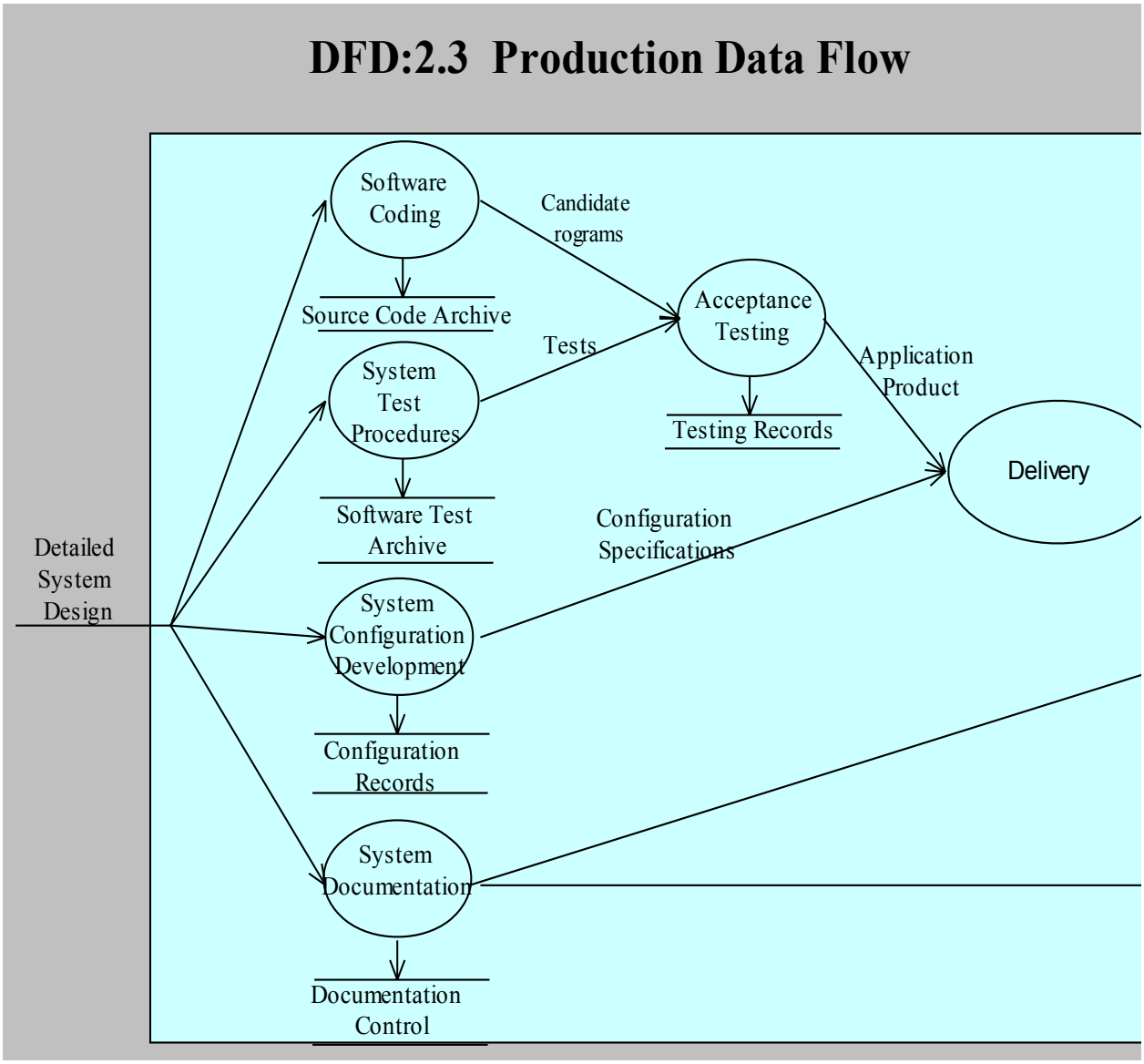


Figure 18 DFD 2.3: Production Data Flow

The most complex of this set of drawings, the production data flow diagram displays the majority of the processes that must take place in this model. This phase of development should include documentation of performance testing of application sub-systems. Test data should be captured, not only to demonstrate the adequacy of the program but to support maintenance efforts as a measure of effects of changes.

Testing at this level implies the use of either the target system or a simulator as a test-bed, so that test results can attempt to verify the functional adequacy of the sub-systems. Also implied by testing at this stage the use of standardized, repeatable software tests, the design, validation, and implementation of which should recursively

follow the same process as development of the target product software. Testing at this level is of a classic "black box" style, showing that the software provides the expected outputs from a set of reasonable inputs. Later demonstrations for the customer may repeat this same testing.

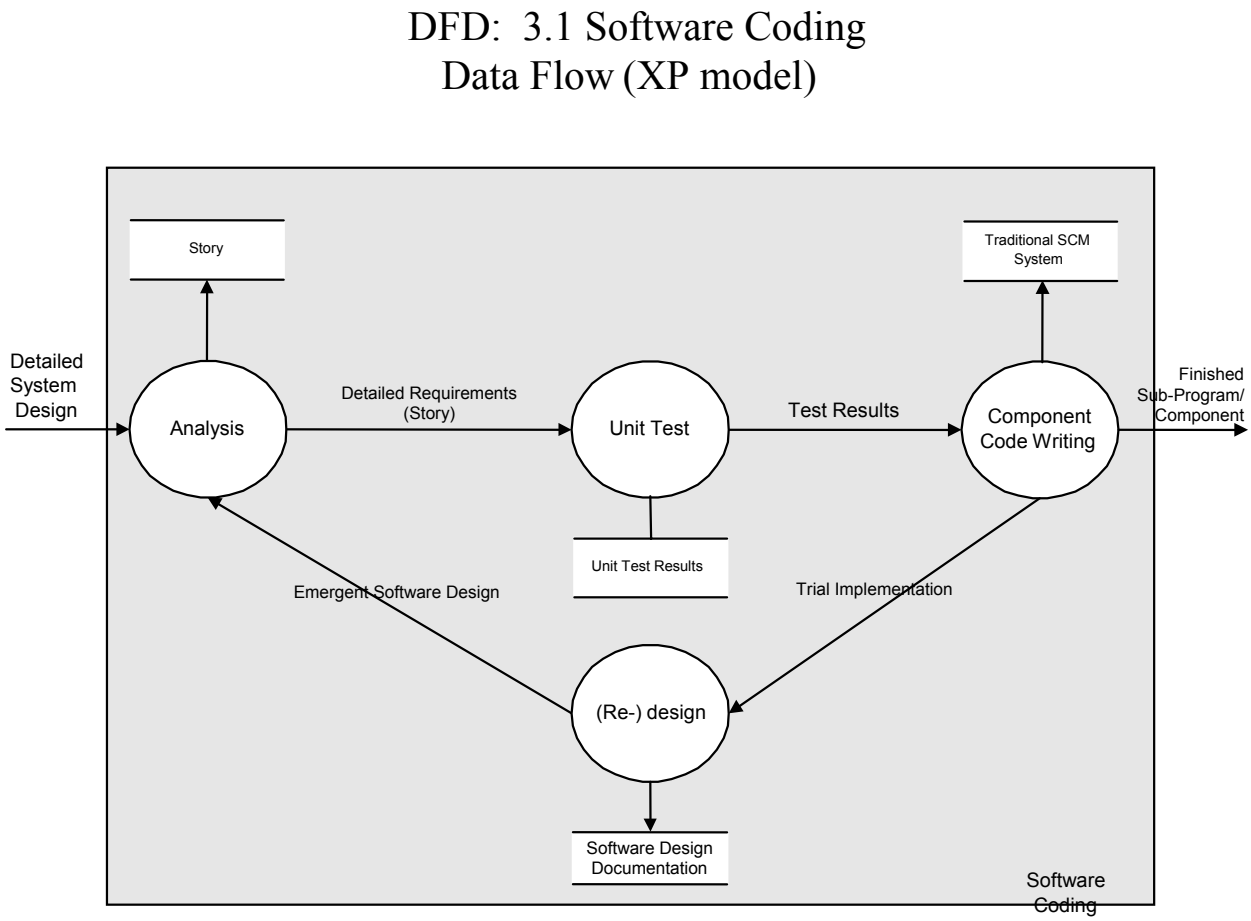


Figure 19 DFD 3.1: Software Coding Data Flow

For software coding processes, the drawing uses the Extreme Programming (XP) model to illustrate the encapsulation of almost any process model within the overall system. In fact, many different low-level processes could be used at this level of the infrastructure. The infrastructure support tool would have to allow for the adoption of new methods at the lowest levels, while retaining their integration with other peer-level processes and with higher levels.

If the normal coding standards and procedures are followed at this stage, code teams engage in peer review inspections of source code. The code review process generates artifacts and inputs to metrics gathering processes,

which should also be captured for archiving. While peer inspections help ensure that the software source code meets coding and comment standards, the inspection records themselves also might provide insight into the programmer's problem-solving methods. Any insight into the original programmer's thought processes are valuable to another programmer in the maintenance phase who is attempting to provide a quick fix for a user-generated bug.

Testing at this stage is "white box" testing, where program teams write stubs and drivers to attempt to test every line of code. These stubs and drivers, low-level test tools, should also be captured to provide diagnostics for later maintainers of the code. Metrics gathered on productivity should include the production, debugging, and maintenance of the testing tools alongside the production code. These tests tools, as well as the higher-level tests, could also provide the beginnings of a diagnostic suite for the target system; useful at integration and to later troubleshooting -- perhaps even delivered to users to aid in field maintenance of the code.

References

[Alamares, 2001] Alamares, Joanne, *Test and Evaluation, a Process,* Texas Tech Master of Engineering Report, June 29, 2001. Available as pdf from http://www.theatlasnet.org

[Boehm, 1981] Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall Inc., Englewood Cliffs, NJ 08632, 1981

[Boehm, 1998] Boehm, Barry W., *A Spiral Model of Software Development and Enhancement,* IEEE Computer, 5, 61-72, May 1998.

[Boehm, 2000] Boehm, Barry W., *Spiral Development: Experience, Principles, and Refinements*, Presented February 9, 2000, edited by Wilfred J. Hansen, Carnegie Mellon Software Engineering Institute, Pittsburg, PA 15213-3890, available http://www.sei.cmu.edu/cbs/spiral2000/february2000/SR08.pdf, July 2000

[Boehm, 2001] Boehm, Barry W., *Software Economics,* presented at the Software Design and Management Software Pioneers conference, Bonn, Germany, June 29, 2001, available http://www.sdm.de/dt/tec/eve/2001/ppts/f_8_boehm.pdf, 2001

[Brooks, 1987] Brooks, Frederick P Jr., *No Silver Bullet: Essence and Accidents of Software Engineering,* IEEE Computer; 20, 4, 10-19; April 1987

[Brooks, 1995] Brooks, Frederick P. Jr., *The Mythical Man-Month, 20th Anniversary Edition,* Addison-Wesley, Reading, MA, 1995

[Conger, 1994] Conger, Sue A., *The New Software Engineering*, Wadsworth Publishing Company, Belmont, CA 94002, 1994.

[Daich, 1996] Daich, Gregory T., *Emphasizing Software Test Process Improvement*, CrossTalk, The Journal of Defense Software Engineering, Software Technology Support Center. Hill AFB, UT, available http://www.stsc.hill.af.mil/crosstalk/1996/jun/emphasiz.asp, June 1996

[Finkler, 1992] Finkler, Steven A., Ph.D., CPS, Finance & Accounting For *Nonfinancial* Managers, Prentiss Hall, Paramus, NJ 07652, 1992

[Goldsmith, 2001] Goldsmith, Mike, *The Software Development Lifecycle for Small to Medium Database Applications*, Small Systems Solutions, http://www.sss4d.com/Ref/SDLCsmdb.pdf, 30 January 2001

[McBreen, 2002] McBreen, Pete, *Software Craftsmanship*, Addison-Wesley, Boston, MA, 2002

[McConnell, 1993] McConnell, Steve, *Code Complete*, Microsoft Press, Redmond WA, 98052, 1993

[McConnell, 1996] McConnell, Steve, *Rapid Development*, Microsoft Press, Redmond WA, 1996

[McLeod, Smith, 1996] McLeod, Graham and Smith, Derek, *Managing Information Technology Projects*, Course Technology, One Main Street, Cambridge, MA, 1996.

[Ramamoorthy, 1996] Ramamoorthy, Chitoor V., *Statistical Software Engineering*, Panel on Statistical Methods in Software Engineering, National Research Council, available http://books.nap.edu/books/0309053447/html/index.html, 1996

[Ramamoorthy, 2000] Ramamoorthy, Chitoor V., *A study of the Service Industry – Functions, Features, and Control*, IECICE Transactions Communications, Vol. E83-B, No. 5, May 2000

[Raytheon, 2002] Raytheon, *IPDS 2.2.0 Global View,* Raytheon Integrated Product Development System Release 2.2.0, available on Raytheon intranet http://ipds.msd.ray.com/Current/ipds/enablers/methods/global.pdf. 2002

[Royce, 1970] Royce, W. W., *Managing the Development of Large Software Systems: Concepts and Techniques*, Proceedings of the IEEE, WESCON, 1970.

[Schneiderwind, 1987] Schneiderwind, Norman F., *The State of Software Maintenance*, IEEE Transactions on Software Engineering; SE-13, 3, 303-310; March 1987

[Trewn and Yang, 2000] DR. Jayant Trewn and Dr. Kai Yang, *A Treatise On System Reliability and Design Complexity*, Proceedings of ICAD2000, First International Conference on Axiomatic Design, Institute for Axiomatic Design, Cambridge, MA, June 21-23, 2000.

[Viewz, 2002] Author unknown, *Operating System Guide, Introduction to Windows,* http://www.viewz.com/shoppingguide/osprint.htm, Viewz, Canada's Online Computer Guide, Published and Edited by Stewart Hall, downloaded 4 September 2002, copyright Viewz, 2002

Additional Resources

The *Graphical Development Process Assistant,* from the University of Bremen, http://www.informatik.uni-bremen.de/gdpa provided diagrams and some English language discussion relative to Royce and Boehm's interpretations of software lifecycle models.

The following resources also provided reproductions of graphics attributed to Boehm and Royce.

http://sunset.usc.edu/~jungpark/research/COCOMOII_SERI.pdf

[Yue] Essay: *Software Cost Estimation*, Timothy Yue http://sern.ucalgary.ca/~tyue/seng621/essay/sce.html, University of Calgary, downloaded September 6, 2002,