Processes to Produce Secure Software

Towards more Secure Software

Volume II

Contributed Papers

Software Process Subgroup of the Task Force on
Security across the Software Development Lifecycle

National Cyber Security Summit

March 2004

Edited by Samuel T. Redwine, Jr. and Noopur Davis

Processes to Produce Secure Software

Towards more Secure Software

Volume II

Contributed Papers

Software Process Subgroup of the Task Force on
Security across the Software Development Lifecycle
National Cyber Security Summit

March 2004

Edited by Samuel T. Redwine, Jr. and Noopur Davis

Foreword

The Software Process Subgroup within the Task Force on Security across the Software Development Lifecycle of the Cyber Security Summit – Co-Chaired by Sam Redwine (JMU), Geoff Shively (PivX), and Gerlinde Zibulski (SAP) – produced this report. The Task Force and its Subgroups were established December 2-3 in San Jose, California at the Cyber Security Summit sponsored by the Department of Homeland Security (DHS) and several industry groups. The three-month effort to produce this report, spanning December 2003 through February 2004, is part of the DHS-private sector partnership. The Subgroup's life should extend beyond the production of this report, but its specific activities may vary.

This Volume II of the report collects material supplied to the Subgroup.

Editors:

Samuel T. Redwine, Jr.
Noopur Davis

Contributors to Volume II:

Anthony Hall – Praxis Critical Systems
Roderick Chapman – Praxis Critical Systems
Noopur Davis – Software Engineering Institute
Joe Jarzombek – Information Assurance Directorate Office of the Assistant Secretary of Defense (Networks and Information Integration)
Richard C. Linger – Software Engineering Institute
Peter Neumann – SRI International
Stacy J. Prowell – University of Tennessee

The editors want to thank the contributors to Volume II for their efforts and for the many review comments they provided. In part, Sam Redwine's work was supported by Virgina's Commonwealth Technology Research Fund and the NIST's Critical Infrastructure Protection Project through the Institute for Infrastructure and Information Assurance (3IA) at James Madison University.

The Software Process Subgroup members are listed in the Foreword to Volume I.

Any corrections or comments to this report should be sent to Sam Redwine – redwinst@jmu.edu.

Table of Contents

Principles for Assuredly Trustworthy Composable Architectures:

February 20, 2004

Peter G. Neumann, Principal Investigator
Principal Scientist, Computer Science Laboratory
SRI International EL-243, 333 Ravenswood Ave
Menlo Park, California 94025-3493, USA
Neumann@csl.sri.com; http://www.csl.sri.com/neumann
Telephone: 1-650-859-2375; Fax: 1-650-859-2844

With the permission of the author, this document consists primarily of Chapter 2 (with some introductory material) from the emerging final report, *Principled Assuredly Trustworthy Composable Architectures,* for SRI Project 11459, Contract number N66001-01-C-8040, as part of DARPA's Composable High-Assurance Trustworthy Systems (CHATS) program, for which Douglas Maughan was the Program Manager. The draft text of the entire report is available on-line:
http://www.csl.sri.com/neumann/chats4.html as well as
http://www.csl.sri.com/neumann/chats4.pdf

Abstract

This report presents the results of our CHATS project. We characterize problems in and approaches to attaining computer system and network architectures with the overall goal of being better able to develop and more rapidly configure highly trustworthy systems and networks able to satisfy critical requirements (including security, reliability, survivability, performance, and other vital characteristics). We consider ways to enable effective systems to be predictably composed out of interoperable subsystems, to provide the required trustworthiness — with reasonably high assurance that the critical requirements will be met under the specified operational conditions, and (hopefully) do something sensible outside of that range of operational conditions. This work thus spans the entire set of goals of the DARPA CHATS program — trustworthiness, composability, and assurance — and much more.

By *trustworthiness,* we mean simply *worthy of being trusted to fulfill whatever critical requirements may be needed* for a particular component, subsystem, system, network, application, mission, enterprise, or other entity. Trustworthiness requirements might typically involve (for example) attributes of security, reliability, performance, and survivability under a wide range of potential adversities.

This report should be particularly valuable to system developers who have the need and/or the desire to build systems and networks that are significantly better than conventional mass-market software. The conclusions of the report will also be useful to government organizations that fund research and development efforts.

Executive Summary

We are confronting an extremely difficult problem — namely, how to attain demonstrably trustworthy systems and networks that need to operate under stringent requirements for security, reliability, survivability, and other critical attributes, and that can evolve gracefully and predictably over time – despite changes in requirements, hardware, and communications technologies. In particular, we seek to establish a sound basis for the creation of trustworthy systems and networks that can be easily composed out of subsystems and components, with predictably high assurance, and also hopefully do something sensible when forced to operate outside of the expected normal range of operational conditions. Toward this end, we examine a set of *principles* for achieving trustworthiness, consider *constraints* that might enhance composability, pursue *architectures* and *trustworthy subsystems* that are inherently likely to result in trustworthy systems and networks, constrain *administrative practices* in such a way that reduces the risks of bad operations, and seek approaches that can significantly increase *assurance.* The approach is intended to be theoretically sound as well as practical and realistic. We also outline directions for new research and development that could significantly improve the future for dependably trustworthy systems.

With respect to the future of trustworthy systems and networks, perhaps the most important recommendations involve the urgent establishment and use of realistic highly disciplined and principle-driven architectures, as well as development practices that systematically encompass trustworthiness and assurance as integral parts of what must become coherent development processes and sound subsequent operational practices. Only then can we have any realistic assurances that our computer-communication infrastructures — and indeed our critical national infrastructures — will be able to behave as needed, in times of crisis as well as in normal operation. This challenge does not have easy turn-the-crank solutions. Addressing it requires considerable skills, understanding, experience, education, and enlightened management. Success can be greatly increased in many ways, including the availability of dependable hardware components, robust system and network architectures, consistent use of good software engineering practices, careful attention to human-oriented interface design, well-conceived and sensibly used programming languages, compilers that are capable of enhancing the trustworthiness of source code, techniques for increasing interoperability among heterogeneous distributed systems and subsystems, methods and tools for analysis and assurance, design and development of systems that are inherently easier to administer and that provide better support for operational personnel, and many other factors. The absence or relative inadequacy with respect to each of these factors today represents a potential weak link in a process that is currently riddled with too many weak links. On the other hand,

much greater emphasis on these factors can result in substantially greater trustworthiness, with predictable results.

The approach taken here is strongly motivated by historical perspectives of promising research efforts and extensive development experience (both positive and negative) relating to the development of trustworthy systems. It is also motivated by the practical needs and limitations of commercial developments as well as some initial successes in inserting significantly greater discipline into the open-source world. It provides useful guidelines for disciplined system developments and future research.

As a consequence of the inherent complexity associated with the challenges of developing and operating trustworthy systems and networks, we urge you to read this report thoroughly, in its entirety. However, to the inexperienced developer or to the experienced developer who believes in seat-of-the-pants software creation, we offer a few words of caution. Many of the individual concepts should be well known to many of you. If you are looking for easy answers, you may be sadly disappointed; indeed, each chapter should in turn convince you that there are no easy answers. However, if you are looking for some practical advice on how to develop systems that are substantially more trustworthy than what is commercially available today, you may find many encouraging directions to pursue.

Although there are some novel concepts in this report, our main thrust involves various approaches that can make better use of what we have learned over the past many years in the research community and that can be used to better advantage in production systems. Many of the lessons relating to serious trustworthiness can be drawn from past research and prototype development; however, those lessons have been largely ignored in the commercial development communities. We believe that observance of the approaches described here would greatly improve the situation. The opportunities for this within the open-source community are considerable, although also applicable to closed-source proprietary systems (despite various caveats).

Chapter 1

The Foundations of This Report

We essay a difficult task; but there is no merit save in difficult tasks.
Ovid

In the context of this report, the term "trustworthy" is used in a broad sense that is meaningful with respect to any given set of requirements, policies, properties, or other definitional entities. Such requirements might include (for example) attributes of security, reliability, performance, and survivability under a wide range of potential adversities. **Security** requirements typically might specify properties relating to integrity, confidentiality, and ability to withstand denial of service attacks. **Reliability** requirements might include properties relating to the ability to tolerate hardware failures and software flaws, characterization of acceptable degradation in the face of untolerated faults, probabilities of success, expected mean times between failures, and so on. **Performance** requirements might include aggregate throughput measures, processing speeds, storage capacities, and guaranteed real-time response (for example). **Survivability** requirements address continued system availability despite numerous adversities that could compromise the intended goals, and thus (for example) encompass aspects of security, reliability, performance, and other relevant critical requirements (e.g., [25]). **Trustworthiness** can then be thought of as the hopefully well-founded belief that a given system, network, or component will satisfy its requirements, and particularly its critical requirements. **Assurance** provides some sort of measure or indication of the likelihood that the desired trustworthiness is actually well founded. That is, a system can be said to be trustworthy (with respect to its desired requirements) with some level of assurance that it will behave as expected.

The concept of trustworthiness is essentially indistingushable from what is alternatively called **dependability** [2, 3, 20, 34], particularly within the IEEE community. In its very early days, dependability was focused primarily on hardware faults and quickly extended to software faults, and soon thereafter generalized to a notion of faults that includes security threats. Thus, dependability's notions of fault prevention, fault tolerance, fault removal, and fault forecasting (the last of which in turn is more or less equivalent to assurance) seem to encompass everything that trustworthiness does, albeit with occasionally different terminology.

Note that we make a careful distinction throughout between *trust* and *trustworthiness*. **Trustworthiness** implies that something is worthy of being trusted. **Trust** merely implies

that you trust it whether it is trustworthy or not, perhaps because you have no alternative, or because you are naïve, or perhaps because you do not even realize that trustworthiness is necessary, or because of some other reason. We generally eschew the terms *trust* and *trusted* unless we specifically mean trust rather than trustworthiness.

There are many R&D directions that we believe are important for the short- and long-term future — for the computer and network communities at large, for DARPA developers and developers generally, for the CHATS program as a whole, and specifically for our CHATS project. The basis of our project is the exploration of a few of the potentially most timely and significant research directions, which are summarized as follows.

- **Principles.** We revisit fundamental principles of trustworthy system development, cull out those likely to be most effective, explore their practical limitations, and provide a basis for principled architectures, principled development, and principled operation.
- **Composability.** We explore existing obstacles to achieving seamless composability and techniques for attaining practical composability in the future. Composability is meaningful at many layers of abstraction, for components, subsystems, networked systems, and networks of networks. It is also applicable to policies, protocols, specifications, formal representations, and proofs. Subsystem composability takes on a variety of forms, including sequential (with or without feedback, with or without recursion, etc.) and parallel execution.
- **Trustworthy foundations.** We seek to provide a sound basis for specifications, implementation, trustworthiness, and assurance of that trustworthiness for composable interoperable components, with predictable behavior when composed.
- **Trustworthy composable architectures.** We seek to establish composable open distributed-system network-oriented architectures capable of fulfilling critical security, reliability, survivability, and performance requirements, while being readily adaptable to widely differing applications, different hardware and software providers, and changing technologies. By architecture, we specifically mean both the structure of systems and networks and the design of their functional interfaces (at various layers of abstraction).
- **Trustworthy protocols.** We need to develop new protocols and/or extend existing protocols that effectively mask the peculiarities of various networking technologies wherever possible, but able to accommodate a wide range of technologies (e.g., wireless and wired, optical and electronic, etc.), and capable of addressing all relevant critical requirements. This is a very difficult challenge, and necessarily needs the involvement of the IETF, NIST standards efforts, and the development communities.
- **Principled operational practice.** We need to bring the above concepts into the realm of operational practice, which is seriously in need of greater dependability and controllability. Many of the concepts considered here have considerable potential toward that end.

Throughout the history of efforts to develop trustworthy systems and networks, there is an unfortunate shortage of observable long-term progress. Significant research and development

results are typically soon forgotten or else widely ignored in practice. Systems have come and gone, programming languages have come and (sometimes) gone, and certain specific systemic vulnerabilities have come and gone. However, many generic classes of vulnerabilities seem to persist forever — such as buffer overflows, race conditions, off-by-one errors, mismatched types, divide-by-zero crashes, and unchecked procedure-call arguments, to name just a few. Overall, it is primarily only the principles that have remained inviolable — at least in principle — despite their having been widely ignored in practice. It is time to change that unfortunate situation, and honor the principles.

A paper [27] summarizing this report is part of the DISCEX3 proceedings, from the April 2003 DARPA Information Survivability Conference and Exposition.

Chapter 2

Fundamental Principles of Trustworthiness

In this chapter, we itemize, review, and interpret various design and development principles that if properly observed can advance composability, trustworthiness, assurance, and other attributes of systems and networks, within the context of the CHATS effort. We consider the relative applicability of those principles, as well as some of the problems they may introduce.

2.1 Introduction

> *Everything should be made as simple as possible — but no simpler.*
> Albert Einstein

A fundamental hypothesis motivating this report is that achieving assurable trustworthiness requires much greater observance of certain underlying principles. We assert that careful attention to such principles can greatly facilitate the following efforts.

- Establishment of composable open distributed-system network-oriented architectures capable of fulfilling critical security, reliability, survivability, and performance requirements, while being readily adaptable to widely differing applications, different hardware and software providers, and changing technologies. By architecture, we specifically mean both the structure of systems and networks and the design of their functional interfaces, at various layers of abstraction.
- Development of specifications, implementation, trustworthiness, and assurance of that trustworthiness for composable interoperable components, with predictable behavior when those components are composed.
- Attainment of assuredly trustworthy systems and networks, capable of addressing all relevant critical requirements, with new or extended protocols that mask the peculiarities of various networking technologies wherever advantageous.

The benefits of principled system and software development cannot be overestimated, especially in the early stages of the development cycle. Principled design and software

development can stave off many problems later on in implementation, maintenance, and operation. Huge potential cost savings can result from diligently observing relevant principles throughout the development cycle. But the primary concept involved is that of disciplined development; there are many methodologies that provide some kind of discipline, and all of those can be useful in some cases.

In concept, most of the principles discussed here are fairly well known and understood by system cognoscenti. However, their relevance is often not generally appreciated by people with little development or operational experience. Not wishing to preach to the choir, we do not dwell on elaborating the principles themselves. Instead, we concentrate on the importance and applicability of these principles in the development of systems with critical requirements — and especially secure systems and networks. The clear implication is that disciplined understanding and observance of the most effective of these principles can have enormous benefits to developers and system administrators, and also can aid user communities. However, we also explore various potential conflicts within and among these principles, and emphasize that those conflicts must be thoroughly understood and respected. System development is intrinsically complicated in the face of critical requirements. It is important to find ways to manage that complexity, rather than mistakenly believing that it is avoidable.

2.2 Risks Resulting from Untrustworthiness

As noted above, trustworthiness is a concept that encompasses being worthy of trust with respect to whatever critical requirements are in effect, often relating to security, reliability, guarantees of real-time performance and resource availability, survivability in spite of a wide range of adversities, and so on. Trustworthiness depends on hardware, software, communications media, power supplies, physical environments, and ultimately people in many capacities — requirements specifiers, designers, implementers, users, operators, maintenance personnel, administrators, and so on.

There are numerous examples of untrustworthy systems, networks, computer-related applications, and people. We indicate the extensive diversity of cases reported in the past with just a few tidbits relevant to each of various categories. See Computer-Related Risks [24] and the Illustrative Risks index [28] for numerous further examples and references involving many different types of system applications. (In the Illustrative Risks document, descriptors indicate relevance to loss of life, system survivability, security, privacy, development problems, human interface confusions, etc.)

- **Safety**
  - Aviation disasters, attributable to problems with airframes, avionics computer hardware and software, badly designed human interfaces, pilots, air-traffic control systems, air-traffic controllers, maintenance crews, airport security lapses, etc.: KAL 007 (flying on erroneous autopilot course), Air New Zealand crash into Mount Erebus (erroneous course data), Lauda Air (thrust reverser accidentally deployed in flight), Iranian Airbus shootdown (bad operational interfaces). Black Hawk helicopter problems.

  - Medical disasters, attributable to hardware flaws and malfunctions and software bugs, confusing human interfaces: Therac 25 (nonatomic transition from high-intensity to low-intensity mode), Database errors resulting in operation failures, Electromagnetic interference (pacemakers, defibrillators), Electrocution (e.g., a heart-monitoring equipment with a monitoring jack that plugged into an electrical wall socket).

- **Reliability and availability**

  - Failures in defense systems, control systems, telecommunications systems, space, financial systems, etc.: Patriot missiles missing Scuds (excessive clock drift), Yorktown Aegis missile cruiser disabled (Windows NT divide by zero), ARPANET collapse (1980),, AT&T long-distance collapse (1990), 1st Shuttle launch (Columbia backup computer synchronization problem), Discovery laser-beam experiment (elevation of target in miles, not feet), Massive power outages (propagating effects).

- **Security** (The situation here is truly deplorable and diverse. The Illustrative Risks index [28] includes many pages of reported security problems.)

  - Unintentional security flaws
  - Intentionally installed trapdoors, Trojan horses, etc.
  - Insider and outsider exploitations involving loss of confidentiality, loss of integrity, denials of service, viruses, worms, spam, financial frauds and misuse, etc.

- **Survivability**

  - Survivability ultimately depends on reliability, security, and various other attributes. (For example, see [25].) Some of the problems noted above involve failures of system and network survivability, as a result of hardware and software malfunctions, exploitations of security vulnerabilities, accidents, malice, electromagnetic interference and other environmental events, etc.

- **Privacy.**

  - Privacy is often relegated to a second-order consideration. Privacy can in some cases be aided by appropriate technology, but many of the misues are the result of misuse by trusted insiders or are extrinsic – involving indirect misuse external to computer systems. Identity theft is an increasingly pervasive example.

2.3 Trustworthiness Principles

Willpower is always more efficient than mechanical enforcement, when it works. But there is always a size of system beyond which willpower will be inadequate.
Butler Lampson

Developing and operating complex systems and networks with critical requirements demands a different kind of thinking from that used in routine programming. We begin here by considering various sets of principles, their applicability, and their limitations.

We first consider the historically significant Saltzer–Schroeder principles, followed by several other approaches.

2.3.1 Saltzer–Schroeder Security Principles, 1975

The ten basic security principles formulated by Saltzer and Schroeder [35] in 1975 are all still relevant today, in a wide range of circumstances. In essence, these principles are summarized with a CHATS-relevant paraphrased explanation, as follows:

- **Economy of mechanism:** Seek design simplicity (where effective).
- **Fail-safe defaults:** Deny accesses unless explicitly authorized (rather than permitting accesses unless explicitly denied).
- **Complete mediation:** Check every access, without exception.
- **Open design:** Do not assume that design secrecy will enhance security.
- **Separation of privilege:** Use separate privileges or even multiparty authorization (e.g., two keys) to reduce misplaced trust.
- **Least privilege:** Allocate minimal (separate) privileges according to need-to-know, need-to-modify, need-to-delete, need-to-use, and so on. Existence of overly powerful mechanisms such as *superuser* is inherently dangerous.
- **Least common mechanism:** Eschew sharing of trusted multipurpose mechanisms, in particular, minimizing the need for and use of overly powerful mechanisms such as *superuser*. As one example of the flaunting of this principle, exhaustion of shared resources provides a huge source of covert storage channels, whereas the natural sharing of real calendar-clock time provides a source of covert timing channels.
- **Psychological acceptability:** Strive for ease of use and operation — for example, with easily understandable and forgiving interfaces.
- **Work factor:** Make cost-to-protect commensurate with threats and expected risks.
- **Recording of compromises:** Provide nonbypassable tamperproof trails of evidence.

Remember that these are principles, not hard-and-fast rules. By no means should they be interpreted as ironclad, especially in light of some of their mutual contradictions.

The Saltzer–Schroeder principles grew directly out of the Multics experience (e.g., [32]). Each of these principles has taken on almost mythic proportions among the security elite, and to some extent buzzword cult status among many fringe parties. Therefore, perhaps it is not necessary to explain each principle in detail — although there is considerable depth of discussion underlying each principle. (Careful reading of the Saltzer–Schroeder paper [35] is recommended if it is not already a part of your library. In addition, Chapter 6 of Matt Curtin's book [11] on "developing trust" — by which he might really hope to be "developing trustworthiness" — provides some useful further discussion of these principles.)

There are two fundamental caveats regarding these principles. First, each principle by itself may be useful in some cases and not in others. The second is that when taken in combinations, groups of principles are not necessarily all reinforcing; indeed, they may seem to be mutually in conflict. Consequently, any sensible development must consider appropriate use of each principle in the context of the overall effort. Examples of a principle being both good and bad — as well as examples of interprinciple interference — are scattered through the following discussion. Various caveats are considered in the penultimate section.

Table 2.1 examines the applicability of each of the Saltzer–Schroeder principles to the CHATS goals of composability, trustworthiness, and assurance (particularly with respect to security, reliability, and other survivability-relevant requirements).

Table 2.1: CHATS Relevance of Saltzer–Schroeder to CHATS Goals

Principle	**Composability**	**Trustworthiness**	**Assurance**
Economy of mechanism	Beneficial within a sound architecture; requires great care	Important aid to sound design; requires great care	Can simplify analysis
Fail-safe defaults	Some help, but not fundamental	Simplifies design, use, operation	Can simplify analysis
Complete mediation	Very beneficial with disjoint object types	Vital, but hard to achieve with no compromisability	Can simplify analysis
Open design	Design documentation is very beneficial among multiple developers	Secrecy of design is, is bad assumption; open design requires strong system security	Assurance is mostly irrelevant in weak systems; open design enables open analysis (+/-)
Separation of privilege	Very beneficial if preserved by composition	Avoids many common flaws	Focuses analysis
Least privilege	Very beneficial if preserved by composition	Limits flaw effects; simplifies operation	Focuses analysis
Least common mechanism	Beneficial unless there is natural polymorphism	Finesses some common flaws	Simplifies analysis
Psychological acceptability	Could help a little — if nonconflicting	Affects mostly usability and operations	Ease of use can contribute
Work factor	Relevant especially for crypto algorithms, but not their implementations; may not be composable	Misguided if system easily compromised from below, spoofed, bypassed, etc.	Gives false sense of security under nonalgorithmic compromises
Compromise recording	Not an impediment if distributed	After-the-fact, but useful	Not primary contributor

In particular, complete mediation, separation of privilege, and allocation of least privilege are enormously helpful to composability and trustworthiness. Open design can contribute significantly to composability, when subjected to internal review and external criticism. However, there is considerable debate about the importance of open design with respect to trustworthiness, with some people still clinging tenaciously to the notion that security by obscurity is sensible — despite risks of many flaws being so obvious as to be easily detected externally, even without reverse engineering. Indeed, the recent emergence of very good decompilers for C and Java, and the likelihood of similar reverse engineering tools for other languages suggests that such attacks are becoming steadily more practical. Overall, the assumption of design secrecy and the supposed unavailability of source code is often not a deterrent, especially with ever-increasing skills among black-box system analysts. However,

there are of course cases in which security by obscurity is unavoidable — as in the hiding of private and secret cryptographic keys, even where the cryptographic algorithms and implementations are public. There are other cases in theory where weak links can be avoided (e.g., zero-knowledge protocols that can establish a shared key without any part of the protocol requiring secrecy), although in practice they may be undermined by compromises from below (e.g., involving trusted and supposedly trustworthy insiders subverting the underlying operating systems) or from outside (e.g., involving penetrations of the operating systems and masquerading as legitimate users). (See Section 2.3.)

From its beginning, the Multics development was strongly motivated by a set of principles — some of which were originally stated by Ted Glaser and Peter Neumann in the first section of the very first edition of the Multics System Programmers' Manual in 1965. For example, with almost no exceptions, no coding effort was begun until a written specification had been approved; with almost no exceptions, all code was written in a subset of PL/I for which a compiler (early PL, or EPL) had been written by Doug McIlroy and Bob Morris. In addition to the Saltzer–Schroeder principles, further insights can be found in a paper by Fernando Corbató, Saltzer, and Charlie Clingen [10] and in Corbató's Turing lecture [9].

2.3.2 Related Principles, 1969 and later

Another view of principled system development was given by Neumann in 1969 [22], relating to what is often dismissed as merely "motherhood" — but which in reality is both very profound and very difficult to follow consistently. The motherhood principles under consideration in that paper (alternatively, you might consider them just as desirable system attributes) included automatedness, availability, convenience, debuggability, documentedness, efficiency, evolvability, flexibility, forgivingness, generality, maintainability, modularity, monitorability, portability, reliability, simplicity, and uniformity. Some of those attributes indirectly affect security and trustworthiness, whereas others affect the acceptability, utility, and future life of the systems in question. Considerable discussion in [22] was also devoted to (1) the risks of local optimization and the need for a more global awareness of less obvious downstream costs of development (e.g., writing code for bad specs and debugging bad code), operation, and maintenance; and (2) the benefits of higher-level implementation languages (which prior to Multics were rarely used for the development of operating systems [9, 10]).

Fundamental to trustworthiness is the extent to which systems and networks can avoid being compromised by malicious or accidental human behavior and by events such as hardware malfunctions and so-called acts of God. In [25], we consider *compromise from outside, compromise from within,* and *compromise from below,* with fairly intuitive meanings. These notions appear throughout this report.

In later work and more recently in [25], Neumann augmented and refined some of the Saltzer–Schroeder principles. Although most of those principles might seem more or less obvious, they are of course full of interpretations and hidden issues. We summarize an extended set of principles here, particularly as they might be interpreted in the CHATS context.

- **Sound architecture.** Recognizing that it is much better to avoid design errors than to attempt to fix them later, the importance of architectures inherently capable of

evolvable, maintainable, robust implementations is enormous — even in an open-source environment. The value of a well-thought-out architecture is considerable in open-source systems. The value in closed-source proprietary systems could also be significant, if it were thought through early on, although architectural foresight is often impeded by legacy compatibility requirements that tend to lock into inflexible architectures. Good interface design is as fundamental to good architectures as is their structure. Both the architectural structure and the architectural interfaces (particularly the visible interfaces, but also some of the internal interfaces that must be interoperable) benefit from careful early specification.

- **Minimization of what must be trustworthy.** Trustworthiness should be situated where it is most needed, rather than widely distributed (with potentially many weak links) or centralized (with a single weak link). Trustworthiness is expensive to implement and to ensure, and as a consequence significant benefits can result from minimizing what has to be trustworthy. This principle can contribute notably to sound architectures.

- **Abstraction.** The primitives at any given logical or physical layer should be relevant to the functions and properties of the objects at that layer, and should mask lower-layer detail where possible. Ideally, the specification of a given abstraction should be in terms of objects meaningful at that layer, rather than requiring lower-layer (e.g., machine dependent) concepts. Abstractions at one layer can be related to the abstractions at other layers in a variety of ways, thus simplifying the abstractions at each layer rather than collapsing different abstractions into a more complex single layer. Horizontal and vertical abstractions are considered in Chapter 3.

- **Encapsulation.** Details that are relevant to a particular abstraction should be isolated within the implementation of that abstraction and the lower layers on which the implementation depends. One example of encapsulation involves information hiding — for example, keeping internal state information hidden. Another example involves masking the idiosyncrasies of physical devices from the user interface, as well as from higher-layer system interfaces.

- **Modularity.** Modularity relates to the characteristic of system structures in which different entities (modules) can be relatively loosely coupled and combined to satisfy overall system requirements, whereby a module could be modified or replaced as long as the new version satisfies the given interface specification. In general, modularity is most effective when the modules reflect specific abstractions and provide encapsulation within each module.

- **Layered and distributed protection.** Protection should be distributed to where it is most needed, and should reflect the semantics of the objects being protected. With respect to the reality of implementations that transit entities of different trustworthiness, layers of protection are vastly preferable to flat concepts such as single sign-on. With respect to psychological acceptability, single sign-on has enormous appeal — even if it can leave enormous security vulnerabilities as a result of compromise from outside,

from within, or from below in both distributed and layered environments. (Of particular relevance here are work in distributed system protection and digital certificates such as SDSI/SPKI, and digital rights management (e.g., [7, 17, 38]).)

- **Constrained dependency.** Unguarded dependencies on less trustworthy entities should be avoided. However, it is possible in some cases to surmount the relative untrustworthiness of mechanisms on which certain functionality depends — as in the types of trustworthiness-enhancing mechanisms enumerated in Chapter 3. In essence, do not trust anything unless you are satisfied with demonstrations of its trustworthiness.

- **Object orientation.** The OO paradigm bundles together abstraction, encapsulation, modularity of state information, inheritance (subclasses inheriting the attributes of their parent classes — e.g., for functionality and for protection), and subtype polymorphism (subtype safety despite the possibility of application to objects of different types). This paradigm facilitates programming generality and software reusability, and if properly used can enhance software development. This is a contentious topic, in that most of the OO methodologies and languages are somewhat sloppy with respect to inheritance. (Jim Horning notes that the only OO language he knows that takes inheritance seriously was the DEC/ESL OWL/Trellis, which was a descendant of CLU.)

- **Separation of policy & mechanism.** Statements of policy should avoid inclusion of implementation-specific details. Furthermore, mechanisms should be policy-neutral where that is advantageous in achieving functional generality. However, this principle must never be used in the absence of understanding about the range of policies that might be usefully implemented. There is a temptation to avoid defining meaningful policies, deferring them until later in the development — and then discovering that the desired policies cannot be realized with the given mechanisms. This is a characteristic chicken-and-egg problem with abstraction.

- **Separation of duties.** In relation to separation of privilege, separate classes of duties of users and computational entities should be identified, so that distinct system roles can be assigned accordingly. Distinct duties should be treated distinctly, as in system administrators, system programmers, and unprivileged users.

- **Separation of roles.** In relation to separation of privilege, the roles recognized by protection mechanisms should correspond to the various duties. For example, a single all-powerful superuser is intrinsically in violation of separation of duties, separation of roles, separation of privilege, and separation of domains. The separation of would-be superuser functions into separate roles as in Trusted Xenix is a good example of desirable separation. Again there is a conflict between principles: the monolithic superuser mechanism provides economy of mechanism, but violates other principles. Similarly, the notion of a single sign-on provides simplicity for the user, but seriously violates least privilege, separation of concerns, and other principles. In practice, all-powerful mechanisms are sometimes unavoidable, and sometimes even desirable despite the negative consequences (particularly if confined to a secure sub-environment). However, they should be avoided wherever possible.

- **Separation of domains.** In relation with separation of privilege, domains should be able to enforce separate roles. For example, a single all-powerful superuser mechanism is inherently unwise, and is in conflict with the notion of separation of privileges. However, separation of privileges is difficult to implement if there is inadequate separation of domains. Separation of domains can help enforce separation of privilege, but can also provide functional separation as in the Multics ring structure, a kernelized operating system, or a capability-based architecture.
- **Sound authentication.** Authentication is a pervasive problem. Nonbypassable authentication should be applicable to users, processes, procedures, and in general to any active entity or object. Authentication relates to evidence that the identity of an entity is genuine, that procedure arguments are legitimate, that types are properly matched when strong typing is to be invoked, and other similar aspects.
- **Sound authorization and access control.** Authorizations must be correctly and appropriately allocated, and nonsubvertible (although they are likely to assume that the identities of all entities and objects involved have been properly authenticated — see Sound authentication). Crude all-or-nothing authorizations are typically inadequate. In applications for which user-group-world authorizations are inadequate, access-control lists and role-based authorizations may be preferable. Finer-grained access controls may be desirable in some cases, such as capability-based addressing and field-based database protection.
- **Administrative controllability.** The facilities by which systems and networks are administered must be well designed, understandable, well documented, and sufficiently easy to use without inordinate risks.
- **Comprehensive accountability.** Well designed and carefully implemented facilities are essential for comprehensive monitoring, auditing, interpretation, and automated response (as appropriate). Serious security and privacy issues must be addressed relating to the overall accountability processes and audit data.

Table 2.2 summarizes the utility of the extended-set principles with respect to the three goals of the CHATS program acronym, as in Table 2.1.

Table 2.2: CHATS Relevance of Extended-Set Principles to CHATS Goals

Principle	**Composability**	**Trustworthiness**	**Assurance**
Sound architecture	Can considerably facilitate composition	Can greatly increase trustworthiness	Can increase assurance of design and simplify implementation analysis
Minimization of trustworthiness	Beneficial, but not fundamental	Very beneficial with sound architecture	Simplifies design and implementation analysis
Abstraction	Very beneficial with suitable independence	Very beneficial if composable	Simplifies analysis by decoupling it
Encapsulation	Very beneficial if properly done, enhances integration	Very beneficial if composable, avoids certain types of bugs	Localizes analysis to abstractions and their interactions
Modularity	Very beneficial if interfaces and specifications well defined	Very beneficial if well specified; overmodularization impairs performance	Simplifies analysis by decoupling it and if modules are well specified
Layered protection	Very beneficial, but may impair performance	Very beneficial if noncompromisible from above/within/below	Structures analysis according to layers and their interactions
Robust dependency	Beneficial: can avoid compositional conflicts	Beneficial: can obviate design flaws based on misplaced trust	Robust architectural structure simplifies analysis
Object orientation	Beneficial, but labor-intensive; can be inefficient	Can be beneficial, but complicates coding and debugging	Can simplify analysis of design, possibly implementation also
Separation of policy & mechanism	Beneficial, but both must compose	Increases flexibility and evolution	Simplifies analysis
Separation of duties	Helpful indirectly as a precursor	Beneficial if well defined	Can simplify analysis if well defined
Separation of roles	Beneficial if roles nonoverlapping	Beneficial if properly enforced	Partitions analysis of design and operation
Separation of domains	Can simplify composition and reduce side-effects	Allows finer-grain enforcement and self-protection	Partitions analysis of implementation and operation
Sound authentication	Helps if uniformly invoked	Huge security benefits, aids accountability	Can simplify analysis, improve assurance
Sound authorization	Helps if uniformly invoked	Controls use, aids accountability	Can simplify analysis, improve assurance
Administrative controllability	Composability helps controllability	Good architecture helps controllability	Control enhances operational assurance
Comprehensive accountability	Composability helps accountability	Beneficial for post-hoc analysis	Can provide feedback for improved assurance

At this point in our analysis, it should be no surprise that all of these principles can contribute in varying ways to security, reliability, survivability, and other -ilities. Furthermore, many of the principles and -ilities are linked. We cite just a few of the interdependencies that must be considered.

For example, authorization is of limited use without authentication when identity is important. Similarly, authentication may be of questionable use without authorization. In some cases, authorization requires fine-grained access controls. Least privilege requires some sort of separation of roles, duties, and domains. Separation of duties is difficult to achieve if there is no separation of roles. Separation of roles, duties, and domains each must rely on a supporting architecture.

The Saltzer–Schroeder comprehensive accountability principle is particularly intricate, as it depends critically on many other principles being invoked. For example, accountability is inherently incomplete without authentication and authorization. In many cases, monitoring may be in conflict with privacy requirements and other social considerations [12], unless extremely stringent controls are enforceable. Separation of duties and least privilege are particularly important here. All accountability procedures are subject to security attacks, and are typically prone to covert channels as well. Furthermore, the procedures themselves should be carefully monitored. Who monitors the monitors? *(Quis auditiet ipsos audites?)*

2.3.3 Principles of Secure Design (NSA, 1993)

Also of interest here is the 1993 set of principles (or perhaps metaprinciples?) of secure design [5], which emerged from an NSA ISSO INFOSEC Systems Engineering study on rules of system composition. The study was presented not as a finished effort, but rather as something that needed to stand the test of practice. Although there is some overlap with the previously noted principles, the NSA principles are enumerated here as they were originally documented. Some of these principles are equivalent to "the system should satisfy certain security requirements" — but they are nevertheless relevant. Others might sound like motherhood. Overall, they represent some collective wisdom.

- The security engineering of a system must not be done independently from the total engineering of the system.
- A system without requirements cannot fail; it merely presents surprises.
- The system is for the users and not the system designers.
- A systems seldom fully satisfies all of its requirements.
- Many failures of a system to meet its overall requirements are often obvious. However, failures to meet security requirements are often not obvious.
- In an operational system, it is the users' mission and information that is at risk, not the developers' or evaluators' information. The accreditor accepts those risks when deciding to use a system operationally.
- It is only in the context of a system and a security policy that the "security characteristics" of a component can be defined and evaluated.

- Every component in a system must operate in an environment that is a subset of its specified environment; [in particular,] every component in a system must operate in a security environment that is a subset of its specified security environment. (A component should not be asked to respond to events for which it was not designed — and evaluated.)
- Security is a system problem.
- Keep it simple to make it secure.
- There is no security in uncertainty.
- A system should be evaluatable and evaluated.
- Architectural analysis should not be treated lightly.
- A system is only as strong as its weakest link; the fortress walls of security should all be high enough. (Note that weak links are often not obvious.)
- A component should protect itself from other components by adhering to the principle of mutual suspicion.
- A system should be manageable and managed.
- A system should be able to come up in a recognizably secure state.
- A system should recognize error conditions.
- Pay special attention to information flow.
- Secure systems should protect the confidentiality of user data.
- Secure systems should protect the integrity of user data.
- Secure systems should protect the reliability of user processes.

Considerable discussion of these metaprinciples is warranted. For example, "Every component in a system must operate in a security environment that is a subset of its specified environment" implies iteratively that maximum trust is required throughout design and implementation of the other components, which is a gross violation of our notion of minimization of what must be trustworthy. It would be preferable to require that each component check that the environment in which it executes is a subset of its specified environment — which is closely related to Schroeder's notion of mutual suspicion [36], noted further down the list.

"A system is only as strong as its weakest link" is generally a meaningful statement. However, some weak links may be more devastating than others, so this statement is overly simple. In combination with least privilege, separation of domains, and some of the other principles noted previously, the effects of a particular weak link might be contained or controlled. But then, you might say, the weak link was not really a weak link. However, to a first approximation, as we noted above, weak links should be avoided where possible, and restricted in their effects otherwise, through sound architecture and sound implementation practice.

2.3.4 Generally Accepted Systems Security Principles (I^2F, 1997)

The 1990 report of the National Research Council study group that produced *Computers at Risk* [8] included a recommendation that a serious effort be made to develop and promulgate a set of Generally Accepted Systems Security Principles (GASSP). That led to the creation of the International Information Security Foundation (I^2SF). A draft of its GASSP document [33] is available on-line. A successor effort is now underway, after a long pause.

The proposed GASSP consists of three layers of abstraction, nine Pervasive Principles (relating to confidentiality, integrity, and availability), a set of 14 Broad Functional Principles, and a set of Detailed Principles (yet to be developed, because the largely volunteer project ran out of steam, in what Jim Horning refers to as a last gassp!). The GASSP effort thus far actually represents a very worthy beginning, and one more approach for those interested in future efforts. The top two layers of the GASSP principle hierarchy are summarized here as follows.

Pervasive Principles
- PP-1. **Accountability**
- PP-2. **Awareness**
- PP-3. **Ethics**
- PP-4. **Multidisciplinary**
- PP-5. **Proportionality**
- PP-6. **Integration**
- PP-7. **Timeliness**
- PP-8. **Assessment**
- PP-9. **Equity**

Broad Functional Principles
- BFP-1. **Information Security**
- BFP-2. **Education and Awareness**
- BFP-3. **Accountability**
- BFP-4. **Information Management**
- BFP-5. **Environmental Management**
- BFP-6. **Personnel Qualifications**
- BFP-7. **System Integrity**
- BFP-8. **Information Systems Life Cycle**
- BFP-9. **Access Control**
- BFP-10. **Operational Continuity and Contingency Planning**
- BFP-11. **Information Risk Management**
- BFP-12. **Network and Infrastructure Security**
- BFP-13. **Legal, Regulatory, and Contractual Requirements of Info Security**
- BFP-14. **Ethical Practices**

The GASSP document gives a table showing the relationships between the 14 Broad Functional Principles and the 9 Pervasive Principles. That table is reproduced here as Table 2.3.

Table 2.3: GASSP Cross-Impact Matrix

PP:	PP-1	PP-2	PP-3	PP-4	PP-5	PP-6	PP-7	PP-8	PP-9
BFP-1	X	X	X	X	X	X	X	X	X
BFP-2	X	X	X	X					X
BFP-3	X	X	X	X					X
BFP-4	X	X		X				X	
BFP-5	X	X	X	X	X			X	
BFP-6	X	X		X					X
BFP-7	X			X	X	X	X	X	
BFP-8	X			X	X	X	X	X	
BFP-9	X			X	X	X	X	X	
BFP-10	X			X	X	X		X	
BFP-11	X	X		X	X	X	X	X	
BFP-12	X			X	X		X	X	
BFP-13	X	X	X	X					X
BFP-14		X	X	X					X

2.3.5 TCSEC, ITSEC, CTCPEC, and the Common Criteria (1985 to date)

Any enumeration of relevant principles must note the historical evolution of evaluation criteria over the past decades — from the 1985 DoD Trusted Computer System Evaluation Criteria (TCSEC, a.k.a. The Orange Book [21]) and the ensuing Rainbow Books, to the 1990 Canadian Trusted Computer Product Evaluation Criteria (CTCPEC, [6]), and the 1991 Information Technology Security Evaluation Criteria (ITSEC, [13]). These efforts have resulted in an international effort to produce the Common Criteria framework (ISO 15408 [18]), which represents the current state of the art in that particular evolutionary process. (Applicability to multilevel security is also addressed within the Common Criteria framework, although it is much more fundamental to the TCSEC.)

2.3.6 Extreme Programming, 1999

A seemingly radical approach to software development is found in the Extreme Programming (XP) movement [4]. (Its use of "XP" considerably predates Microsoft's.) Although XP appears to run counter to most conventional programming practices, it is indeed highly disciplined. XP might be thought of as very small chief programmer teams somewhat in the spirit of a Harlan Mills' Clean-Room approach, although it has no traces of formalism and is termed a *lightweight methodology.* It involves considerable emphasis on disciplined planning (documented user stories, scheduling of relatively frequent small releases, extensive iteration planning, and quickly fixing XP whenever necessary), designing (with simplicity as a driving force, the selection of a system metaphor, and continual iteration), coding (paired programmers working closely together, continual close coordination with the customer, adherence to agreed-upon standards, only one programmer pair may integrate at one time,

frequent integration, deferred optimization, and no overtime pay), and testing (code must pass unit tests before release, tests must be created for each bug found, acceptance tests are run often, and the results are published). Questions of how to address architecture in the large seem not to be adequately addressed within Extreme Programming (although they are absolutely fundamental to the approach that we are taking in our CHATS project, but perhaps are considered extraneous to XP). See the Web site noted in [4] for considerable background on the XP movement, including a remarkably lucid Frequently Asked Questions document contrasting XP with several other approaches (UML, RUP, CMM, Scrum, and FDD) — although this is a little like comparing apples and oranges.

2.3.7 Other Approaches to Principled Development

Of course, there are too many other design and development methodologies to enumerate here, ranging from very simple to quite elaborate. In some sense, it does not matter which methodology is adopted, as long as it provides some structure and discipline, and is relatively compatible with the abilities of the particular design and development team. For example, Dick Karpinski hands out a business card containing his favorite, Tom Gilb's Project Management Rules: (1) Manage critical goals by defining direct measures and specific targets; (2) Assure accuracy and quality with systematic project document inspections; (3) Control major risks by limiting the size of each testable delivery. These are nice goals, but depend on the skills and experience of the developers — with only subjective evaluation criteria. Harlan Mills' "Clean-Room" technology has some elements of formalism that are of interest with respect to increasing assurance, although not specifically oriented toward security. In general, good development practice is a necessary prerequisite for trustworthy systems, as are means for evaluating that practice.

2.4 Types of Design and Implementation Flaws, and Their Avoidance

Nothing is as simple as we hope it will be. Jim Horning

Some characteristic sources of security flaws in system design and implementation are noted in [24], elaborating on earlier formulations and refinements (e.g., [1, 31]). There are various techniques for avoiding those flaws, including defensively oriented programming languages, defensively oriented compilers, better run-time environments, and generally better software engineering practice.

- **Identification and authentication.** The lack of nonspoofable identities and peer-to-peer authentication within user systems and network infrastructures is a huge obstacle to the robust networking of systems and prevents traceback to identify misuse — assuming that the misuse can be detected. The pervasive use of fixed/reusable passwords (especially those that traverse networks unencrypted or are otherwise exposed) is also a high-risk problem. Elaborate schemes for managing these passwords (such as avoiding dictionary words) ignore many of the risks. An enormous improvement can be achieved

by using one-time authenticators such as cryptographic tokens, and — in certain constrained user environments — biometrics, at least within supposedly trustworthy subsystems and subnetworks. The pervasive use of unauthenticated IP addresses that are easily spoofed is another area of risk. Remote sites and remote users are frequently not properly identified and authenticated. Meaningful authentication is a precursor to the avoidance or restriction of many types of misuse.

- **Authorization.** Our systems and networks suffer from a serious lack of context-sensitive authorization. Monolithic access controls tend to grant all-or-nothing or extremely coarse permissions. The development and consistent use of finer-grained authorization techniques would be very helpful in enforcing separation of privilege and least privilege. In the classified world, gross levels (e.g., Top Secret, Secret, Confidential, and Unclassified) are clearly too inclusive, which is why finer-grained compartments are invoked.

- **Initialization and allocation.** Failures in the initialization of procedures, processes, and indeed stable system and network configuration management represent a large class of system flaws. Consistency checking on entry, determination of suitable availability of appropriate resources, and deletion of possible residues are examples of techniques that can provide improved initialization and allocation.

- **Finalization.** In most programming languages, the lack of graceful termination and complete deallocation is inadequately recognized as a source of flaws. For example, deletion of leftover residues from previous executions is often ignored or relegated to an initialization problem, rather than treated systematically on termination (perhaps on the grounds that it might be avoided altogether in some circumstances). In general, finalization should be symmetrically matched with initialization. Whatever is done in initialization may need to be explicitly undone or at least checked for consistent status at finalization. Programming languages that incorporate garbage collection (GC) attempt to do this implicitly, although not always perfectly. For example, note that Java's finalizers based on pointer unreachability are inherently imprecise. Various other GC-based languages have subtle finalization problems, as do non-GC-based programming languages. Overall, the need for secure and robust finalization remains a research topic,

- **Run-time validation.** A large class of flaws results from inadequate run-time validation. Careful attention to techniques such as argument validation and bounds checks (especially to prevent insertion of Trojan horses such as executables added to arguments, causing buffer overflows), divide-by-zero checks, and strong typing of arguments can have enormous benefits. Brian Randell long ago suggested the benefits of moving checking closer to the operations being performed (whether in space, in time, or in layer of abstraction), to reduce the intervening infrastructure that must be trustworthy. This is also applicable to end-to-end checks and end-to-end security.

- **Consistent naming.** Aliases, pointers, links, caches, and dynamic changes without relinking, and other potentially multiple representations all represent common sources of security vulnerabilities. Symmetric treatment of aliases, symbolic naming and dynamic

linking, use of globally unique names, and recognition of stale caches and cache clearing are examples of beneficial techniques.

- **Encapsulation.** Exposure of procedure and process internals may allow leakage of supposedly protected information or externally induced interference. Proper encapsulation requires a combination of system architecture, programming language design, software engineering, static checking, and dynamic checking.

- **Asynchronous consistency.** Many vulnerabilities arise as a result of timing and sequencing, such as order dependencies, race conditions, synchronization, and deadlocks. Note that many of these problems arise because of sharing of state information (particularly in real time or in sequential ordering) across abstractions that otherwise seem disjoint. Atomic transactions, multiphase commits, and hierarchical locking strategies are examples of constructive design techniques.

- **Other logic errors.** There are also many common logic errors (such as off-by-one counting, omitted negations, or absolute values) that need to be avoided. Many of these arise in the design process, but some involve bad implementation. Useful techniques for detecting some of these errors include defensive programming language design, compiler checks, and formal methods analyzing consistency of programs with specifications. Of particular recent interest is the use of static checking. Such an approach may be formally based, as in the use of model checking by Hao Chen, Dave Wagner, and Drew Dean (as part of our CHATS project). Alternatively, there are numerous approaches that do not use formal methods, ranging in sophistication from `lint` to `LCLint` (Evans) to Extended Static Checking (Nelson, Reino, et al., DEC/Compaq/SRC). Jim Horning notes that even partial specifications increase the power of the latter two, and provide a relatively gentle way to incorporate additional formalism into development. However, it is worth noting that strong type checking and model checking tend to expose various errors that are inconsequential, particularly with respect to security and reliability. Purify and similar tools are useful in catching memory leaks, array-bound violations, and related memory problems. Nevertheless, these and other analytic techniques can be very helpful in improving design soundness and code quality — as long as they are not relied on by themselves as silver bullets.

All of the principles can have some bearing on avoiding these classes of vulnerabilities.

Several of these concepts in combination — notably modularity, abstraction, encapsulation, device independence where advantageous, information hiding, complete mediation, separation of policy and mechanism, separation of privilege, least privilege, and least common mechanism — are relevant to the notion of virtual interfaces and virtual machines. The basic notion of virtualization is that it masks many of the underlying details, and makes it possible to change the implementation without changing the interface. In this respect, several of these attributes are found in the object-oriented paradigm.

Several examples of virtual mechanisms and virtualized interfaces are worth noting. Virtual memory masks physical memory locations and paging. A virtual machine masks the representation of process state information and processor multiplexing. Virtualized input-output masks device multiplexing, device dependence, formatting, and timing. Virtual mul-

tiprocessing masks the scheduling of tasks within a collection of seemingly simultaneous processes. The Multics operating system [32] provides an illustration of virtual memory and virtual secondary storage management (with demand paging hidden from the programs), virtualized input-output (with symbolic stream names and device independence where commonalities exist), and virtual multiprogramming (with scheduling typically hidden from the programming interfaces). The GLU environment [19] is an elegant illustration of virtual multiprocessing (which allows programs to be distributed among different processing resources without explicit processor allocation).

2.5 Roles of Assurance and Formalism

In principle, everything should be simple.
In reality, things are typically not so simple.
(Note: The SRI CSL Principal Scientist is also a Principle Scientist. PGN)

In general, the task of providing some meaningful assurance that a system is likely to do what is expected of it can be enhanced by any techniques that simplify or narrow the analysis — for example, by increasing the discipline applied to system architecture, software design, specifications, code style, and configuration management. Most of the cited principles tend to do exactly that — if they are applied wisely.

Techniques for increasing assurance are considered in greater detail in Chapter 5, including the potential roles of formal methods.

2.6 Caveats on Applying the Principles

For every complex problem, there is a simple solution. And it's always wrong.
H.L. Mencken

As we noted above, the principles referred to here may be in conflict with one another if each is applied independently, and are themselves not simply composable. Consequently, each principle must be applied in the context of the overall development, and we need to expend considerable effort to reformulate the principles to make them more readily composable.

There are also various potentially harmful considerations that must be considered — for example, overuse, underuse, or misapplication of these principles, and certain limitations inherent in the principles themselves. Merely paying lipservice to a principle is clearly a bad idea; principles must be consistently applied to the extent that they are appropriate to the given purpose. Similarly, all of the criteria-based methodologies have many systemic limitations (e.g., [23, 37]); for example, formulaic application of evaluation criteria is always subject to incompleteness and misinterpretation of requirements, oversimplification in analysis, and sloppy evaluations. However, when carefully applied, such methodologies can be useful and add discipline to the development process. Thus, we stress here the importance of fully understanding the given requirements and of creating an overall architecture that is appropriate for realizing those requirements, before trying to conduct any assessments of

compliance with principles or criteria. There is absolutely no substitute for human intelligence, experience, and foresight.

The Saltzer–Schroeder principle of keeping things simple is one of the most popular and commonly cited. However, it can be extremely misleading when espoused (as it commonly is) in reference to systems with critical requirements for security, reliability, survivability, real-time performance, and high assurance — especially when all of these requirements are necessary within the same system environment. Simplicity is a very important concept in principle (in the small), but complexity is often unavoidable in practice (in the large). For example, serious attempts to achieve fault-tolerant behavior often result in at least doubling the size of the overall system. As a result, the principle of simplicity should really be one of managing complexity rather than trying to eliminate it, particularly where complexity is in fact inherent in the combination of requirements. Keeping it simple is indeed a wonderful principle, but often difficult in reality. Nevertheless, *unnecessary* complexity should of course be avoided. The back-side of the Einstein quote at the beginning of Section 2.1 is indeed both profound and relevant, yet often overlooked in the overzealous quest for perceived simplicity.

An extremely effective approach to dealing with intrinsic complexity is through a combination of the principles discussed here, particularly abstraction, modularity, encapsulation, and careful hierarchical separation that architecturally does not result in serious performance penalties, well conceived virtualized interfaces that greatly facilitate implementation evolution without requiring changes to the interfaces or that enable design evolution with minimal disruption, and nonlocal optimization. In particular, hierarchical abstraction can result in relative simplicity at the interfaces of each abstraction and each layer, in relative simplicity of the interconnections, and perhaps even relative simplicity in the implementation of each module. By keeping the components and their interconnections conceptually simple, it is possible to achieve conceptual simplicity of the overall system or networks of systems despite inherent complexity. Furthermore, simplicity can sometimes be achieved through design generality, recognizing that several seemingly different problems can be solved symmetrically at the same time, rather than creating different (and perhaps incompatible) solutions. Note that such solutions might appear to be a violation of the principle of least common mechanism, but not when the common mechanism is fundamental — as in the use of a single uniform naming convention or the use of a uniform addressing mode that transcends different subtypes of typed objects. In general, it is riskful to have multiple procedures managing the same data structure for the same purposes. However, it can be very beneficial to separate reading from writing — as in the case of one process that updates and another process that uses the data. It can also be beneficial to reuse the same code on different data structures, although strong typing is then important.

One of our primary goals in this project is to make system interfaces simple while masking complexity so that the complexities of the design process and the implementation itself can be hidden by the interfaces. This may in fact increase the complexity of the design process, the architecture, and the implementation. However, the resulting system complexity need be no greater than that required to satisfy the critical requirements such as for security, reliability, and survivability. It is essential that tendencies toward bloatware be strongly resisted. (They seem to arise largely from the desire for bells and whistles — extra features — and fancy graphics.)

A networking example of the constructive use of highly principled hierarchical abstraction is given by the protocol layers of TCP/IP. An operating system example is given by the capability-based Provably Secure Operating System (PSOS) [14, 29, 30]) in which the functionality at each of more than a dozen layers was specified formally in only a few pages each, with at least the bottom 6 layers intended to be implemented in hardware. The underlying addressing is based on a capability mechanism that uniformly encompasses and protects objects of arbitrary types — including files, directories, processes, and other system- and user-defined types. The PSOS design is particularly noteworthy because a single capability-based operation at layer 12 (user processes) could be executed as a single machine instruction at layer 6 (system processes), with no iterative interpretation required unless there were missing pages or unlinked files that require operating system intervention (e.g., for dynamic linking of symbolic names, à la Multics). To many people, hierarchical layering instantly brings to mind inefficiency. However, the PSOS architecture is an example in which the hierarchical design could be implemented extremely efficiently — because of the architecture.

We note that formalism for its own sake is generally counterproductive. Formal methods are not likely to reduce the overall cost of software development, but can be helpful in decreasing the cost of software quality and assurance. They can be very effective in carefully chosen applications, such as evaluation of requirements, specifications, critical algorithms, and particularly critical code. Once again, we should be optimizing not just the cost of writing and debugging code, but rather optimizing more broadly over the life cycle.

There are many other common pitfalls that can result from the unprincipled use of principles. Blind acceptance of a set of principles without understanding their implications is clearly inappropriate. (Blind rejection of principles is also observed occasionally, particularly among people who establish firm requirements with no understanding of whether those requirements are realistically implementable — and among strong-willed developers with a serious lack of foresight.)

Lack of discipline is clearly inappropriate in design and development. For example, we have noted elsewhere [25, 26] that the open-source paradigm by itself is not likely to produce secure, reliable, survivable systems in the absence of considerable discipline throughout development, operation, and maintenance. However, with such discipline, there can be many benefits. (See also [16] on the many meanings of "open source" and a Newcastle Dependable Interdisciplinary Research Collaboration (DIRC) final report [15] on dependability issues in open source, part of ongoing work.)

Any principle can typically be carried too far. For example, excessive abstraction can result in overmodularization, with enormous overhead resulting from intermodule communication and nonlocal control flow. On the other hand, conceptual abstraction through modularization that provides appropriate isolation and separation can sometimes be collapsed (e.g., for efficiency reasons) in the implementation — as long as the essential protection boundaries are not undermined. Thus, modularity should be considered where it is advantageous, and not otherwise.

Application of each principle is typically somewhat context dependent, and in particular dependent on the specific architecture. In general, principles should always be applied relative to the integrity of the architecture.

One of the severest risks in system development involves local optimization with respect to components or individual functions, rather than global optimization over the entire architecture, its implementation, and its operational characteristics. Radically different conclusions can be reached depending on whether or not you consider the long-term complexities and costs introduced by bad design, sloppy implementation, increased maintenance necessitated by hundreds of patches, incompatibilities between upgrades, noninteroperability among different components with or without upgrades, and general lack of foresight. Furthermore, unwise optimization (local or global) must not collapse abstraction boundaries that are essential for security or reliability — perhaps in the name of improved performance. As one example, real-time checks (bounds checks, argument validation, etc.) should be kept close to the operations involved, for obvious reasons.

Perhaps most insidious is the *a priori* lack of attention to critical requirements, such as any that might involve the motherhood attributes noted in [22] and listed above. Particularly in dealing with security, reliability, and survivability in the face of arbitrary adversities, there are few if any easy answers. But if those requirements are not dealt with from the beginning of a development, they can be extremely difficult to retrofit later. One particularly appealing survivability requirement would be that systems and networks should be able to reboot, reconfigure, and revalidate their soundness following arbitrary outages, without human intervention. That requirement has numerous architectural implications that are considered in Chapter 4.

Once again, everything should be made as simple as possible, but no simpler. Careful adherence to principles that are deemed effective is likely to help achieve that goal.

2.7 Summary

> *In theory, there is no difference between theory and practice. In practice, there is an enormous difference.* (Many variants of this concept are attributed to various people. This is my own adaptation.)

What would be extremely desirable in our quest for trustworthy systems and networks is theory that is practical and practice that is sufficiently theoretical. We firmly believe that thoughtful and judiciously applied adherence to sensible principles that are appropriate for any particular development can greatly enhance the security, reliability, and overall survivability of the resulting systems and networks. These principles can also contribute greatly to operational interoperability, maintainability, operational flexibility, long-term evolvability, higher assurance, and many other desirable characteristics.

To illustrate some of these concepts, we have given a few examples of systems and system components whose design and implementation are strongly principled. The omission of other examples does not in any way imply that they are less relevant. We have also given some examples of just a few of the potential difficulties in trying to apply these principles.

Please remember that the supposedly best practices can be manhandled (or womanhandled) by very good programmers, and that bad programming languages can still be used wisely. There are no easy answers. However, having sensible system and network architectures is generally a good starting point, as discussed in Chapter 4, where we specifically

consider classes of system and network architectures that are consistent with the principles noted here, and that are highly likely to be effective in fulfilling the CHATS goals. In particular, we seek to approach inherently complex problems architecturally, structuring the solutions to those problems as conceptually simple compositions of relatively simple components, with emphasis on the predictable behavior of the resulting systems and networks.

Bibliography

[1] R.P. Abbott et al. Security analysis and enhancements of computer operating systems. Technical report, National Bureau of Standards, 1974. Order No. S-413558-74.

[2] A. Avižienis and J-C. Laprie. Dependable computing: from concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[3] A. Avižienis and J. C. Laprie, editors. *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, Santa Barbara, CA, August 1989. Springer-Verlag, Vienna, Austria.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, 1999. (http://www.extremeprogramming.org).

[5] P. Boudra, Jr. Report on rules of system composition: Principles of secure system design. Technical report, National Security Agency, Information Systems Security Organization, Office of Infosec Systems Engineering, I9 Technical Report 1-93, Library No. S-240, 330, March 1993. For Official Use Only.

[6] Canadian Systems Security Centre, Communications Security Establishment, Government of Canada. *Canadian Trusted Computer Product Evaluation Criteria*, December 1990. Final Draft, version 2.0.

[7] A. Chander, D. Dean, and J.C. Mitchell. Deconstructing trust management. In *Proceedings of the 2002 Workshop on Issues in the Theory of Security*, Portland, Oregon, January 2002. IFIP Working Group 1.7.

[8] D.D. Clark et al. *Computers at Risk: Safe Computing in the Information Age*. National Research Council, National Academy Press, 2101 Constitution Ave., Washington, D.C. 20418, 5 December 1990. Final report of the System Security Study Committee.

[9] F.J. Corbató. On building systems that will fail (1990 Turing Award Lecture, with a following interview by Karen Frenkel). *Communications of the ACM*, 34(9):72–90, September 1991.

[10] F.J. Corbató, J. Saltzer, and C.T. Clingen. Multics: The first seven years. In *Proceedings of the Spring Joint Computer Conference*, volume 40, Montvale, New Jersey, 1972. AFIPS Press.

[11] M. Curtin. *Developing Trust: Online Security and Privacy.* Apress, Berkeley, California, and Springer-Verlag, Berlin, 2002.

[12] D.E. Denning, P.G. Neumann, and Donn B. Parker. Social aspects of computer security. In *Proceedings of the 10th National Computer Security Conference*, September 1987.

[13] European Communities Commission. *Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonised Criteria (of France, Germany, the Netherlands, and the United Kingdom)*, June 1991. Version 1.2. Available from the Office for Official Publications of the European Communities, L-2985 Luxembourg, item CD-71-91-502-EN-C. Also available from UK CLEF, CESG Room 2/0805, Fiddlers Green Lane, Cheltenham UK GLOS GL52 5AJ, or GSA/GISA, Am Nippenkreuz 19, D 5300 Bonn 2, Germany.

[14] R.J. Feiertag and P.G. Neumann. The foundations of a Provably Secure Operating System (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979. http://www.csl.sri.com/neumann/psos.pdf.

[15] C. Gacek and C. Jones. Dependability issues in open source software. Technical report, Department of Computing Science, Dependable Interdisciplinary Research Collaboration, University of Newcastle upon Tyne, Newcastle, England, 2001. Final report for PA5, part of ongoing related work.

[16] C. Gacek, T. Lawrie, and B. Arief. The many meanings of open source. Technical report, Department of Computing Science, University of Newcastle upon Tyne, Newcastle, England, August 2001. Technical Report CS-TR-737.

[17] C. Gunter, S. Weeks, and A. Wright. Models and languages for digital rights. In *Proceedings of the 2001 Hawaii Intenational Conference on Systems Science*, Honolulu, Hawaii, March 2001. (http://www.star-lab.com/tr/star-tr-01-04.html).

[18] International Standards Organization. *The Common Criteria for Information Technology Security Evaluation, Version 2.1, ISO 15408.* ISO/NIST/CCIB, 19 September 2000. (http://csrc.nist.gov/cc).

[19] R. Jagannathan and C. Dodd. GLU programmer's guide v0.9. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, November 1994. CSL Technical Report CSL-94-06.

[20] J.C. Laprie, editor. *Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance.* Springer-Verlag, 1990.

[21] NCSC. *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC).* National Computer Security Center, December 1985. DOD-5200.28-STD, Orange Book.

[22] P.G. Neumann. The role of motherhood in the pop art of system programming. In *Proceedings of the ACM Second Symposium on Operating Systems Principles, Princeton, New Jersey*, pages 13–18. ACM, October 1969.

[23] P.G. Neumann. Rainbows and arrows: How the security criteria address computer misuse. In *Proceedings of the Thirteenth National Computer Security Conference*, pages 414–422, Washington, D.C., 1–4 October 1990. NIST/NCSC.

[24] P.G. Neumann. *Computer-Related Risks.* ACM Press, New York, and Addison-Wesley, Reading, Massachusetts, 1995.

[25] P.G. Neumann. Practical architectures for survivable systems and networks. Technical report, Final Report, Phase Two, Project 1688, SRI International, Menlo Park, California, June 2000. (http://www.csl.sri.com/neumann/survivability.html).

[26] P.G. Neumann. Robust nonproprietary software. In *Proceedings of the 2000 Symposium on Security and Privacy*, pages 122–123, Oakland, California, May 2000. IEEE Computer Society. (http://www.csl.sri.com/neumann/ieee00.ps and http://www.csl.sri.com/neumann/ieee00.pdf).

[27] P.G. Neumann. Achieving principled assuredly trustworthy composable systems and networks. In *Proceedings of the DARPA Information Survivability Conference and Exhibition, DISCEX3, volume 2*, pages 182–187. DARPA and IEEE Computer Society, April 2003.

[28] P.G. Neumann. Illustrative risks to the public in the use of computer systems and related technology, index to RISKS cases. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 2003. The most recent version is available online in html form for browsing at http://www.csl.sri.com/neumann/illustrative.html), and also in .ps and .pdf form for printing in a much denser format.

[29] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[30] P.G. Neumann and R.J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, pages 208–216, Las Vegas, Nevada, December 2003. IEEE Computer Society. http://www.acsac.org/ .

[31] P.G. Neumann and D.B. Parker. A summary of computer misuse techniques. In *Proceedings of the Twelfth National Computer Security Conference*, pages 396–407, Baltimore, Maryland, 10–13 October 1989. NIST/NCSC.

[32] E.I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, Massachusetts, 1972.

[33] W. Ozier. GASSP: Generally Accepted Systems Security Principles. Technical report, International Information Security Foundation, June 1997. web.mit.edu/security/www/gassp1.html.

[34] B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors. *Predictably Dependable Computing Systems.* Basic Research Series. Springer-Verlag, Berlin, 1995.

[35] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975. (http://www.multicians.org).

[36] M.D. Schroeder. Cooperation of mutually suspicious subsystems in a computer utility. Technical report, Ph.D. Thesis, M.I.T., Cambridge, Massachusetts, September 1972.

[37] W.H. Ware. A retrospective of the criteria movement. In *Proceedings of the Eighteenth National Information Systems Security Conference*, pages 582–588, Baltimore, Maryland, 10–13 October 1995. NIST/NCSC.

[38] S. Weeks. Understanding trust management systems. In *Proceedings of the 2001 Symposium on Security and Privacy*, Oakland, California, May 2001. IEEE Computer Society. (http://www.star-lab.com/tr/star-tr-01-02.html).

Software Engineering
Correctness by Construction

Issue: 1.1
Status: Provisional
13th January 2004

Originator

Anthony Hall, Rod Chapman

Approver

Martin Croxford

Copies to:

SEI
Noopur Davies

Praxis Critical Systems
Dewi Daniels

Management Summary

Correctness by Construction is a radical, effective and economical method of building software with high integrity for security-critical and safety-critical applications. Praxis Critical Systems use it to produce software with extremely low defect rates – fewer than 0.1 defects per thousand lines of code – with good productivity – up to around 30 lines of code per day.

The principles of Correctness by Construction are:

1 Don't introduce errors in the first place.

2 Remove any errors as close as possible to the point that they are introduced.

These are achieved by

1 Using a sound, formal, notation for all deliverables. For example, we use Z for writing software specifications, so it is impossible to be ambiguous. We code in SPARK, so it is impossible to introduce errors such as buffer overflow.

2 Using strong, tool-supported methods to validate each deliverable. For example we carry out proofs of formal specifications and static analysis of code. This is only possible because we use formal notations.

3 Carrying out small steps and validating the deliverable from each step. For example, we develop a software specification as an elaboration of the user requirements, and check that it is correct before writing code. We build the system in small increments, and check that each increment behaves correctly.

4 Saying things only once. For example, we produce a software specification, which says what the software will do, and a design, which says how it will be structured. The design does not repeat any information in the specification, and the two can be produced in parallel.

5 Designing software that's easy to validate. We write simple code that directly reflects the specification, and test it using tests derived systematically from that specification.

6 Doing the hard things first. For example we produce early prototypes to test out difficult design issues or key user interfaces.

As a result, Correctness by Construction is both effective and economical:

1 Defects are removed early in the process when changes are cheap. Testing becomes a confirmation that the software works, rather than the point at which it must be debugged.

2 Evidence needed for certification is produced naturally as a by-product of the process.

3 Early iterations produce software that carries out useful functions and builds confidence in the project.

Contents

1 Introduction

This document describes Correctness by Construction, the Praxis Critical Systems process for developing high integrity software. This is a flexible process which we have used to develop security-critical and safety-critical software. It delivers software with very low defect rates, by rigorously eliminating defects at the earliest possible stage of the process. It is an economical process because the time spent on early deliverables is more than recouped in the very small amount of rework necessary at late stages of the project.

The process consists of a number of steps each producing a deliverable, supported by a number of generic activities such as configuration management. The process is flexible in that the techniques used for each step can vary according to the project, and the timing and extent of steps can be changed according to the needs of the application. However, all variants of the process are based on the strong principle that each step should serve a clear purpose and be carried out using the most rigorous techniques available that match the particular problem. In particular we almost always use formal methods to specify behavioural, security and safety properties of the software, since only by using formality can we achieve the necessary precision.

Section 2 is an overview of the process and describes its main characteristics. Section 3 gives more detail of the process steps. Section 4 describes generic activities that take place throughout the process. Section 5 gives examples of process use. Appendix A describes SPARK, a language designed for secure and safe systems development.

2 Overview of the Process

2.1 Process Outline

Figure 1 is a simplified diagram of the process. It uses the symbols shown in Figure 2. It shows the main activities and deliverables, and the general flow of time from top to bottom. It does not show some crucial aspects of the process:

1 There is more overlap between different activities than can be shown in a figure.

2 The figure omits the outputs of the validation steps. Any validation step can affect any previous deliverable and cause re-entry to any previous activity.

3 We build the system top down and incrementally.

Correctness by construction depends on knowing what the system needs to do and being sure that it does it. The first step, therefore, is to develop a clear statement of requirements. However, it is impossible to develop code reliably from requirements: the semantic gap is just too big. We therefore use a sequence of intermediate descriptions of the system to progress in tractable, verifiable steps from the user-oriented requirements to the system-oriented code. At each step we typically have several different descriptions of different aspects of the system. We ensure that these descriptions are consistent with each other and we ensure that they are correct with respect to the earlier descriptions.

1 The **User Requirements** describe the purpose of the software, the functions it must provide and the non-functional requirements such as security, safety and performance.

2 The **Software Specification** is a complete and precise description of the behaviour of the software viewed as a black box. It contains no information about the software's internal structure.

3 The **High Level Design** describes the architecture of the software.

4 A number of **Detailed Designs** describe the operation of different aspects of the software, such as its process structure or database schema.

5 **Module Specifications** define the state and behaviour encapsulated by each software module.

6 **Code** is the executable code of each module.

7 Each **Build** is a version of the software which offers a subset of its behaviour. Typically early builds contain only infrastructure software and little application functionality. Each build acts as a test harness for subsequent code.

8 The **Installed Software** is the final build, configured and installed in its operational environment.

Section 3 describes each of these deliverables in more detail.

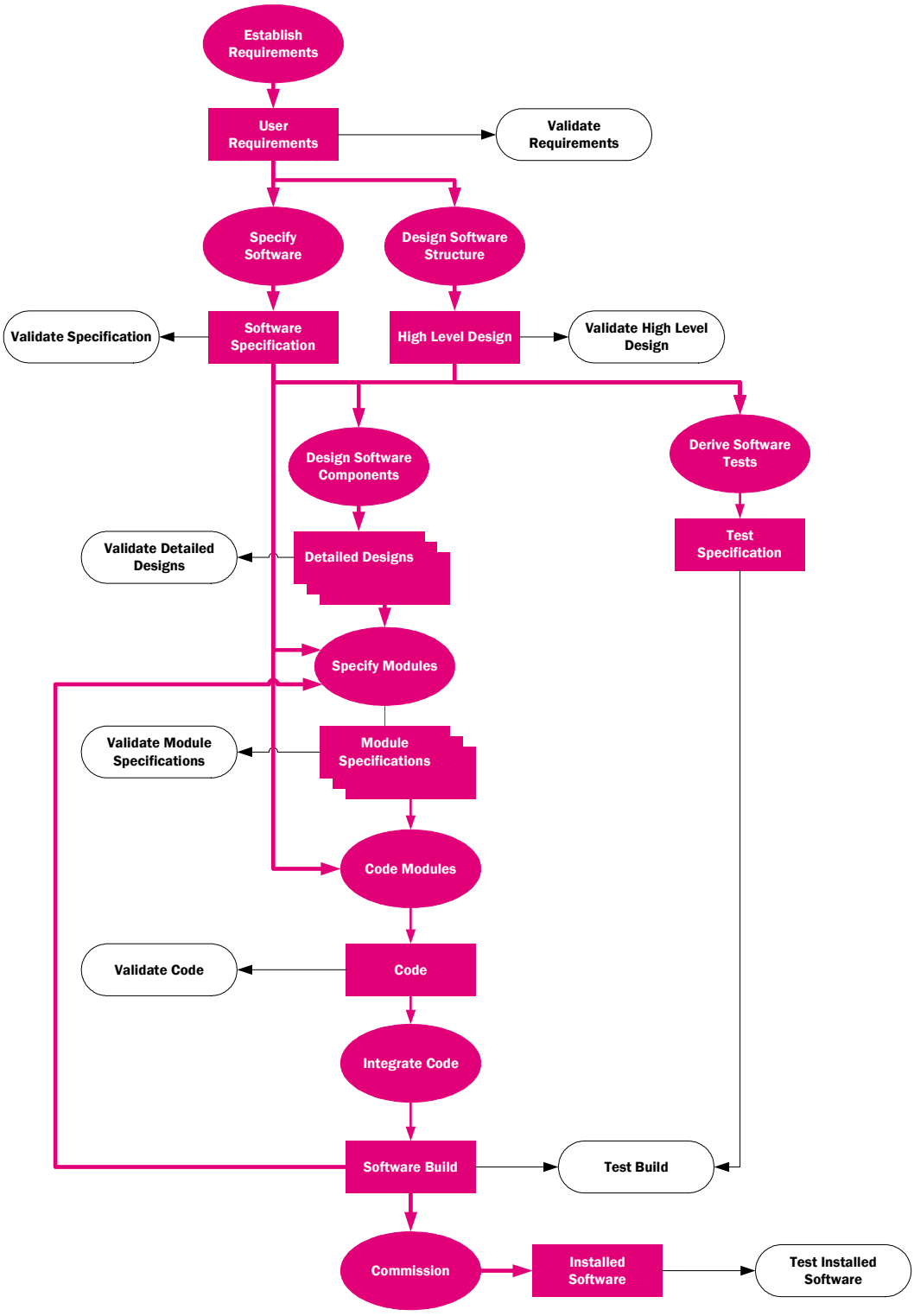


Figure 1 Core Process

This diagram is simplified by omitting most of the parallelism and iteration.

Figure 2 Key to Process Diagram

2.2 Process Characteristics

2.2.1 Risk Driven

We choose the set of activities and the order we do activities to minimise the risk of late problems. We therefore do the most risky activities first. If, for example, we are uncertain about the feasibility of meeting some requirement, we will do design trailblazing to establish a feasible design ahead of completing the requirements or specification. We also choose how much specification and design to do on the basis of risk: if an area is straightforward, we may go straight from requirements to code, while in a difficult area we will write very detailed and formal specifications.

2.2.2 Confidence building

The Correctness by Construction process provides evidence, throughout the process, about the correctness of the software being built. This evidence builds confidence that there will be no late-breaking serious faults. It also supports evaluation and certification of security-critical software, for

example against the Common Criteria, and of safety-critical software, for example against DEF STAN 00-55.

2.2.3 Parallel

Although the figure appears to describe a largely sequential process, we actually use a lot of parallelism to reduce timescales. There are three ways we can achieve this:

1 Where two different kinds of activity are independent, we do them in parallel. For example, the high level design is based largely on non-functional requirements and does not depend on details of the functional specification, so it can be done in parallel with the software specification.

2 Where the system can be partitioned into different areas, these can be developed in parallel. They may be at different stages of development at the same time, or progress through the same stages at the same time.

3 Incremental builds allow us to carry out testing of one build in parallel with coding of the subsequent build.

2.2.4 Iterative

Whenever we find a fault, we iterate back to the point at which the fault was introduced and rework all subsequent deliverables. (Obviously we do this in batches, not for each individual fault.) This ensures that all deliverables are kept consistent at all baselines.

2.2.5 Rigorous

At each stage, we use descriptions that are as formal as possible. This has two benefits. First, formal descriptions are more precise than informal ones, and therefore they force us to understand issues and questions before we actually produce the code. Second, there are more powerful verification methods for formal descriptions than there are for informal ones, so we have more confidence in the correctness of each step. In particular, formal methods allow some degree of automated checking of the deliverables and of the relationships between them.

2.2.6 Early validation

The aim of correctness by construction is to prevent faults and to eliminate as early as possible any faults that are introduced. Therefore each deliverable is validated as rigorously as possible. Wherever possible we use formal notations and automated tools to validate specifications and designs before any faults get through to code.

2.2.7 Efficient

There are two reasons why Correctness by Construction is an efficient process. The first is that it minimises late rework. Because faults are detected and removed as early as possible, few faults survive to the late stages of the project. The second is that it minimises duplication and repetition of work. The deliverables all describe different aspects of the software and there is little overlap between them. This contrasts with methods where each deliverable is essentially an expansion of the previous one.

2.2.8 Measured

We keep metrics on size, productivity and defect rates across the process.

2.2.9 Improved through root cause analysis

We do root cause analysis of significant faults and continuously improve the process.

2.2.10 Flexible

Correctness by construction is not a single, rigid process. Rather it is a framework and set of principles. For any particular project we tailor it based on the nature and criticality of the project. Projects may differ in many aspects:

1 Level of rigour
Some projects require fully formal proofs of correspondence between formal specifications; others may not justify any formality at all.

2 Techniques/notations at each stage
Different kinds of software require different notations. For example embedded systems need a very different style of specification from database applications.

3 Subsets of activities
Some projects may omit some of the activities, or add extra activities.

4 Content of design
The amount of detail in the design will depend on the size and complexity of the system.

5 Formality of evaluation
The amount of evidence that is collected and the rigour with which the evidence is controlled can be adapted according to how rigorously the software is to be evaluated. The process is capable of developing to software to the highest levels of safety (for example safety integrity level 4 as defined by UK MoD DEF STAN 00-56) and security (for example Common Criteria assurance level EAL 7).

3 Process Steps

This section describes the production and validation of each of the process deliverables.

3.1 Requirements

The User Requirements define:

1 The overall objectives of the system.

2 The system context: the people and other systems that interact with it.

3 Relevant facts about the application domain.

4 Functions to be provided by the system and scenarios showing how they achieve the overall objectives.

5 Non-functional characteristics such as capacity, throughput, reliability, safety and security.

We establish and describe the requirements using Praxis' REVEAL® method. They may be captured in a document or in a requirements tool such as DOORS or Requisite Pro.

We pay particular attention to the changeability of requirements. We identify those requirements and assumptions that are relatively stable, and those that are more likely to change. This allows us to design the system to cope with the likely changes that will occur during its development and use.

The requirements for secure systems include the security target. This will be stated in English language. For high levels of assurance, we also write a Formal Security Policy Model. This formalises the technical aspects of the security target. This has two benefits:

1 It makes the security target absolutely precise.

2 It allows more rigorous validation of subsequent deliverables.

The user requirements are validated by review. The review includes the users of the system and also developers and testers, who ensure that the requirements are feasible and testable.

3.2 Specification

The specification is a complete black-box description of the behaviour of the software. It describes several aspects:

1 Functionality
We specify the functionality by writing an abstract description of the system state and a description of the effect of each operation in terms of inputs, outputs and state changes. We always give a

complete description including both normal and error outcomes. We write this description in a formal notation such as Z [4], although we also use notations such as UML class diagrams to describe the system state and, in some cases, state diagrams to describe behaviour.

2 Concurrency
Some systems have a high degree of concurrency visible to the users and there may be rules about what concurrent behaviours are allowed. If so we describe these rules, using a formal language such as CSP [5].

3 User Interface
We describe in detail the required look and feel of the user interface.

4 System Interfaces
We produce an interface control document defining the interface with each connected system.

We validate the software specification by:

1 Review
This is a manual check that the specification is
   a self consistent (both within each part, and that the abstract specifications are consistent with the user and system interfaces);
   b correct with respect to the user requirements;
   c complete;
   d implementable.

2 Prototyping
We build a prototype of the user interface and evaluate it with suitable user representatives. We may also build prototypes of critical functionality to validate the abstract specification.

3 Formal Analysis
When we have a formal specification we can carry out proofs to show that it is self consistent and has some completeness properties. In critical secure applications we can also prove correspondence between the specification and the formal security policy model.

3.3 High Level Design

The high level design is a top level description of the system's internal structure and an explanation of how the components worked together. There are several different descriptions, looking at the structure from different points of view. In a secure system the descriptions typically include:

1 distribution of functionality over machines and processes;

2 database structure and protection mechanisms;

3 mechanisms for transactions and communications.

We use different notations for the different aspects of the design. Not all aspects have formal notations. Where possible we do use formal notations: in particular we use CSP to define any complex process structure.

The high level design is primarily derived from the non-functional requirements and can be developed in parallel with the software specification.

We validate the high level design by:

1 Review, to ensure that it

   a is self consistent;

   b satisfies the requirements;

   c is implementable.

2 Automated analysis
   If we have a formal design in CSP we use automated tools such as model checkers to validate that the design has desired properties such as freedom from deadlock.

3.4 Detailed Design

The detailed design serves two purposes

1 defining the set of software modules and processes and allocating the functionality across them;

2 providing more detail of particular modules wherever that is necessary.

3.4.1 Module Structure

The module structure describes the software architecture and how functionality described in the specification and high-level design is allocated to each module. We recognize that the structure of the implementation may differ from that of the specification, for example where an atomic transaction in the specification is distributed over many processes or machines in the implementation.

We use an information-flow centric design approach, called INFORMED, to drive and evaluate the module structure. We developed INFORMED within Praxis Critical Systems to support the design of high integrity software. It leads to a software architecture that exhibits low coupling and high cohesion. This, in turn, benefits the later maintainability of the system in the face of subsequent changes. For secure systems, we categorize system state and operations according to their impact on security. We aim for an architecture that minimizes and isolates security-critical functions, so reducing the cost and effort of the (possibly more rigorous) verification of those units. We use a similar approach for safety-critical software.

For embedded and real-time systems, we also consider throughput, timing and scheduling issues at this stage. Systems with particularly stringent hard real-time requirements, for example, might constrain the style of implementation architecture that can be employed.

3.4.2 Low-level details

We deliberately do not write a detailed design of every aspect of the system. Often the software specification and module structure together give enough information to create software directly. We do not duplicate information that is already in the specification or HLD. However we may provide more detailed designs for, for example:

1 database table structures;

2 complex areas of functionality: these arise where there is a big difference between the implementation structure and the conceptual structure in the software specification, or where the software specification has omitted details of some complex processing;

3 user interface code;

4 low-level device handling;

5 rules for mapping specification or design constructs into code: for example, rules for translating CSP into Ada tasking constructs or Z types into implementation structures.

The detailed designs use different notations for different aspects. We use formal specifications of low-level modules to clarify complex functionality.

The detailed design is validated by

1 review, to ensure that it is self consistent, efficient and satisfies the specification and the high level design;

2 formal analysis
A formal low-level specification can be proved correct with respect to a higher level specification.

3.5 Test Specifications

We derive test specifications primarily from the software specification, together with the requirements and the high-level design. We use boundary value analysis to generate tests which cover the specification. We then supplement these with tests for behaviour which is introduced by the design but is not visible in the specification. In addition we generate tests for non-functional requirements directly from the requirements document.

3.6 Module Specifications

For each module identified in the module structure, we may construct a more detailed specification for its implementation. We may code directly from the system specification if that specification is already suitably detailed and the "gap" between the specification and code is sufficiently narrow.

The module specification serves as a contract between the system specification or a detailed design and the code itself. We favour languages that have a well-defined semantics and directly support design-by-contract. Depending on the application domain, the module specification may be expressed in a model-based notation (such as UML class specifications), an executable specification (such as Statecharts or control-law specifications), or a design-by-contract programming language (such as Eiffel or SPARK).

We validate the module specifications by review, using tools as far as possible, to check for internal consistency, validity with respect to the system specification and so on.

3.7 Code

For coding, we use languages and tools that are most appropriate for the task at hand. Validation requirements play a large role in this choice—languages must be amenable to verification and analysis so that the required evidence of fitness-for-purpose can be generated effectively. We also recognize that no one language is most suitable for all modules—in a secure system, for example, we might use different languages for the security kernel, the system's infrastructure, and the user-interface.

Code is derived from the system specification, module specifications, and low-level designs. We use automatic code generation where domain-specific tools (such as GUI-builders or control system design packages) are mature.

We validate code using both static and dynamic techniques. We use static verification tools are far as possible, since these prevent and detect defects earlier in the life-cycle than testing would allow. Such tools range from simple style and subset-checking up to fully formal program verification systems. We always perform manual code-review, although this is only ever performed after application of static analysis, and we tune the review process to account for the classes of defect that the tools can eliminate.

3.8 Building

We build the system top down and incrementally, with a formal build every few weeks. The first build consists of the system framework, such as the top-level processes on each machine and the connections to the external devices. It does not, typically, contain much application functionality. Each build acts as a test harness for later code.

We test each build by running a subset of the system tests. We use automated regression testing to ensure that all previous tests are still passed. We measure code coverage as we carry out the tests. When we find gaps in coverage, we do one of 3 things:

1 Usually the gap is caused by code which is there to implement some aspect of the design not visible at the specification level. In that case we add suitable tests.

2 Sometimes the gap reflects code which is not in fact necessary, and the code is removed.

3 Sometimes the code cannot be reached by normal operation of the system, but is still necessary – for example defensive code. In that case, and only then, we write unit tests at the module level.

Apart from this last case, we do no formal unit testing. Unit testing is costly and ineffective at finding errors in comparison with proof and static analysis [3].

3.9 Commissioning

The commissioning process largely depends on the criticality of the system being delivered and its operational environment. At the least, we use a documented process for the labelling and delivery of software builds to the customer. A build is accompanied by a "release certificate" that summarizes the status and composition of that particular build. For some applications, we also issue a safety certificate and a warranty.

We supply a Commissioning Guide to the customer, which details the installation of the software onto the target environment. This may also contain details of how the system's hardware is constructed, and an inventory of the required components.

For secure systems, we go further. Software may be delivered in tamper-evident bags, for example, according to the customer's (and regulator's) requirements. Such systems may be commissioned in a physically secure environment, and commissioning may be witnessed by us, the customer, independent auditors, evaluators, regulators and so on.

4 Generic Activities

4.1 Process Planning

We write a technical plan for each project. This describes what parts of the process we will use, what techniques we will use at each stage and what validation activities we will carry out. Thus the process is tailored for each particular application.

4.2 Staff Competence and Training

We ensure that all staff working on a project are competent in the relevant areas. We use in-house or bought-in training to maintain our skill levels.

4.3 Tracing

We maintain tracing information showing how each description is related to its successor and predecessor. For example we record how each requirement is satisfied in the specification. We use automated tools to check that all requirements are completely traced through to code, and that all code is ultimately traceable back to the requirements.

4.4 Fault management

Fault management is a key part of Correctness by Construction, since the whole aim is to remove faults as early as possible. Each deliverable is subject to fault management as soon as it has been baselined. Faults are identified by the validation activities and also by use of deliverables in subsequent stages – for example, a coder may find a fault in the specification.

When a fault is identified, we do two things:

1 Identify and fix all the deliverables affected by the fault. These may include deliverables earlier than that in which the fault was first identified, as well as deliverables derived from it.

2 Do root cause analysis to determine why the fault was introduced, and if possible change the process to avoid faults of this sort appearing in future.

4.5 Change management

The key to change management is impact analysis. We find that the rigorous specification and design information makes impact analysis highly effective. For each change we are able to give a detailed assessment of the effect on each deliverable.

Change management is also helped by design for change, starting with the changeability requirements described in section 3.1.

4.6 Configuration management

All deliverables including documents as well as code are under formal tool-supported configuration management. This enables us to identify versions of individual items and baseline configurations of consistent sets of documents and code.

4.7 Team organisation

For critical projects we do all formal testing using a team independent of the implementers.

4.8 Metrics collection

We collect metrics on effort, sizes of deliverables, numbers of faults and for each fault its point of introduction and point of detection.

5 Examples of Process Use

This section presents metrics for five projects that have used instances of the Correctness-by-Construction approach. These projects differ in size, application domain and complexity, although all are classed as "high integrity"—three of the projects have critical safety-relation functions, while the other two have significant security requirements.

The following paragraphs give a brief description of each project. The projects are identified by name where, given clients' confidentiality, we are able to do so at the time of writing.

5.1 CDIS

CDIS is a real-time air traffic information system. It has stringent performance and availability requirements and has proved very reliable in over 11 years of use at the London Terminal Control Centre. The methods used in developing CDIS have been described in an article [9] and there has been an independent assessment of the project [10].

5.2 SHOLIS

The Ship/Helicopter Operating Limits Information System aids the safe operating of helicopter operations on naval vessels. It is essentially an information system, giving advice on the safety of helicopter operations given a particular operating scenario and environmental conditions such as the incident wind vector and the roll and pitch of the ship. The system's primary safety function is to raise audible and visible alarms when environmental conditions step outside of pre-defined operating limits.

SHOLIS was the first project to carry out a full SIL 4 development under the UK MoD's Def Stan 00-55. Further information on SHOLIS can be found in [3].

5.3 The MULTOS CA

The MULTOS CA is the "root" certification authority for the MULTOS smartcard system. The CA produces digital certificates that are used in the manufacturing of MULTOS smartcards, and also certificates that allow trusted applications to be loaded onto a card in the field. The system has demanding throughput and availability requirements, and so is both distributed and fault-tolerant.

The CA was developed as far as was practicable in line with the UK ITSEC scheme at evaluation level E6—roughly equivalent to Common Criteria EAL7. Further information on the development of the CA can be found in [1].

5.4 Project A

This project is a military stores management system. It enforces a small number of safety functions, and was developed in line with the UK's Def Stan 00-55[8] standard at SIL 3. This project is embedded, and combines a simple user-interface with complex hard real-time requirements.

5.5 Project B

This project is the core of a biometric access-control system. It has been developed using Correctness-by-Construction to meet or exceed the requirements of the Common Criteria at evaluation/assurance level EAL5.

5.6 Metrics

Table 1 presents key metrics for each of the above projects. The first column identifies each project. The second identifies the year in which the system was first commissioned—the projects are presented in chronological order. Column three shows the size of the delivered system in physical lines of code. This is always executable lines and declarations, but does not include comments, blanks lines, or "annotations" used for design-by-contract. The fourth column presents productivity—this is the lines of code divided by the total project effort for **all** project phases from project start up to the completion of commissioning. The final column reports defect rate in defects per thousand lines of code.

Project	**Year**	**Size (loc)**	**Productivity (loc per day)**	**Defects (per kloc)**
CDIS	1992	197,000	12.7	0.75
SHOLIS	1997	27,000	7.0	0.22 (note 1)
MULTOS CA	1999	100,000	28.0	0.04 (note 2)
A	2001	39,000	11	0.05 (note 3)
B	2003	10,000	38.0	not yet known (note 4)

Table 1: Correctness-by-Construction project metrics

Notes:

1. 0 defects during acceptance test and sea-trial. 6 defects subsequently discovered and corrected in first 3 years of in-service use.

2. 4 defects reported and corrected during 1-year warranty period following commissioning.

3. 2 defects in 2 years following delivery. However, the system is not yet rolled out for operational service.

4. This project has been undergoing independent evaluation for some months. No defects have been detected so far but the final results will not be available until February 2004.

A SPARK

The SPADE Ada Kernel (SPARK) is a language designed for the construction of high-integrity systems. The executable part of the language is a subset of Ada95, but the language requires additional annotations that make it possible to carry out data and information flow analysis[6], and to prove properties of code, such as partial correctness and freedom from exceptions.

The design goals of SPARK are as follows:

Logical soundness: there should be no ambiguities in the language;

Simplicity of formal description: it should be possible to describe the whole language in a relatively simple way;

Expressive power: the language should be rich enough to construct real systems;

Security: it should be possible to determine statically whether a program conforms to the language rules;

Verifiability: formal verification should be theoretically possible and tractable for industrial-sized systems.

The annotations in SPARK appear as comments (and so are ignored by a compiler), but are processed by the Examiner tool. These largely concern strengthening the "contract" between the specification and the body of a unit (for instance specifying the information flow between referenced and updated variables.) The presence of the annotations also enables the language rules to be checked efficiently, which is crucial if the language is to be used in large, real-world applications.

SPARK actually has its roots in the security community. Research in the 1970's[7] into information flow in programs resulted in SPADE Pascal and, eventually, SPARK. SPARK is widely used in safety-critical systems, but we believe it is also well-suited to the development of secure systems. SPARK offers static protection from several of the most common implementation flaws that plague secure systems:

- **Buffer overflows**. Proof of the absence of predefined exceptions (for such things as buffer overflows) offer strong *static* protection from a large class of security flaw. Such things are an anathema to the safety-critical community, yet remain a common form of attack against networked computer systems. The process of attempting such proofs also yields interesting results: a proof which doesn't "come out" easily often is indicative of a bug, and the proof *forces* an engineer to read, think about, and understand their programs in depth, which can only be a good thing.

- **Run-Time Library Defects.** SPARK can be compiled with no supporting run-time library, implying that an application can be delivered with no COTS component. At the highest assurance levels, this may be of significant benefit, where evaluation of such components remains problematic.

- **Timing and memory attacks.** SPARK is amenable to the *static* analysis of timing and memory usage. This problem is known to the real-time community, where analysis of worst-case execution

time is often required. In the development of secure systems, it may be possible to use such technology to ensure that programs exhibit as little *variation* in timing behaviour as possible, as a route to protect against timing analysis attacks.

- **Input Data Validation.** The SPARK verification system is conservative, and does not trust data coming from the external environment. Formally speaking, the verification condition generator does not automatically add hypotheses regarding input data, so that subsequent proofs (e.g. for a range check where such an input is used) cannot be discharged until the validity of that input has been explicitly checked. In short, SPARK forces the programmer to validate input data (or at least provides a very strong reminder to do so!)

Additionally, SPARK provides additional forms of verification, such as:

- Program-wide, complete data- and information-flow analysis. These analyses make it impossible for a SPARK program to contain a dataflow error (e.g. the use of an uninitialized variable)—a common implementation error that can be the cause of subtle (and possibly covert) security flaws.

- Proof of correctness of SPARK programs is achievable, and so allows a program to be shown to correspond with some suitable formal specification. This allows for formality in the design and specification of a system to be extended through its implementation and can meet the requirements of the CC scheme at the highest evaluation levels.

More information about SPARK can be found at www.sparkada.com

Document Control and References

Changes history

Issue 0.1 (12th January 2004): First draft for internal inspection.

Issue 1.0 (12th January 2004): First external issue following inspection.

Issue 1.1 (13th January 2004): Management summary added and minor corrections made.

Changes forecast

May subsequently be developed further.

Document references

1 *Correctness by Construction: Developing a Commercial Secure System,* Anthony Hall and Roderick Chapman, *IEEE Software* Jan/Feb 2002, pp18-25.

2 *Will it Work?*, Jonathan Hammond, Rosamund Rawlings and Anthony Hall, in *Proceedings of RE'01, 5th IEEE International Symposium on Requirements Engineering*, August 2001

3 *Is Proof More Cost-Effective Than Testing?*, Steve King, Jonathan Hammond, Rod Chapman and Andy Pryor, IEEE Transactions on Software Engineering, Vol 26 No 8, pp675–686, August 2000

4 *The Z Notation: A Reference Manual*, J. M. Spivey, 2nd Edition. Prentice-Hall, 1992.

5 *Communicating Sequential Processes*, C. A. R. Hoare, Prentice Hall, 1985.

6 Bergeretti, J-F., and Carré, B. A., *Information-Flow and Data-Flow Analysis of While Programs*. ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, January 1985. p.p. 37–61.

7 Denning, D. E., and Denning, P. J. *Certification of Programs for Secure Information Flow*. CACM Vol. 20, No. 7. July 1977.

8 United Kingdom Ministry of Defence, *The Procurement of Safety Critical Software in Defence Equipment.* DEF STAN 00-55.

9 *Using Formal Methods to Develop an ATC Information System*, Anthony Hall, IEEE Software, March 1996, pp 66-76.

10 *Investigating the Influence of Formal Methods*, Shari Lawrence Pfleeger and Les Hatton, IEEE Computer, February 1997, pp 33-43.

Developing Secure Software with Cleanroom Software Engineering

Richard C. Linger
CERT Research Center
Software Engineering Institute
Carnegie Mellon University, Pittsburgh, PA
rlinger@sei.cmu.edu

Stacy J. Prowell
Department of Computer Science
The University of Tennessee
Knoxville, Tennessee
sprowell@cs.utk.edu

1. Defective Software Cannot be Secure

Secure systems must be composed of secure software components, whether developed or acquired. The first requirement for secure software is specifications that define secure behavior. The specifications must define security functionality and be free of vulnerabilities that can be exploited by intruders. The second requirement for secure software is correct implementation with respect to secure specifications. Software is correct if it exhibits only the behavior defined by its specification. Defective software can exhibit behavior not specified, or even known to its developers and testers. Such behavior can likewise be exploited by intruders. Software is rarely well-specified, and testing is inadequate to eliminate all defects, so it is not surprising that many systems experience a barrage of intrusions and compromises. Nevertheless, the necessary foundation for secure software is well-understood; it is specifications that define secure behavior, and software that correctly implements that behavior. Engineering trade-offs and human fallibility will always be present in software development and operation, so there can be no guarantees of security. But there can be little doubt that the goal of secure specifications correctly implemented will go a long way toward improving the current state of system security. Present methods of software development have produced the present situation and will continue to do so. Intuitive, trial-and-error programming practices are widely used in industry, in large part because so many people have learned to develop software informally and intuitively. Users have learned to expect security failures, and management has learned to put up with the problems they cause. However, the potentially serious consequences of intrusion and compromise argue for better methods.

2. Cleanroom Software Engineering For Developing Secure Software

Cleanroom software engineering [1,2] is a theory-based, team-oriented process for developing and certifying correct software systems under statistical quality control with high productivity. The name "Cleanroom" conveys an analogy to the precision engineering of hardware cleanrooms. Cleanroom covers the entire development life cycle, and is independent of programming language and development environment. Cleanroom technology includes project management by incremental development for risk reduction, function-based specification and design for intellectual control, functional correctness verification for approaching zero defects, statistical testing for certification of software fitness for use, and reverse engineering of existing software to analyze functionality and correctness.

Cleanroom is based on theory, but is not a formal method accessible only to advanced practitioners. It is a practical engineering discipline for use by journeyman engineering teams for fast development of high quality software to schedule and budget. Cleanroom is ensured by its theoretical foundations to be failure-free in its processes, whatever human fallibility may be present in their application. Cleanroom achieves management risk reduction by developing software in a pipeline of executable, earned-value increments designed to accumulate into the final system. Statistical quality control over software development is achieved by separating the design process from the statistical testing process in the pipeline of increments. And conformance to user requirements is achieved by embedding the pipeline in a customer feedback loop of increment evaluation.

Cleanroom teams are small with high capability. In small projects, members of a single team are organized into specification, development, and certification roles. In large projects, multiple teams are employed and entire teams are assigned these roles.

The technologies of Cleanroom are summarized below:

Incremental Development. System development is organized into a series of fast increments for specification, development, and certification. Increment functionality is defined such that successive increments 1) can be tested in the system environment for quality assessment and user feedback, and 2) accumulate into the final system—successive increments plug into and extend the functionality of prior increments; when the final increment is added, the system is complete. The theoretical basis for incremental development is referential transparency between specifications and their implementations. Incremental development is a powerful risk management strategy in large-scale system development. At each stage, an executing partial product provides clear evidence of progress and earned value. The incremental development motto is "quick and clean;" increments are small in relation to entire systems, and developed fast enough to permit rapid response to user feedback and changing requirements.

Specification and Design. Cleanroom treats programs as implementations of mathematical functions or relations. Function specifications can be precisely defined for each increment in terms of black box behavior, that is, mappings from histories of use into responses, or state box behavior, that is, mappings from stimulus and current state into response and new state. At the lower level of program design, intended functions of individual control structures can be defined and inserted into code as comments for use in correctness verification. At each level, behavior with respect to security properties can be defined and validated.

Correctness Verification. Sizable programs contain an enormous number of execution paths that cannot all be verified. However, programs are composed of a finite number of control structures that can be verified against their intended functions, thereby reducing verification to a finite engineering process. A correctness theorem defines the conditions to be verified for each control structure type. Verification is carried out in special team inspections with the objective of producing software approaching zero defects prior to first execution. Vulnerabilities and intrusion pathways, if present, are revealed in the verification process. Experience shows any errors left behind by human fallibility in verification tend to be superficial coding problems, not deep design defects.

Statistical Testing. With no or few defects present at the completion of coding, the role of testing shifts from debugging to certification of software fitness for use through usage-based statistical testing. Models of usage states and their probabilities are sampled to generate test cases that simulate user operations. The models treat legitimate and intrusion usage on a par, thereby capturing both benign and threat environments. In contrast to traditional testing, usage-based testing permits valid statistical prediction of quality with respect to all the executions not tested, a powerful management tool for reducing risk and cost. Usage-based testing tends to find any high-failure-rate defects early, thereby quickly improving the mean time to failure (MTTF) of the software. It is not unusual for testing in traditional software development to consume half of project resources. Cleanroom testing is more efficient, with a resulting improvement in project productivity.

3. Cleanroom Fielded Quality Under 0.1 Errors/KLOC

The Cleanroom process has been applied with excellent results. For example, the Cleanroom-developed IBM Cobol Structuring Facility product automatically transforms unstructured legacy Cobol programs into structured form for improved maintenance, and played a key role in Y2K program analysis. This 85-KLOC program experienced just seven minor errors, all simple fixes, in the first three years of intensive field use, for a fielded defect rate of 0.08 errors/KLOC [3].

Selective application of Cleanroom techniques also yields good results. For example, as reported in [4], Cleanroom specification techniques were applied to development of a distributed, real-time system. Cleanroom specifications were developed for system components, and then transformed into expressions in the process algebra CSP. This allowed use of theorem provers to demonstrate that the resulting system was deadlock-free and independent of timing issues, thereby permitting migration to faster hardware in the future without software modifications. The resulting system consisted of 20 KLOC of C++ which ran correctly in its first test on the target hardware. In twelve months of field use of the system, only eight minor defects were discovered; all localized coding errors easy to diagnose and fix.

A number of Cleanroom projects involve classified activities that cannot be reported upon. Overall experience shows, however, that fielded defect rates range from under 0.1 errors/ KLOC with full Cleanroom application, to 0.4 errors/KLOC with partial Cleanroom application. But this is only part of the story. Equally significant is the fact that many code increments never experience the first error in testing, measured from first execution, or in field use. In addition, errors found in testing tend to be simple coding problems, not more serious specification or design errors.

The following sections describe Cleanroom technologies in more detail.

4. Cleanroom Project Management by Incremental Development

4.1 Cleanroom Engineering Activities

There are three primary engineering activities in the Cleanroom process:

- First, a specification team creates a high-level specification and incremental development plan for a pipeline of software increments that will accumulate into the final software product. For each successive increment, a specification is created that includes the statistics of its use as well as its function and performance requirements. Each increment is sized for rapid development and verification, say, on the order of five- to twenty-thousand lines of code.

- Second, a development team designs and codes specified increments using object-based design and functional verification for delivery to the certification team, and provides subsequent correction of any failures uncovered during certification or field use.

- Third, a certification team employs statistical testing to execute and certify successive partial accumulations of increments for correctness with respect to functional specifications, based on the usage specification. It notifies the development team of any failures discovered during certification, and recertifies as failures are corrected.

These activities take place within the framework of fast iterative development. Figure 1 depicts relationships among the activities, together with their performance objectives. The management objective is development of certified software to schedule and budget. The specification objective is correct definition of increment function and usage. The development objective is software delivered to testing with no defects. The testing objective is valid certification of software fitness for use. And the customer feedback objective is conformance to requirements.

The Cleanroom process is a development and certification methodology for releasing software with no known failures, especially any important failures. It requires a test design based not only on function and performance specifications, but also on how important each test case is to system behavior. Such a test design is based on a strategy derived from the statistics of usage expected for the software. Sizable software products exhibit an essentially infinite number of possible executions. No testing process, no

matter how ambitious, can hope to exercise more than a small fraction of these executions. All testing is sampling, and the key question is how to draw the sample. If the sample is statistically representative of anticipated field use, performance of the software on the sample can predict its performance on all the executions not tested, which field users will experience sooner or later. There is an explicit feedback process between certification and development on any failures found in statistical usage testing. This feedback process provides an objective measure of the correctness of the software as it matures in the development pipeline. It does, indeed, provide a statistical quality control process for software project management.

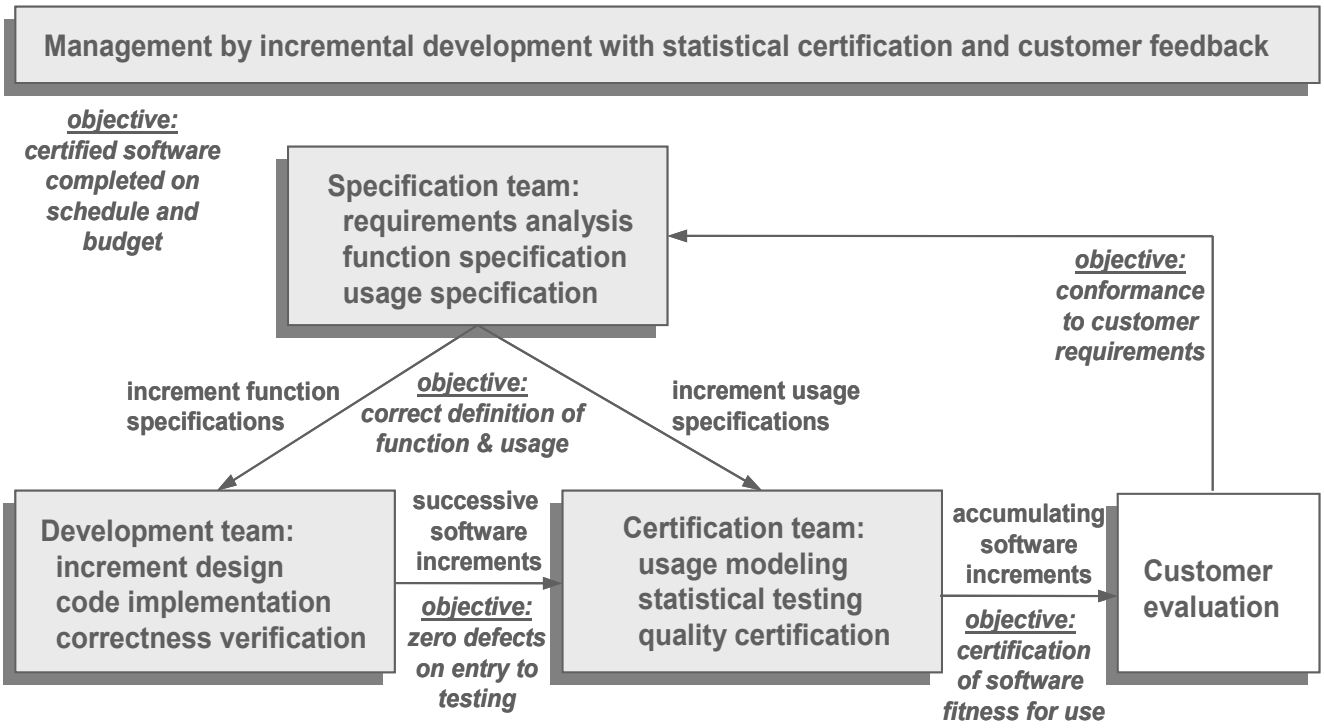


Figure 1. Cleanroom Engineering Processes and Objectives

4.2 Increment Definition and Development

A high-level specification for a software system will identify various classes of users (people and/or other programs), together with commands and data for invoking various system capabilities. For example, bringing up an interactive system will require certain kinds of administrative user commands and initialization data of which ordinary interactive users may not even be aware. However, bringing the system up is an integral part of system operations. During system operation, several distinct classes of users may be interacting simultaneously and independently, such as users adding data to the system, or making inquiries, or monitoring system use and performance. Within each class, many users may be interacting simultaneously and independently, as well.

However, as simultaneously and concurrently as these various users seem to interact with the system, the individual computers in the system each operate strictly sequentially in real time, shifting from one user to another so rapidly that each user gets almost immediate response, even though ten, or a thousand, other users may have been serviced between the system's last response and the user's next stimulus. As a rule, users are separated from one another by operating in different, relatively protected, data spaces that represent the tasks they are performing.

The incremental approach organizes such specification complexities in system behavior into a coherent progression of successive software increments for development and certification. As noted earlier, the functional content selected for each increment is critical, to arrive at successive increments that accumulate into the final system and are testable in the system environment. Early increments typically establish the operational environment and infrastructure, later increments add functionality. In any case, the objective is for each new increment to integrate seamlessly into the previous accumulation with no rework required. Incremental development enables a spiral management process based on visibility and intellectual control. Each increment is itself a spiral of the process from specification to design to verification to testing and certification. Each spiral affords a management opportunity to account for shortfalls and windfalls in planning subsequent spirals. The accumulating increments are provided to users for requirements validation and feedback as the system evolves into final form.

5. Cleanroom Specification and Design

The Cleanroom process of specification and design is based on a usage hierarchy of modules described by a set of operations that define and access internally stored data. In order to create and control such designs in practical ways, Cleanroom specification provides standard, finer grained sub-descriptions for modules in three forms, namely as black boxes, state boxes, and clear boxes, as follows:

1. Black Box. External view of a module, whose behavior is described as a mathematical function from history of stimuli (SH) to next response (R).

2. State Box. Intermediate view of a module, whose behavior is described as a mathematical function from stimuli (S) and current state to next response (R) and new state, plus an initial internal state.

3. Clear Box. Internal view of a module, whose behavior is described in a procedural control structure of uses of other modules. Such a control structure may define sequential, conditional, iterative, or concurrent use of the other modules, right down to individual variables.

The black box view of a module can be refined into a state box by identifying those elements of its stimulus history that must be retained as state data to support all possible black box behavior in the state box definition. Verification of the state box requires demonstration that its external behavior is identical to that of the black box from which it was derived. The state box view of a module can be refined into a clear box by specifying procedures that carry out the state box transition function, possibly through use of other modules. Verification of the clear box requires demonstration that its external behavior is identical to that of the state box from which it was derived.

5.1 Black Box Specification by Sequence Enumeration

Sequence enumeration is a method of iteratively discovering the complete, consistent, and traceably correct black box specification from the initial requirements. Initial system requirements are often written by end users, domain experts, and requirements analysts, and are quite distinct from the rigorous specification required by Cleanroom. These initial requirements often do not cover all scenarios of use, any may contain seemingly inconsistent descriptions of required behavior. Thus, as the sequence enumeration process unfolds, additional requirements are typically discovered and documented. This requirements discovery and documentation is a natural and important part of the Cleanroom specification process.

The first step in sequence enumeration is to construct the list of external interfaces to the system. These may be interfaces with other hardware and software components, or GUI interfaces with the end user. This list of interfaces is called the *system boundary*, and it delimits the problem to be solved.

Next, each interface is considered and stimuli are identified. In this context, a *stimulus* is some event in the system's environment which is observed by the executing software, and a *response* is any system behavior which is observable from the environment. The complete list of stimuli is the prerequisite for starting sequence enumeration. For example, for a simple hand calculator the stimuli might be power on, pressing a digit, pressing an operator, pressing equal, and pressing clear.

A *stimulus history* is the sequence of all stimuli observed by the system over time. All information required by the software to generate responses must either be stored in the software itself, or it must come from the stimulus history. Thus it is possible to write down, for each possible stimulus history, the appropriate next response, and this is in fact what is done during sequence enumeration. For the hand calculator with stimulus history "ON 1 2 + 3 4 =," the appropriate next response is to display 46. That is, the value of the black box function is 46 for stimulus history "ON 1 2 + 3 4 =."

Sequence enumeration is performed by writing down each stimulus history, in order by length, and then noting the appropriate next response (the appropriate response for the most recent stimulus). There are two special cases to deal with: histories for which no external event should be observed, and histories which, given the operational definitions of the system and stimuli, are not possible.

There are often histories for which the appropriate next "response" is to do nothing. Consider a combination safe for which the incorrect combination has just been entered. The safe should not be unlocked; the software should do nothing. For such histories the "response" is said to be the null response.

Consider the following history for the simple calculator: "1 2 + 3 4 = ON." The earliest event in any history for the calculator must be ON. If there were some other event, say pressing the one digit, then for this event to be a stimulus it would have to be observed by the executing software. Thus the ON event would have to have occurred previously, and would appear earlier in the history. In short, the history "1 2 + 3 4 = ON" cannot occur; it is impossible. Further, no history which starts with this sequence can occur. All such histories are said to be *illegal*, and this is noted in place of the response.

As each history is written down, or "enumerated," the appropriate next response (possibly null) is written. If the history is illegal, then "illegal" is written in place of the response. Then traces to the initial requirements are defined which justify the choice of response; every line of the sequence enumeration must have such justification.

For each history there are initially two possibilities:

- The requirements do not specify a response; they are incomplete. A response must be chosen, and the reason for the decision written down as a derived requirement.

- The requirements seemingly specify two distinct responses for the history. A single response must be chosen for each history (though the response may consist of many individual outputs), and the reason for the decision written down as a derived requirement.

Consider the history "ON 1 2 Clear 1 + 1 =" and the history "ON 1 + 1 =." Both histories have the same appropriate next response: display two. Further, if these histories are extended they will always have the same response because the Clear event in the first history makes the prior entries irrelevant. When two histories are intended to agree for all extensions, it makes no sense to enumerate both; only one must be enumerated. In sequence enumeration, the first sequence encountered is extended, but the second sequences is noted as "equivalent" or "reduced" to the first, and need not be extended.

The sequence enumeration process thus continues by extending each legal, unreduced sequence with every stimulus. These new sequences are then considered and a response, trace, and possibly an equivalence noted. When all sequences of the longest length are either illegal or reduced to prior sequences, the sequence enumeration is complete, and the black box specification is fully defined.

Sequence enumeration is an appropriate technique for specifying systems in which the history of events is important. It is inappropriate for systems in which the history is unimportant, such as numerical computations. Fortunately, for most computational problems the mathematical description can serve as the black box specification.

5.2 Sequence Abstraction

The sequence enumeration process described above results in the generation of many sequences to be considered. To control the enumeration process and make it tractable, abstract stimuli are invented and documented. An abstract stimulus is a named condition based on the prior history and the current stimulus. For example, for a combination safe it might make sense to define the following abstract stimuli:

- If three correct combination digits have been entered, and the current stimulus is the fourth correct combination digit, then this is "correct combination," denoted CC.

- If three combination digits have been entered, and the current stimulus is the fourth digit, and the four digits entered are *not* the correct combination, then this is "incorrect combination," denoted IC.

The introduction of this abstract stimuli CC and IC allows one to discard the individual combination digit entries from the enumeration and instead enumerate with the abstract stimulus CC. For example, the history "ON Lock 1 2 3 4" might correspond to the abstract sequence "ON Lock CC." The abstract history does not replace the original history, it is simply a different way to view the history which omits details in a referentially-transparent way. Appeal can be made to individual digit presses when necessary, for example using predicates [6].

For the calculator one could define an abstract stimulus N which applies "if the current stimulus is a digit and the prior stimulus was not a digit." This simple abstract stimulus allows one to change the view during enumeration from histories of the form "ON 1 2 + 3 4 =" to abstract histories of the form "ON N + N =." Further, one can now equate the history "ON N + N +" and "ON N +," resulting in less work during enumeration.

Abstractions are introduced to solve problems encountered during enumeration. Thus one discovers and documents abstractions while performing sequence enumeration.

5.3 Sequence Enumeration Example

The following miniature example is adapted from [6]. The requirements for a simple combination safe are as follows:

1. The combination consists of four digits (0-9) which must be entered in the correct order to unlock the safe. The combination is fixed in the safe firmware.
2. Following an incorrect combination entry, a "clear" key must be pressed before the safe will accept further entry. The clear key also resets any combination entry.
3. Once the three digits of the combination have been entered in the correct order, the safe unlocks and the door may be opened.
4. When the door is closed, the safe automatically locks.

5. The safe has a sensor which reports the status of the lock.
6. The safe ignores keypad entry when the door is open.
7. There is no external confirmation for combination entry other than unlocking the door.
8. It is assumed (with risk) that the safe cannot be opened by means other than combination entry while the software is running.

The system boundary consists of the interfaces with external power, the keypad, the door sensor, and the lock actuator. The stimuli are digit presses (0-9), pressing the clear key (Clear), and closing the door (Door). In addition, the power on event is a stimulus. At power on, the door sensor can be read, giving the stimuli power on with door locked (PL), and power on with door unlocked (PU). Power off is not a stimulus, because the software cannot observe the event.

Enumeration of individual digit presses is an inefficient way to explore the specification of this system. Therefore the abstract stimuli CC and CI defined previously will be used. The following enumeration is obtained.

History	Response	Equivalence	Trace
CC	Illegal		9
CI	Illegal		9
Clear	Illegal		9
Door	Illegal		9
PL	Null		5
PU	Null		5
PL CC	Unlock	PU	1,3,7
PL CI	Null		1,2,7
PL Clear	Null	PL	2,7
PL Door	Illegal		8
PL PL	Null	PL	5,10
PL PU	Null	PU	5,10
PU CC	Null	PU	6
PU CI	Null	PU	6
PU Clear	Null	PU	6
PU Door	Lock	PL	4
PU PL	Null	PL	5,10
PU PU	Null	PU	5,10
PL CI CC	Null	PL CI	2,7
PL CI CI	Null	PL CI	2,7
PL CI Clear	Null	PL	2,7
PL CI Door	Illegal		8
PL CI PL	Null	PL	5,10
PL CI PU	Null	PU	5,10

Note that the histories of the longest length are all either illegal or reduced to a previous sequence; the enumeration is complete. Further, the following additional requirements are discovered during the enumeration process.

9. Histories with stimuli prior to system initialization are illegal by system definition.
10. Re-initialization (power-on) makes previous history irrelevant.

5.4 Sequence Analysis and the State Box

The enumeration is organized to facilitate systematic discovery of the correct behavior of a system in all scenarios of use, but not necessarily in the best form for further development. The results from the enumeration must be re-organized to obtain the state box. This is done via sequence analysis.

The unreduced, legal histories in the enumeration are called *canonical histories*, and they represent the states of the system. In the above enumeration, the following histories are canonical.

- PL
- PU
- PL CI

The empty history (the history of no events) is taken to represent the initial conditions of the system. Then each history of length one is considered, and the question is asked "how does processing the given stimulus change the conditions?" This leads to information about the state prior to the stimulus and conditions after the stimulus. This process is then repeated with the histories of length two (which must be single-stimulus extensions of canonical histories of length one), etc., until all sequences have associated conditions.

One possible outcome of the sequence analysis for the three canonical sequences is the following.

Sequence	Door	Combination
Empty	unknown	
PL	locked	none
PU	unlocked	
PL CI	locked	bad

Now conversion of the enumeration into a state box is reduced to bookkeeping. Each row of the enumeration can be viewed as a canonical history, a single-stimulus extension, a response, and a new sequence (either the equivalent sequence, or the full sequence itself if no equivalence is noted).

As an example, consider the stimulus CC. First all rows ending in this stimulus are extracted from the enumeration.

History	Response	Equivalence	Trace
CC	Illegal		9
PL CC	Unlock	PU	1,3,7
PU CC	Null	PU	6
PL CI CC	Null	PL CI	2,7

The illegal sequence CC can be omitted. Now the last stimulus (CC) is dropped, and each history is replaced with its conditions from the sequence analysis. The columns are re-labeled, and the state box table is obtained.

Current State	Response	Changed State	Trace
Door=locked and Combination=none	Unlock	Door=unlocked	1,3,7
Door=unlocked	Null	No change	6
Door=locked and Combination=bad	Null	No change	2,7

Note that this table is correct *by construction*, and that it is traced to the requirements as a side effect of its construction.

5.3 Implementation

The components of the state box must be allocated against whatever software architecture has been defined. Specifically, the following must be defined in terms of the implementation.

- How each stimulus is gathered.
- How each response is generated.
- How each item of state data is implemented.

In addition, if there are any abstractions remaining, these must be removed at this point by specifying an implementation. Any of these issues may have considerable complexity, and one may choose to iterate the black, state, and clear boxes for the chosen item. For example, generating a particular response may require considerable calculation.

After implementations are chosen for each component of the state box, one can write an implemented state box, in which the cells are replaced with the source code which implements the cell. This helps organize the verification problem (see the next section). Each cell of the implemented state box is verified against the corresponding cell of the original state box.

For example, the following might be the implemented state box based on the previous state box specification.

Current State	Response	Changed State	Trace
`door && !combobad`	`*(LOCK)=0;`	`door=0`	1,3,7
`!door`			6
`door && combobad`			2,7

The contents of the implemented state box tables are inserted into the software architecture, yielding the clear box as executable code. The above implemented state box table might result in the following code.

```
/* Correct combination entered.  See requirements 1,3,7. */
if (door && !combobad) {
    *(LOCK)=0;
}
```

Cleanroom methods enforce completeness and precision in specification and design. With system functionality and structure well defined and understood, subsequent verification and certification efforts become more effective and efficient than otherwise possible.

6. Cleanroom Correctness Verification

Cleanroom development depends on verification in special team inspections to achieve functional correctness of software increments. The mathematical foundations of functional verification arise from a view of the sequential logic of programs as implementations of mathematical functions or relations. Such functions need not be numerical, of course, and most programs do not define numerical functions. But for every legal input a program directs the computer to produce a unique output, whether correct as specified or not. And the set of all such input, output pairs is a mathematical function. With these mathematical foundations, software development becomes a process of constructing rules for functions that meet required specifications, which need not be a trial-and-error process.

A program or program part defines a single, possibly complex function. In program design, that function is expressed in control structures and sub-functions, which are in turn expressed in control structures and sub-functions, etc., continuing in this manner until statements of the programming language are reached. This process produces an algebraic structure of single entry/single-exit sequenced, nested, and iterated control structures, each of which is a refinement of a sub-function documented with it in the design. As noted earlier, while sizable programs can embody an essentially infinite number of execution paths, they are composed of a finite number of control structures, each of which can be verified in team inspections in a finite number of steps. These required steps are defined by correctness questions (based on a correctness theorem) as shown in Table 1 for representative structures. The table also defines the function equations that are the basis for the correctness questions, expressed in terms of function composition, case analysis, and for the whiledo, composition and case analysis in a recursive equation based on the equivalence of an iteration control structure and an ifthen control structure. In the table, P represents the control structure, f represents the intended function, g and h represent sub-functions, p represents a predicate, square brackets represent the function of the enclosed program, "|" represents the "or" operator, "o" represents the composition operator, and "I" is the identity function.

In spite of the experiences and assumptions of this first human generation of software development, there is nothing experimental about program behavior except its invention by people. As mathematical artifacts, programs admit mathematical inspection and verification of whether they meet mathematical specifications. Of course mathematics does not mean numerical and most programs are not strictly numerical. A simple sort program performs a mathematical function in mapping a set of items into a sorted sequence of those same items. In this first human generation of programming, programs are drafted, tested, fixed, retested, refixed, etc., as an experimental activity. In this process, intellectual control is lost, ending with programs people hope are right, but which are frequently not quite right.

Since programs are strict rules for mathematical functions or relations, their correctness can be determined by mathematical inspection and verification against specifications. Just as place notation and long division made correct operations in arithmetic more practical, methods exist in software engineering to make functional correctness verification a practical reality. The mathematics are relatively simple, more like long division than nuclear physics. These simple mathematics are applied over and over in verifying large programs. Good program organization, both in control and data, make this process possible and practical for disciplined software engineers. Experience shows that failures detected following verification are very different from failures following debugging. Verification failures are usually due to simple coding errors, and very seldom due to deeper problems with design. These errors are easily found and fixed in early testing, with few, if any, subsequent failures.

Structure	Program	Function Equation	Correctness Question
Sequence	P: [f] do g; h enddo	f = [P] = [g;h] = [g] o [h]	For all possible arguments, does g followed by h do f?

Ifthenelse	P: [f]
	if p
	then
	g
	else
	h
	endif
		f = [P] = [if p then g else h endif] =
		([p] = true → [g] |
		[p] = false → [h])
			For all possible arguments,
			whenever p is true,
			does g do f,
			and whenever p is false,
			does h do f?

Whiledo	P: [f] while p do g enddo	f = [P] = [while p do g enddo] = [if p then g; while p do g enddo endif] = [if p then g; f endif] = f = ([p] = true → [f] o [g] | [p] = false → I)	For all possible arguments, is termination guaranteed, and whenever p is true, does g followed by f do f, and whenever p is false, does doing nothing do f?

Table 1. Control Structure Semantics and Verification Conditions

The control structures of Table 1 are expressed in design language form, but are easily written in Java, C, or any other procedural language. The function f is independent of the programming language, and can be expressed in a variety of forms, from natural language to mathematical notation. As can be seen in the Table, sequence verification requires checking one condition (composition of sequence parts), ifthenelse verification requires checking two conditions (iftest true and false cases), and whiledo verification requires checking three conditions (termination, plus whiletest true and false cases).

Figure 2 enumerates the correctness conditions required for verification of a miniature program (and illustrates recording of intended functions, shown enclosed in square brackets). Fifteen conditions must be checked, easily accomplished in a few minutes in a team inspection. If, say, five people check each condition, a simultaneous reasoning mistake by all five would be required for a faulty verification, an unlikely occurrence.

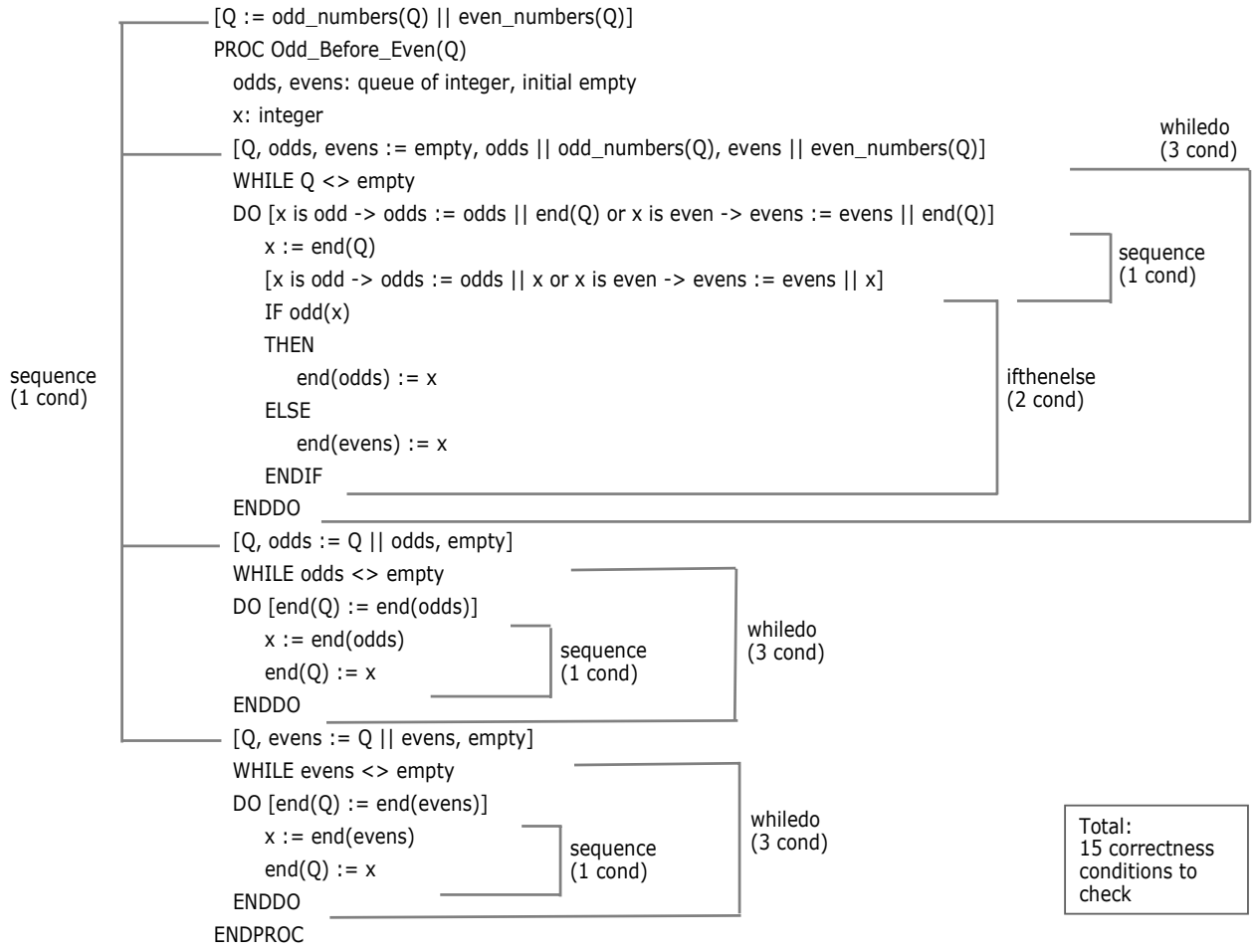


Figure 2. Correctness Conditions for a Miniature Program

7. Cleanroom Certification of Software Fitness for Use

Software specifications deal with functional behavior and performance. Functional behavior is ordinarily decomposed into various sub-functions in ways understandable by users, and often obtained from users as requirements. Performance will usually affect design in fundamental ways, and expected usage of the software will have critical impacts on performance issues. For example, a database system, with significantly more querying than data addition or deletion, may call for a design with high-performance queries at the expense of data addition and deletion performance, but such a design can be entirely unsatisfactory with different usage. Thus, expected usage statistics can play a key role in software system design.

There is another critical use for usage statistics as part of software specifications; to permit the certification of software. Software behavior depends not only on how correct the software is but also on how it is used. For every possible state of internally stored data, any command and input data is handled either correctly or incorrectly, denoted by a failure in the latter case at some level of seriousness.

With a statistical usage specification, the probability of each selection of user (person or program) commands and input data will be known. Markov models can be used to represent statistical usage specifications as directed graphs that step from one usage state to another according to anticipated

transition probabilities. Any such traversal through the model is a test case for the software. Sampling the model to produce test cases according to usage probabilities creates a test suite that represents anticipated field usage. These cases can be executed and the results compared to the functional specification, which defines the new internal state for each command and input as well as the response to the user. When such testing is carried out under a formal statistical protocol, the results predict eventual field experience with the software.

7.1 Testing for Software Certification

In the Cleanroom process, assuring quality of the resulting system is a lifecycle activity that encompasses all specification, design, verification, and testing. The purpose of Cleanroom testing is to demonstrate a certified level of performance based on expected use.

Software is either correct or incorrect with respect to a well-defined specification, in contrast to hardware, which is reliable to some level in performing a design that is assumed to be correct. For small programs, it may be possible to exhaustively test the software to determine correctness. Even then, failures can be overlooked from human fallibility. But software of any size or complexity can only be tested partially, and typically a very small fraction of possible inputs are actually tested. At first glance, the fractions are so small for systems of ordinary size that the task of testing looks impossible. But when combined with mathematical verification, getting correct software is indeed possible.

Certifying the correctness of such software requires statistical testing with inputs characteristic of actual usage. For interactive software, the statistical correlation of successive inputs must be treated, as well. If any failures arise in testing or subsequent usage, the certification must be redone. So certifying the correctness of software is an empirical process that is bound to succeed if the software is indeed correct and may succeed for some time if the software is incorrect.

While possibly frustrating at first glance, this is all humans can assert about the correctness of software. In both verification and testing, human fallibility is present. But on second glance, the sequential history of certification efforts provides a human basis for assessing the quality of the software and expectations for achieving future correctness.

7.2 The Software Certification Process

Certification of software on a scientific basis requires a statistical usage specification as well as functional and performance specifications. As noted, a statistical usage specification is typically given in a Markov model as possible use transitions plus probabilities that the transitions will occur. Thus, frequent and infrequent uses have higher and lower probabilities, respectively. If necessary, infrequent uses with high consequences of failure, for example, invoking the code for emergency reactor shutdown, can be modeled for specific certification by adjusting relative probabilities. The testing must be carried out by statistical selection of tests from these usage specifications. Some uses of the software may be much more important than other uses, and the statistical selections can be given in various levels of stratified sampling. Thus, not only basic statistical usage is to be defined, but the relative importance of correctness for each usage. An extreme form of stratified testing is important cases chosen with probability 1.0.

The balance between a few important cases and general cases takes good engineering design in the best use of test capabilities that is seldom explicitly addressed today. The number of tests is a matter of test design, from which the reliability of software that passes the test design can be calculated. This is new information that is often not known today until the software is put to actual use. Without usage specifications, testing can be inadequate and result in a surprising number of failures during field use of software because it is used differently than expected.

For sizable systems, statistical testing is a repeating, stepwise process, each step carried out when a new accumulation of increments is delivered from the developers. If a failure is found in testing, the software should be returned to the developers for correction. When the required correction is identified and made, testing can be redone. Measurements such as Time To Failure (TTF) can be recorded for each failure discovered. Time Without Failure (TWF) can be tracked when no failures have appeared. This TWF can be tracked after the software is distributed to users as part of the characterization of its correctness.

7.3 Stratification Planning

First a test boundary must be identified. As with the system boundary, this is a list of interfaces, but it may be different from the system boundary. For example, the system boundary may include several hardware interfaces, but for the purpose of testing it may be impractical to drive these interfaces directly. For each interface identified in the test boundary, it must be possible to control all inputs (in order to execute the test), and to record all outputs (in order to correctly detect failure and success).

Since testing is based on expected usage, the factors which affect use of the system must be identified. Typically, one identifies the following.

- Users. Who or what will issue inputs to the system under test?
- Uses. What are the purposes for which the system will be used?
- Environments. What are the environments in which the system will be used?

The primary classes of users, uses, and environments are identified. A combination of user, use, and environment which makes sense for testing is called a *stratum*. In the stratification plan all strata of interest are identified. Finally, the percentage of overall test budget which is to be allocated to the stratum is identified. The stratification plan may also identify other tests to be executed to satisfy contractual or other requirements.

7.4 Certification as Statistical Experiment

Even for simple software systems, the domain of possible tests is quite large. For example, software which only multiples two 64-bit floating point numbers has an input space of approximately 3.4×10^{38} inputs. When inputs can be sequenced to obtain different results, the domain of possible tests becomes infinite. Thus all testing is sampling.

Treating software testing as a statistical experiment requires that one construct a model of the population; in this case a model which denotes the software uses of interest and their expected relative weights. The sample may then be drawn based on the model, and evaluated with respect to the model.

Markov chain usage models have proven themselves effective in specifying the population of tests and their relative weights. Further, Markov chain usage models have a significant analytical capabilities [7]. One can perform a static analysis of the model to determine if it matches anticipated usage, and can analyze the results of testing to determine statistics such as reliability, TTF, and TWF.

Every Markov chain usage model must have a corresponding definition of use. This is a general statement of what a single use (or test) will contain. For example, a use might be defined for a telephone as starting with the phone idle and on-hook, and ending with the phone again idle and on-hook after having been successfully connected in a call at least once. From the definition of use, one identifies two special states, called the *source* and the *sink*. The source state is the unique start state for testing, while the sink state is the unique terminating state for the test. Both states must be verifiable. This is to ensure that the tests are independent. The remainder of the Markov chain is created to satisfy the definition of use. A single use (or test) is a path from the source to the sink, and it consists of a sequence of usage events to be executed

against the system under test. For example, a Markov chain implementing the definition of use given above is shown in Figure 3 as a directed graph with transition probabilities.

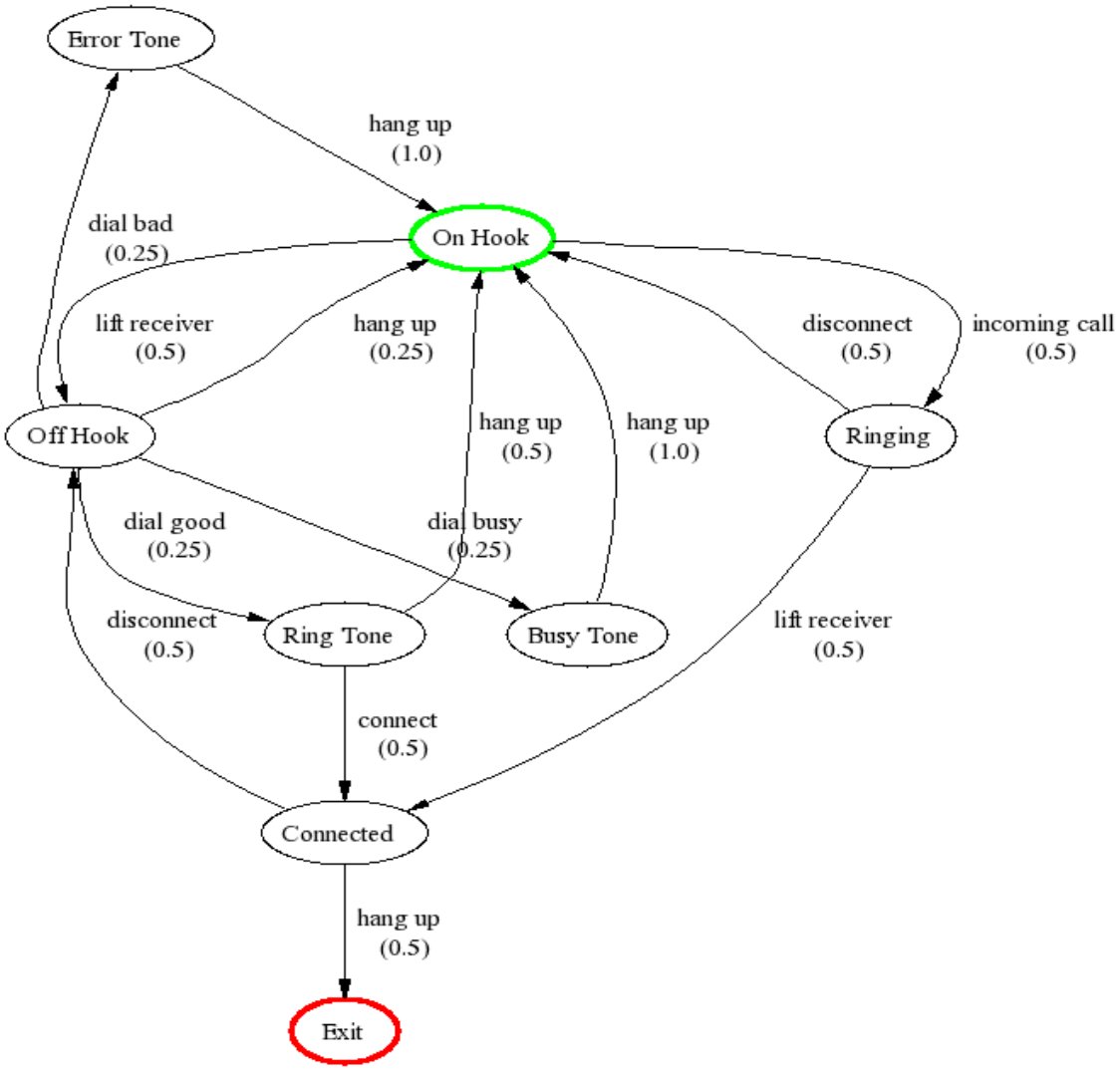


Figure 3. Telephone Markov Model

Analysis of the model reveals that a test consists of 18 events, on average, with a standard deviation of about 8 events. Other statistics include:.

- An incoming call occurs in about 94% of tests in the long run.
- On average the phone will be connected in a call twice during a single test.

The statistics are used to correct the model so that it more closely matches the expectation of use.

One test case which can be generated from the phone model is lift receiver, dial busy, hang up, incoming call, lift receiver, hang up. This sequence of events would be executed against the system under test, and the response to each event recorded. One could then verify that each response was correct with respect to the system's specification.

Tests may be generated in a variety of ways.

- Tests may be generated randomly, conditioned by the probabilities in the model.
- Tests may be generated in order by weight, so that the highest-probability test case is generated first, the second-highest probability test is generated next, etc.
- One or more tests can be generated which cover the model (visit all arcs) with the smallest number of events. This is called the minimum coverage set.

Delivered software and the Markov chain usage model must match one another. It must be possible to execute the tests generated from the model against the software under test. In order to gain confidence that this can be done, one typically executes the minimum coverage set, as this requires executing each usage event from every state of the usage model.

After tests have been generated and executed, and the results of the test recorded, one can use the information to generate quality estimates such as reliability, TTF, and TWF [7]. This information can also be used to determine when to stop software testing [8].

8. Cleanroom Reverse Engineering

Cleanroom foundations provide a basis for reverse engineering of existing programs, for example, the source code of acquired components, to improve understandability and analyze functionality. First, a structure theorem defines a simple stepwise process for transforming the logic of poorly structured programs into structured form to increase their understandability. Second, the Cleanroom function equations of control structures defined in Table 1 for correctness verification can also serve as the starting point of a stepwise process for extracting and documenting the as-built functional specifications of existing programs, that is, how programs transform inputs into outputs in all circumstances.

Function extraction as defined by these foundations can be usefully performed as a manual process. However, automation possibilities are of substantial interest, because traditional software engineering provides no practical means to fully evaluate the behavior of programs. In this case, "evaluation" means understanding full functional behavior, whether right or wrong, intended or the result of malicious intervention. In today's state of art, a software engineer cannot say for sure what a sizable program does in all circumstances of use without an impractical expenditure of time and effort. The result is often unknown functionality in programs available for malicious exploitation.

Understanding program behavior today is an error-prone, resource-intensive process carried out in human time scale; nevertheless, it is essential for uncovering security gaps and vulnerabilities. And because attackers can make malicious modifications to programs at any time, the task of behavior discovery never ends. Sizable programs are hard to understand because they contain an intractable number of execution paths, any of which can contain errors or security exposures. Faced with massive sets of possible executions, programmers can often do no more that gain a general understanding of mainline behavior.

While automation of function extraction would help address these problems, computation of program behavior is a difficult problem that poses many theoretical and engineering challenges. It turns out that the function equations of Cleanroom illuminate a challenging, but feasible, strategy for automating function extraction, with the opportunity to move from an uncertain understanding of program behavior derived in human time scale to a precise understanding computed in CPU time scale.

As noted above, **t**he function-theoretic model treats program control structures as implementations of mathematical functions or relations, and programs are composed of a finite number of control structures. This finite property of program logic viewed from the perspective of function theory opens the possibility of automated calculation of program behavior. Every control structure in a program has a non-procedural

behavior signature that defines its net functional effect. Behavior signatures can be extracted and composed with others in a stepwise process based on an algebra of functions that traverses the control structure hierarchy. The resulting behavior signature of an entire program represents the specification that it implements. This specification coalesces and aggregates full functional behavior, including the behavior of any vulnerabilities or malicious code, no matter how distributed or disguised in the program text. The theoretical challenges to automated function extraction may have acceptable engineering solutions. For example, while no general theory for loop behavior calculation can exist, pattern recognition can help provide an engineering approach.

It is a formidable task to achieve security and reliability goals for systems without knowing what their programs do in all circumstances. In the current state of practice, this knowledge is accumulated in bits and pieces from specifications, designs, code, and test results. Ongoing program maintenance and evolution limit the relevance of even this hard won but perishable knowledge. But programs are mathematical artifacts subject to mathematical analysis. While human fallibility is still present in interpreting the analytical results, there can be little doubt that routine availability of calculated behavior would substantially reduce errors, vulnerabilities, and malicious code in software, and make intrusion and compromise more difficult and detectable. Furthermore, broader questions about security capabilities for authentication, encryption, filtering, etc., are in large part questions about the behavior of programs that implement these functions. And beyond security considerations, behavior calculation will impact many software engineering activities, from specification and design to testing and maintenance.

9. The Future of Cleanroom

The need for reliable software systems will only increase as society's dependency on information technology becomes more pervasive. At the same time, competitive pressures are forcing software development to become more responsive and productive. Cleanroom development produces high-quality software with high productivity at reduced cost by eliminating debugging and rework and reducing time and effort required for testing. It allows project management to reduce risk by linking resource consumption and earned value through incremental development. And it provides customers with the surety of valid quality certification at delivery. Cleanroom application is growing as these benefits become more widely recognized.

Many Cleanroom processes are automatable. For example, many operations in correctness verification can be carried out automatically, and others will benefit from interactive cooperation between human analysis and machine guidance and recordkeeping. As Cleanroom matures, more tool capabilities can be expected to emerge.

10. References

[1] H. Mills and R. Linger, "Cleanroom Software Engineering," *Encyclopedia of Software Engineering, 2nd ed.,* (J. Marciniak, ed.), John Wiley & Sons, New York, 2002.

[2] S. Prowell, C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering: Technology and Process,* Addison Wesley, Reading, MA, 1999.

[3] R. Linger, "Cleanroom Process Model," *IEEE Software,* IEEE Computer Society, March 1994.

[4] G. Broadfoot and P. Broadfoot, "Academia and Industry Meet: Some Experiences of Formal Methods in Practice," *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, Chiang Mai, Thailand, December 2003, IEEE Computer Society.

[5] M. Pleszkoch and R. Linger, "Improving Network System Security with Function Extraction Technology for Automated Computation of Program Behavior," *Proceedings of Hawaii International Conference on System Sciences-38 (HICSS-38),* Hawaii, January, 2004, IEEE Computer Society Press.

[6] S. Prowell and J. Poore, "Foundations of Sequence-Based Software Specification," *IEEE Transactions on Software Engineering*, v. 29, n. 5, May 2003.

[7] S. Prowell, "Computations for Markov Chain Usage Models," University of Tennessee Computer Science Technical Report UT-CS-03-505, 2003.

[8] S. Prowell, "A Cost-Benefit Stopping Criterion for Statistical Testing," *Proceedings of Hawaii International Conference on System Sciences-38 (HICSS-38),* Hawaii, January, 2004, IEEE Computer Society Press.

Security and Capability Maturity Models

Joe Jarzombek, PMP
Deputy Director for Software Assurance
Information Assurance Directorate
Office of the Assistant Secretary of Defense
(Networks and Information Integration)
Joe.Jarzombek@osd.mil

[Process models provide goal-level definitions for and key attributes of specific processes (for example, security engineering processes), but do not include operational guidance for process definition and implementation – they state requirements and activities of an acceptable process but not how to do it. Process models are not intended to be how-to guides for improving particular engineering skills. Instead, organizations can use the goals and attributes defined in process models as high-level guides for defining and improving their management and engineering processes in the ways they feel are most appropriate for them. Eds.]

Introduction

Capability Maturity Models (CMMs) are a type of process model intended to guide organizations in improving their capability to perform a particular process. CMMs can also be used to evaluate organizations against the model criteria to identify areas needing improvement. CMM-based evaluations are not meant to replace product evaluation or system certification. Rather, organizational evaluations are meant to focus process improvement efforts on weaknesses identified in particular process areas. CMMs are currently used by over a thousand organizations to guide process improvement and evaluate capabilities.

There are currently three CMMs that address security, the Capability Maturity Model Integration® (CMMI®), the integrated Capability Maturity Model (iCMM), and the Systems Security Engineering Capability Maturity Model (SSE-CMM) . A common Safety and Security Assurance Application Area (similar to a Process Area) is currently under review for the iCMM and CMMI, along with a new Process Area for Work Environment, and the proposed goals and practices have been piloted for use. All of these CMMs are based on the Capability Maturity Model (CMM®)

The iCMM and CMMI have both been in use for more than three years. Some initial evidence exists of processes defined using this model that reduced overall defect content [Goldenson]. Since the Safety and Security Application Area is still under development, no evidence currently exists of reduced security vulnerabilities. However, both the CMMI and the iCMM are based on the CMM and there is evidence showing that higher maturity organizations tend to produce software with fewer defects. Table 1 shows average defect densities by CMM maturity level [Davis].

Table 1: Average Defect Density of Delivered Software

CMM Level	Defect/KLOC
Level 1	7.5
Level 2	6.24
Level 3	4.73
Level 4	2.28
Level 5	1.05

Capability Maturity Model Integration

The Capability Maturity Model Integration® (CMMI®) helps organizations improve their processes. Improvement areas covered by this model include systems engineering, software engineering, integrated product and process development, supplier sourcing, process management and project management.

Further information on the CMMI is available at http://www.sei.cmu.edu.

integrated Capability Maturity Model

The iCMM is widely used in the Federal Aviation Administration. Version 2.0 of the iCMM builds on the CMM and iCMM v1.0 and provides a single model of best practices for enterprise-wide improvement, including outsourcing and supplier management. Version 2 added process areas to address integrated enterprise management, information management, deployment/transition/disposal, and operation/support. It integrates the following additional (beyond the sources for version 1) standards and models: ISO 9001:2000, EIA/IS 731, Malcolm Baldrige National Quality Award and President's Quality Award criteria, CMMI-SE/SW/IPPD and CMMI-A, ISO/IEC TR 15504, ISO/IEC 12207, and ISO/IEC CD 15288.

The iCMM v2 is available at the following web sites: www.faa.gov/aio or www.faa.gov/ipg

Systems Security Engineering Capability Maturity Model (SSE-CMM)

The SSE-CMM® is a process model that can be used to improve and assess the security engineering capability of an organization. The stated purpose for developing the model is that, although the field of security engineering has several generally accepted principles, it lacks a comprehensive framework for evaluating security engineering practices against the principles. The SSE-CMM, by defining such a framework, provides a way to measure and improve performance in the application of security engineering principles.

The model is organized into Process Areas. Each Process Area is comprised of a related set of process goals and activities.

The twenty two Process Areas of the SSE-CMM are:

Administer Security Controls

Assess Impact

Assess Security Risk

Assess Threat

Assess Vulnerability

Build Assurance Argument

Coordinate Security

Monitor Security Posture

Provide Security Input

Specify Security Needs

Verify and Validate Security

Ensure Quality

Manage Configuration

Manage Project Risk

Monitor and Control Technical Effort

Plan Technical Effort

Define Organization's Systems Engineering Process

Improve Organization's Systems Engineering Process

Manage Product Line Evolution

Manage Systems Engineering Support Environment

Provide Ongoing Skills and Knowledge

Coordinate with Suppliers

Version 2 of the Systems Security Engineering Capability Maturity Model (SSE-CMM)® is now an ISO standard and a version 3 is now available. Further information about the model is available at http://www.sse-cmm.org.

Safety and Security Assurance Application Area for CMMI and iCMM

Because of the integration of process disciplines, CMMI and iCMM are used by more organizations than the SSE-CMM; yet the two integrated models have had gaps in their coverage of safety and security. Therefore, organizations within the US Federal Aviation Administration (FAA) and US Department of Defense (DoD) are sponsoring a joint effort with the objective of identifying best safety and security practices for use in combination with the two integrated CMMs: FAA-iCMM v2.0, and CMMI V1.1. This project is being co-managed by FAA Chief Engineer for Process Improvement and Deputy Director for Software Assurance in DoD, with broad participation from government and industry.

Both the iCMM and the CMMI provide an excellent technical and process model foundation for safety and security; however, without the proposed application area or Work Environment Process Area, they don't include sufficient focus on safety and

security practices. In order to provide this focus without duplication of material in the existing models, a new model construct, Application Area, was developed by the project. The Safety and Security Application Area (AA) identifies standards-based application practices (APs) expected to be used as criteria in guiding process improvement and in appraising an organization's capabilities for providing safe and secure products and services. These application practices are used in conjunction with Capability Maturity Model Integrated (CMMI) or the FAA integrated Capability Maturity Model (iCMM).

The purpose of Safety and Security Assurance is to establish and maintain a safety and security capability, define and manage requirements based on risks attributable to threats, hazards, and vulnerabilities, and assure that products and services are safe and secure. Its four goals are:

1. An infrastructure for safety and security is established and maintained.
2. Safety and security risks are identified and managed.
3. Safety and security requirements are satisfied.
4. Activities and products are managed to achieve safety and security requirements and objectives.

Source Material selected by experts from safety and security communities of practice to be integrated and incorporated in the AA comprises three safety standards and four security standards.

- For safety:
  - MIL-STD-882C: System Safety Program Requirements
  - IEC 61508: Functional Safety of Electrical/ Electronic/ Programmable Electronic Systems
  - DEF STAN 00-56: Safety Management Requirements for Defence Systems
- For security:
  - ISO 17799: Information Technology - Code of practice for information security management
  - ISO 15408: The Common Criteria (v 2.1) Mapping of Assurance Levels and Families
  - ISO/IEC 21827: Systems Security Engineering (SSE) CMM (v2.0)
  - NIST 800-30: Risk Management Guide for Information Technology Systems

Scope of Safety and Security Assurance Extension to CMMI and iCMM

As described in the draft material currently out for review, the Safety and Security Application Area groups together related application practices (APs) that are considered essential for achieving the requisite outcomes particular to the Safety and Security disciplines. The application practices are implemented by performing practices that are already in process areas of the reference model, with explicit guidance derived from source standards. Thus, this application area provides a guide for identifying which selected process areas and practices in a reference model need to be implemented to address the purpose of safety and security. The application practices also provide additional interpretive guidance for ways that the practices in the reference model might be implemented in the particular context of safety and security.

The new AA was developed in order to make the practice of safety and security in organizations explicitly improvable and appraisable; as such, the safety and security application practices needed to be structured as "expected" practices. Simply adding informative material to existing practices in the reference models would have provided no assurance that safety and security would be included in process improvement or appraisal of capabilities. The AA also provides direct visibility, in a single location, to those practices needed for safety and security assurance.

Achievement of Safety and Security goals can be assessed based on evidence of implemented practices:

AA Goal 1 – An infrastructure for safety and security is established and maintained.

AP01.01. Ensure safety and security awareness, guidance, and competency.

AP01.02. Establish and maintain a qualified work environment that meets safety and security needs.

AP01.03. Establish and maintain storage, protection, and access and distribution control to assure the integrity of information.

AP01.04. Monitor, report and analyze safety and security incidents and identify potential corrective actions.

AP01.05. Plan and provide for continuity of activities with contingencies for threats and hazards to operations and the infrastructure.

AA Goal 2 – Safety and security risks are identified and managed.

AP01.06. Identify risks and sources of risks attributable to vulnerabilities, security threats, and safety hazards.

AP01.07. For each risk associated with safety or security, determine the causal factors, estimate the consequence and likelihood of an occurrence, and determine relative priority.

AP01.08. For each risk associated with safety or security, determine, implement and monitor the risk mitigation plan to achieve an acceptable level of risk.

AA Goal 3 – Safety and security requirements are satisfied.

AP01.09. Identify and document applicable regulatory requirements, laws, standards, policies, and acceptable levels of safety and security.

AP01.10. Establish and maintain safety and security requirements, including integrity levels, and design the product or service to meet them.

AP01.11. Objectively verify and validate work products and delivered products and services to assure safety and security requirements have been achieved and fulfill intended use.

AP01.12. Establish and maintain safety and security assurance arguments and supporting evidence throughout the lifecycle.

AA Goal 4 – Activities and products are managed to achieve safety and security requirements and objectives.

AP01.13. Establish and maintain independent reporting of safety and security status and issues.

AP01.14. Establish and maintain a plan to achieve safety and security requirements and objectives.

AP01.15. Select and manage products and suppliers using safety and security criteria.

AP01.16. Measure, monitor and review safety and security activities against plans, control products, take corrective action, and improve processes.

Other Models

There are many other process and quality improvement models, methods, and practices available. We did not explore these in detail. Some better known ones are ISO 9001 and ISO 9000-3, ISO15504, Total Quality Management, and Six Sigma.

Final Remark

A key point needs to be made about the use of models to guide process improvement and evaluate capabilities. Product evaluation and/or certification processes normally examine the generation of assurance evidence. The application of process models, and the appraisal (assessment/evaluation) conducted as part of an organizational evaluation, not only focuses improvement efforts on weaknesses in particular disciplines or process areas, but also provides confidence in the assurance evidence generation processes that are used in product evaluation and system certification.

References

[Davis] Davis, Noopur, and Mullaney, Julia, "The Team Software Process in Practice: A Summary of Recent Results," Technical Report CMU/SEI-2003-TR-014, September 2003.

[Goldenson] Goldenson, Dennis R. and Gibson, Diane L. "Demonstrating the Impact and Benefits of CMMI", Special Report CMU/SEI-2003-SR-009, The Software Engineering Institute, Carnegie Mellon University, 2003

The Team Software Process[SM] (TSP[SM])

Noopur Davis
Julia Mullaney

February 2004

Table of Contents

List of Figures

List of Tables

Acknowledgements

This paper is derived from the Software Engineering Institute's Technical Report titled "The Team Software Process in Practice: A Summary of Recent Results", CMU/SEI-2003-TR-014, http://www.sei.cmu.edu/publications/documents/03.reports/03tr014.html.

Executive Summary

Most software organizations critically need better cost and schedule management, quality management, and cycle-time reduction. This report demonstrates that teams using the Team Software Process[SM] (TSP) meet these critical business needs by delivering essentially defect[1]-free software on schedule and with better productivity.

The report starts with an overview of the TSP to provide the context for the results reported. These results include the benefits realized by a first-time TSP team, a summary of data from 20 TSP projects in 13 organizations, and stories from people who have used the TSP.

These TSP teams delivered their products an average of 6% later than they had planned. The schedule error for these teams ranged from 20% earlier than planned to 27% later than planned. This compares favorably with industry data that show over half of all software projects were more than 100% late or were cancelled. These TSP teams also improved their productivity by an average of 78%.

The teams met their schedules while producing products that had 10 to 100 times fewer defects than typical software products. They delivered software products with average quality levels of 5.2 sigma, or 60 defects per million parts (lines of code). In several instances, the products delivered were defect free.

The report concludes with an overview of the Team Software Process for Secure Software Development (TSP-Secure).

[SM] Team Software Process, TSP, Personal Software Process, and PSP are service marks of the Software Engineering Institute.

[1] A defect is anything that leads to a fix in a product. A defect may be a requirements defect, design defects, security defects, usability defects, or an implementation defect.

1 Introduction

The success of organizations that produce software-intensive systems depends on well-managed software development processes. Implementing disciplined software methods, however, is often challenging. Organizations seem to know *what* they want their teams to be doing, but they struggle with *how* to do it. The Team Software Process[SM] (TSP[SM]), coupled with the Personal Software Process[SM] (PSP[SM]), was designed to provide both a strategy and a set of operational procedures for using disciplined software process methods at the individual and team levels. Organizations that have implemented the TSP and PSP have experienced significant improvements in the quality of their software systems and reduced schedule deviation [Ferguson 99, McAndrews 00].

The report starts with an overview of the PSP and the TSP to provide a context for the results reported. This is followed by a summary of the performance of more than 20 projects from 13 organizations that have used the PSP and the TSP. The report concludes with a brief overview of the Team Software Process for Secure Software Development.

[SM] Personal Software Process, PSP, Team Software Process, and TSP are service marks of Carnegie Mellon University.

2 TSP Overview

The objective of the TSP is to create a team environment that supports disciplined individual work and builds and maintains a self-directed team. The TSP guides self-directed teams in addressing critical business needs of better cost and schedule management, effective quality management, and cycle-time reduction. It defines a whole product framework of customizable processes and an introduction strategy that includes building management sponsorship, training for managers and engineers, coaching, mentoring, and automated tool support.

The TSP can be used for all aspects of software development: requirements elicitation and definition, design, implementation, test, and maintenance. The TSP can support multidisciplinary teams that range in size from two engineers to over a hundred engineers. It can be used to develop various kinds of products, ranging from real-time embedded control systems to commercial desktop client-server applications.

The TSP builds on and enables the PSP. The PSP shows engineers how to measure their work and use that data to improve their performance. The PSP guides individual work. The TSP guides teamwork and creates an environment in which individuals can use the PSP to excel. Data from early pilots show that the TSP has been successful in addressing critical business needs [Ferguson 99, McAndrews 00].

2.1 History

In the 1980s, Watts Humphrey guided the development of the Capability Maturity Model® for Software (SW-CMM®). An early misperception of SW-CMM by some people was that it did not apply to small organizations or projects. In order to illustrate its application to small organizations, Humphrey took on the challenge to apply the SW-CMM to the smallest organization possible: an organization of a single individual. From 1989 to 1993, Humphrey wrote more than 60 programs and more than 25,000 lines of code (LOC). In developing these 60 programs, Humphrey used all of the applicable SW-CMM practices up through Level 5. He concluded that the management principles embodied in the SW-CMM were just as applicable

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

to individual software engineers. The resulting process was the PSP. He subsequently worked on corporate and academic methods to train others to use the PSP technology.

As engineers started applying their PSP skills on the job, it was soon discovered that they needed a supportive environment that recognized and rewarded sound engineering methods. In many organizations, the projects in crisis receive all the attention. Projects and individuals who meet commitments and do not have quality problems often go unnoticed. Humphrey found that if managers do not provide a supportive environment and do not ask for and constructively use PSP data, engineers soon stop using the PSP. Humphrey then developed the Team Software Process to build and sustain effective teams.

2.2 What Makes PSP and TSP Work

Typical software projects are often late, over budget, of poor quality, and difficult to track. Engineers often have unrealistic schedules dictated to them and are kept in the dark as to the business objectives and customer needs. They are required to use imposed processes, tools, and standards, and often take shortcuts to meet schedule pressures. Very few teams can consistently be successful in this environment. As software systems get larger and more complex, these problems only get worse.

The best projects are an artful balance of conflicting forces. They must consider business needs, technical capability, and customer desires. Slighting any facet can jeopardize the success of the project. To balance these conflicting forces, teams must understand the complete context for their projects. This requires self-directed teams that

- understand business and product goals
- produce their own plans to address those goals
- make their own commitments
- direct their own projects
- consistently use the methods and processes that they select
- manage quality

Figure 1 illustrates how the PSP and TSP build and maintain self-directed teams. Successful self-directed teams require skilled and capable individual team members. Capable team members are critical because each instruction of a software module is handcrafted by an individual software engineer. The engineer's skills, discipline, and commitment govern the quality of that module and the schedule on which that module is produced. In turn, the modules come together to compose software products. Therefore, a software product is a team effort. The product's modules are designed, built, integrated, tested, and maintained by a team of software engineers whose skills, discipline, and commitment govern the success of the project.

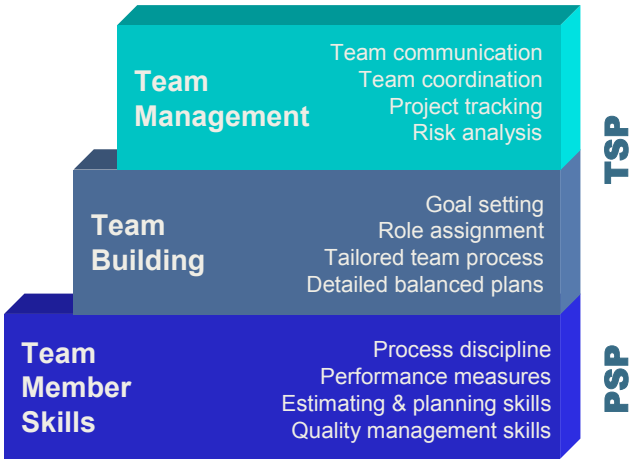


Figure 1: Elements of the PSP and the TSP

The objective of the PSP is to put software professionals in charge of their work and to make them feel personally responsible for the quality of the products they produce. The objectives of the TSP are to provide a team environment that supports PSP work and to build and maintain a self-directed team. PSP and TSP are powerful tools that provide the necessary skills, discipline, and commitment required for successful software projects.

2.3 The PSP

The PSP is based on the following planning and quality principles [Humphrey 00]:

- Every engineer is different; to be most effective, engineers must plan their work and they must base their plans on personal data.
- To consistently improve their performance, engineers must measure their work and use their results to improve.
- To produce quality products, engineers must feel personally responsible for the quality of their products. Superior products are not produced by accident; engineers must strive to do quality work.
- It costs less to find and fix defects earlier in a process than later.
- It is more efficient to prevent defects than to find and fix them.
- The right way is always the fastest and cheapest way to do a job.

Today, most software engineers do not plan and track their work, nor do they measure and manage product quality. This is not surprising, since engineers are neither trained in these disciplines nor required to use them. The dilemma is that until they try using disciplined methods, most software engineers do not believe that these methods will work for them. They won't try these methods without evidence, and they can't get the evidence without trying the methods. The PSP addresses this dilemma by putting an engineer in a course envi-

ronment to learn the methods. The engineers use the methods in the course and can see from their personal and class data that the methods can and do work for them.

The PSP course is composed of ten programming assignments and five reports. The PSP methods are introduced in six upwardly compatible steps, PSP0 through PSP 2.1 (see Figure 2). The engineers write one or two programs at each step and gather and analyze data on their work. Then they use their data and analyses to improve their work.

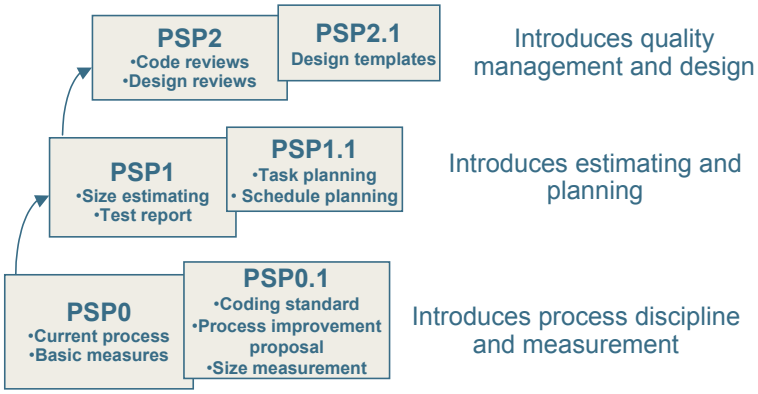


Figure 2: The PSP Course

PSP0 and PSP0.1. Engineers write three programming assignments using PSP0 and PSP0.1. The objective is for the engineer to learn how to follow a defined process and to gather basic size, time, and defect data.

PSP1 and PSP1.1. Once engineers have gathered some historical data, the focus moves to estimating and planning. Engineers write three programming assignments using PSP1 and PSP1.1. Engineers learn statistical methods for producing size and resource estimates, and use earned value for schedule planning and tracking.

PSP2 and PSP2.1. Once engineers have control of their plans and commitments, the focus of the course then changes to quality management. Engineers write four programming assignments using PSP2 and PSP2.1. Engineers learn early defect detection and removal methods and improved design practices.

Mid-term and final reports. After the first six assignments have been completed, engineers write mid-term reports, and after all ten programming assignments have been completed, engineers write final reports. These reports document the engineers' analyses of their performance. Engineers are required to analyze their data to understand their current performance, to define challenging yet realistic goals, and to identify the specific changes that they will make to achieve those goals.

By the end of the course, engineers are able to plan and control their personal work, define processes that best suit them, and consistently produce quality products on time and for planned costs.

In 1997, a study was conducted to analyze the impact of PSP training on 298 software engineers [Hayes 97]. This study found that engineers were able to significantly improve their estimating skills and the quality of the software products they produced. Engineers were able to achieve these notable improvements without negatively affecting their productivity. In terms of product quality and schedule variance, individuals were able to perform at a level that one would expect from a SW-CMM Level 5 organization.

The 1997 study was recently repeated on a much larger data set of over a thousand software engineers. The larger data set represents a more diverse group of instructors, engineers, programming languages, development environments, etc. The purpose of the replication was to demonstrate the statistically significant improvements in estimating and quality practices, i.e., to answer the question, can engineers learn to use their data to significantly improve their performance? The results from this replication are essentially the same as in the original study, with some minor differences. The findings are presented in an SEI technical report [Davis].

2.3.1 PSP Measurement Framework

Engineers collect three basic measures: size, time, and defects. For the purposes of the PSP course, size is measured in lines of code (LOC). In practice, engineers use a size measure appropriate to the programming language and environment they are using; for example, number of database objects, number of use cases, number of classes, etc. In order to ensure that size is measured consistently, counting and coding standards are defined and used by each engineer. Derived measures that involve size, such as productivity or defect density, use new and changed LOC (N LOC) produced only. "New and changed LOC" is defined as lines of code that are added or modified; existing LOC is not included in the measure. Time is measured as the direct hours spent on each task. It does not include interrupt time. A defect is anything that detracts from the program's ability to completely and effectively meet the users' needs. A defect may be a specification defect, a design defect, or an implementation defect. A defect is an objective measure that engineers can identify, describe, and count.

Engineers use many other measures that are derived from these three basic measures. Both planned and actual data for all measures are gathered and recorded. Actual data are used to track and predict schedule and quality status. All data are archived to provide a personal historical repository for improving estimation accuracy and product quality. Derived measures include:

- estimation accuracy (size/time)

- prediction intervals (size/time)
- time in phase distribution
- defect injection distribution
- defect removal distribution
- productivity
- reuse percentage
- cost performance index
- planned value
- earned value
- predicted earned value
- defect density
- defect density by phase
- defect removal rate by phase
- defect removal leverage
- review rates
- process yield
- phase yield
- failure cost of quality (COQ)
- appraisal COQ
- appraisal/failure COQ ratio

2.4 The TSP

The TSP is based on the following principles:

- The engineers know the most about the job and can make the best plans.
- When engineers plan their own work, they are committed to the plan.
- Precise project tracking requires detailed plans and accurate data.
- Only the people doing the work can collect precise and accurate data.
- To minimize cycle time, the engineers must balance their workload.
- To maximize productivity, focus first on quality.

The TSP has two primary components: a team-building component and a team-working or management component. The team-building component of the TSP is the TSP launch, which puts the team in the challenging situation of developing their plan.

"Successful team-building programs typically expose a group to a challenging situation that requires cooperative behavior of the entire group [Morgan 93]. As the group's members learn to surmount this challenge, they generally form a close-knit and cohesive group. The TSP follows these principles to mold development groups into self-directed teams. However, instead of using an artificial situation like rock climbing or white water rafting, it uses the team launch. The challenge in this case is to produce a detailed plan for a complex development job and then to negotiate the required schedule and resources with management."[2]

2.4.1 The TSP Launch

The first step in developing a team is to plan the work, which is done during the TSP launch. The launch is led by a qualified team coach. In a TSP launch, the team reaches a common understanding of the work and the approach they will take, produces a detailed plan to guide the work, and obtains management support for the plan. A TSP launch is composed of nine meetings over a four-day period, as shown in Figure 3.

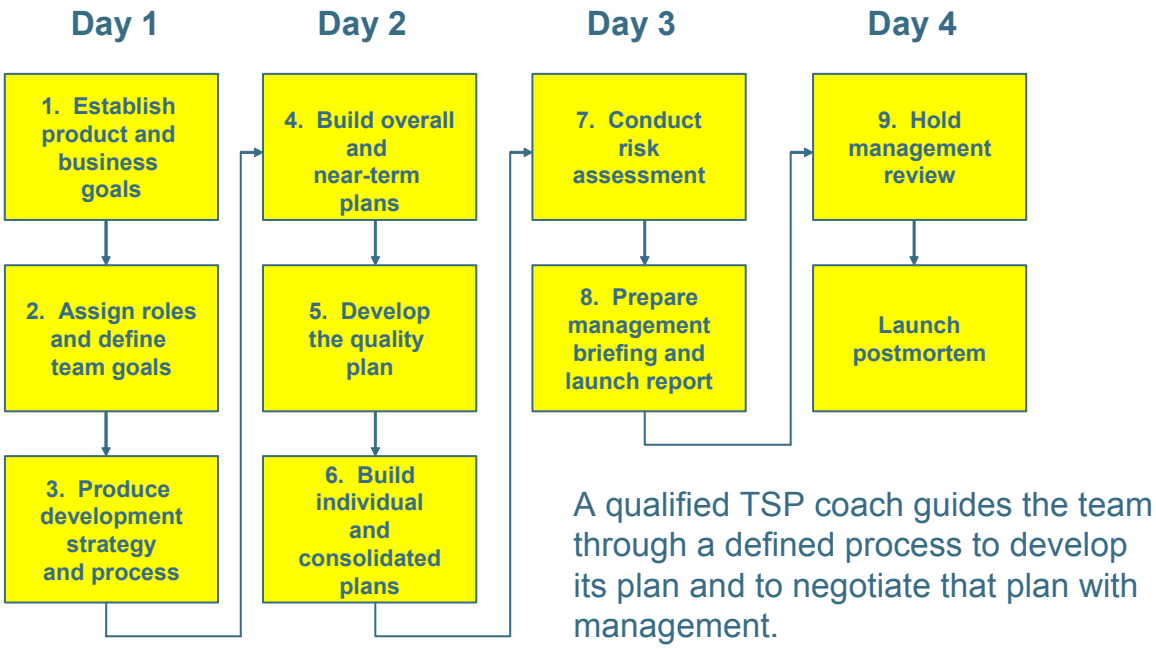


Figure 3: The TSP Launch

The first step in the launch is for the team to understand what they are being asked to do. This is accomplished in meeting 1 by having marketing (or an appropriate customer representative) and management meet with the team. Marketing describes the product needs. Management describes the business needs and any resources and constraints under which the team

---

[2] Personal correspondence with Watts Humphrey.

will have to work. This is also a chance for management to motivate the team. The team has the opportunity to ask any questions they might have about the product or business needs. In the next seven meetings, the team develops an engineering plan to meet the business needs.

In meeting 2, the team sets its goals and organizes itself. The team reviews the business and product goals presented in meeting 1, and derives a set of measurable team goals. Next, the team also decides which team members will take on which routine team management tasks. These tasks are designated by manager roles:

- customer interface manager
- design manager
- implementation manager
- test manager
- planning manager
- process manager
- support manager
- quality manager

Each team member selects at least one role. For teams with more than eight members, roles are shared. With smaller teams, team members may select multiple roles.

In launch meeting 3, the team determines its overall project strategy. The team members produce a conceptual design, devise the development strategy, define the detailed process they will use, and determine the support tools and facilities they will need. They list the products to be produced.

In meeting 4, the team develops the team plan. This is done by estimating the size of the products to be produced, identifying the general tasks needed to do the work and estimating their effort, defining the tasks for the next development cycle to a detailed work-step level, and drawing up a schedule of the team's availability week by week through the completion of the project.

In meeting 5, the team defines a plan to meet its quality goals. The team does this by estimating the number of defects injected and removed in each phase and then calculating the defect density of the final product. The team ensures that the tasks needed to achieve its quality goal are included in the team plan. The quality plan provides a measurable basis for tracking the quality of the work as it is done.

In meeting 6, tasks on the team plan for the next cycle of work are allocated to team members, and each team member creates an individual plan. In building their plans, the engineers

refine the team estimates using their own historical data, break large tasks into smaller tasks to facilitate tracking, and refine their hours available per week to work on this project. The team meets again to review the individual task plans and to ensure that the work load is balanced. The individual plans are consolidated into a team plan. The team uses this plan to guide and track its work during the ensuing cycle.

The team conducts a risk assessment in meeting 7. Risks are identified and their likelihood and impact are assessed. The team defines mitigation and contingency plans for high-priority risks. Risks are documented in the team plan and assigned to team members for tracking.

Meeting 8 is used to develop a presentation of the team's plan to management. If the team's plan does not meet management goals, the team includes alternative plans that come closer to meeting management's goals. For instance, the team might be able to meet a schedule by adding resources to the team or by reducing the functionality delivered.

By the end of the launch, the team has formed a cohesive unit and created a plan that balances the needs of the business and customer with a feasible technical solution. The team has agreed on the technical solution that they propose to build and understands how that product will satisfy business and customer needs. The team agrees on the strategy and process for developing the product. The team has a detailed plan that it can use to guide and track the work. Team members all know who is responsible for which tasks and areas. Everyone on the team understands and agrees with the quality goal, and the team can monitor progress against that goal. Finally, the team has explored all of the things that might go wrong and has done its best to mitigate those risks. In short, the TSP launch provides a team with all of the conditions necessary to become a self-directed team.

In meeting 9, the team presents the plan to management for their approval to start the work. The team explains the plan, describes how it was produced (Figure 4), and demonstrates that all team members agree with and are committed to the plan. If the team has not met management's objectives, it presents one or more alternative plans. The principal reason for showing alternative plans is to provide management with options to consider in case the team's plan does not meet the organization's business needs.

Figure 4: The TSP Launch Products

At the end of the TSP launch, the team and management agree on how the team will proceed with the project. The team has a plan it believes in, is committed to, and can track against. The launch not only creates a winning plan, it builds a cohesive team.

The TSP includes guidance for ensuring that the energy and commitment from a TSP launch are sustained as the team does its work. A TSP coach works with the team and the team leader to help the team to collect and analyze data, follow the process defined by the team, track issues and risks, maintain the plan, track progress against goals (especially the team's quality goal), and report status to management.

2.4.2 TSP Measurement Framework

The TSP uses the same basic measures of the PSP—size, time, and defects—and adds task completion dates. For all measures, planned and actual data are collected at the individual level. The TSP measurement framework consolidates individual data into a team perspective. The data collected are analyzed weekly by the team to understand project status against schedule and quality goals. The TSP measurement framework also makes available other views of the data, such as by product or part, phase, task, week, day, etc. Personal and team data are archived to provide a repository of historical data for future use.

The team conducts weekly meetings to report progress against their plans and to discuss team issues. They also use their TSP data to make accurate status reports to management on a regular basis. Because management can rely on the data, management's job changes from continuously checking project status to ensuring that there are no obstacles impeding the team's progress. This also allows management to make sound business decisions, since they are based on accurate engineering data. For example, when management is confident in the

team's estimate, management can decide how to allocate resources to obtain a schedule that best meets the business needs. When a team commitment is in jeopardy, the team solves the problem or raises the issue with management as early as possible. In all cases and at all levels, decisions are made based on data.

2.4.3 The TSP Introduction Strategy

The SEI has been transitioning TSP into organizations since 1997 and has gained significant experience with issues surrounding the introduction of this technology. Based on these experiences, the SEI has defined an introduction strategy (Figure 5) and has developed supporting materials to facilitate the implementation of that strategy.

The introduction strategy starts with trial use. The TSP is first piloted on several small projects to evaluate both the transition approach and the impact of TSP on the organization. The pilots also build the understanding, sponsorship, and support needed for broad acceptance of the TSP in the organization.

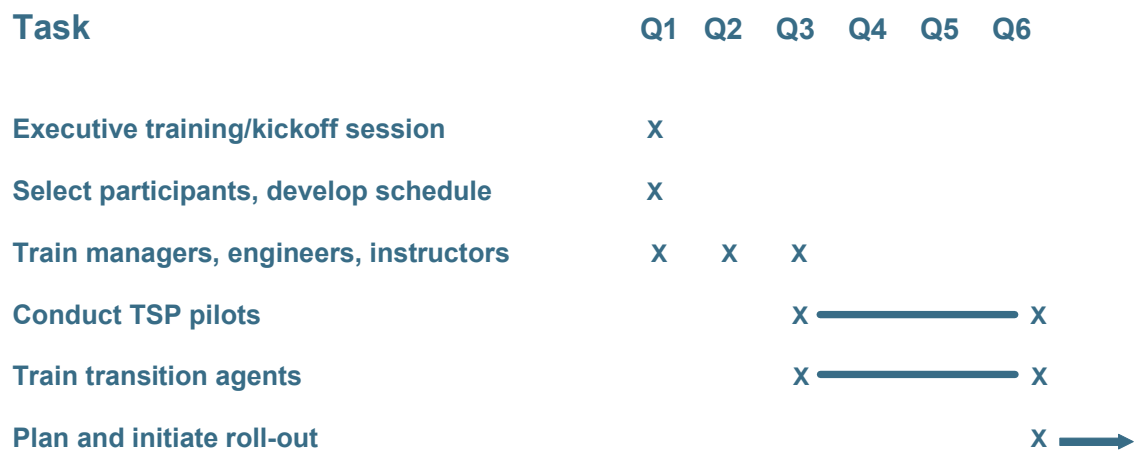


Figure 5: TSP Introduction Timeline

All team members and all of their management are trained prior to the start of the pilot effort. The senior management attends a one-and-a-half-day executive seminar and planning session; the middle and line management attend three days of training; the engineers complete the two-week PSP for Engineers course. The pilot teams are then started with a launch, and they begin to use the TSP process as they do project work. Pilot projects can rapidly demonstrate the benefits of using the TSP, and results from the pilot projects can be used to tailor and improve both the TSP and the introduction strategy.

3 TSP Results

3.1 Data Source

The data summarized in this section come from all TSP presentations developed for the Software Engineering Process Group (SEPG) conferences (http://www.sei.cmu.edu/sepg) and the SEI Software Engineering Symposiums for the years 2001 through 2003 [Ciurczak 02, Davis 01, Janiszewski 01, Narayanan 02, Pracchia 03, Riall 02, Serrano 03, Schwalb 03, Sheshagiri 02, Webb 02].[3] Detailed data submitted to the SEI by the teams represented in those presentations was also examined. The data presented here represent thirteen organizations and over twenty projects from these organizations. Some organizations presented summary data from more than one project without specifying the number of projects, so the exact number of projects could not be determined.

1. ABB, Inc.
2. Advanced Information Services
3. Bettis/KAPL
4. Cognizant Technology Solutions
5. Electronic Brokering Services (EBS) Dealing Resources, Inc.
6. Hill Air Force Base
7. Honeywell
8. Microsoft Corporation
9. Naval Air Warfare Center
10. Quarksoft, S.C.
11. SDRC
12. United Defense, LP
13. Xerox

[3] Also Ciurczak, John, "The Quiet Quality Revolution at EBS Dealing Re-sources, Inc.", Strickland, Keith, "The Road Less Traveled" and Webb, Dave, "Implementing the Team Software Process." Submitted for presentation at the Software Engineering Institute's Software Engineering Symposium, 2001.

3.2 Results

The data presented here are from a diverse group of organizations. Product size range is from 600 LOC to 110,000 new and changed LOC produced, team size range is from 4 team members to 47 team members, and project duration range is from a few months to a couple of years. Application types include real-time software, embedded software, IT software, client-server applications, and financial software, among others. Several programming languages and development environments were used (mostly third and fourth generation languages and development environments). We did not attempt to classify the data based on any of these differences. Instead, we gathered all the measures reported for each organization and calculated the range and average of the values reported. The ranges and averages do not include data from every project, as not all organizations reported the same measures.

We have also tried to compare the TSP projects presented here with typical projects in the software industry. This comparison is rather difficult to make, since there are not much data available on some of the measures tracked in the TSP. For schedule data, we used the Standish Group Chaos Report.[4] For time-in-phase data, we used several sources, including several estimation models, data from the NASA Software Engineering Laboratory [SEL 93], and pre-TSP data from some of the organizations we have worked with [Humphrey 02, Jones 95a, Jones 96, Jones 00]. For quality data, we mostly used Capers Jones as our source [Jones 95a, Jones 96, Jones 00], backed by pre-TSP data from some organizations we have worked with, as well as data from Watts Humphrey [Humphrey 02].

Jones uses function points as the size measure for normalizing defects (defects/function point). Since the TSP uses LOC as the default size measure, we had to convert function points to LOC. We used the "backfiring" method he described [Jones 95b] for this conversion. Jones suggests using a default of 80 LOC per function point for third-generation languages, and a default of 20 LOC per function point for fourth-generation languages. However, we chose to be conservative and used a default of 100 LOC per function point, as Jones does when discussing non-specific procedural languages.

3.2.1 Schedule Deviation

A premise of the TSP is to start with the best plan possible, using sound estimating and planning methods, and then update the plan as needed when you learn more about the work, or if the work itself changes. Because of the constant awareness of plan status, and because teams adjust their plans based on the plan status, TSP teams are able to reduce schedule error. The schedule data presented in Table 1 shows that TSP teams missed their schedule by an average of 6%.

---

[4] "CHAOS '94 – Charting the Seas of Information Technology." The Standish Group International, Inc., 1994.

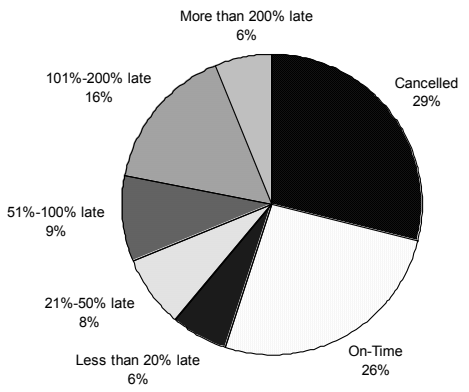
Measure	TSP Projects	Typical Projects (Standish Group Chaos Report)
Schedule error average	6%	
Schedule error range	-20% to 27%	

Table 1: Schedule Deviation

3.2.2 Quality

One reason TSP teams are able to meet their schedule commitment is that they plan for quality and deliver high-quality products to test. This shortens time spent in test, which is usually the most unpredictable activity in the entire development life cycle. The data in Table 2 show that TSP teams are delivering software that is more than two orders of magnitude better in quality than typical projects (0.06 defects/KLOC versus 7.5 defects/KLOC). Products being developed by TSP teams have an average of 0.4 defects/KLOC in system test, with several teams reporting no defects found in system test. TSP teams spent an average of 4% of their total effort in post-development test activities; the maximum effort that any team spent in test was 7%. Similarly, the average percentage of total schedule (project duration in calendar time) spent in post-development test activities was 18%. Typical non-TSP projects routinely spend 40% of development effort and schedule in post-development test activities. The 0.5 average days to test a thousand lines of code is a result of the higher quality of code entering system test. Some teams report that system test time was essentially equal to defect-free test time (time it takes to verify that the software works). Average failure COQ (percentage of total effort spent in failure activities) is much below the 50% typically found in the software industry.

Measure	TSP Projects *Average* *Range*	Typical Projects *Average*
System test defects (defects/KLOC)	0.4 0 to 0.9	15
Delivered defects (defects/KLOC)	0.06 0 to 0.2	7.5
System test effort (% of total effort)	4% 2% to 7%	40%
System test schedule (% of total duration)	18% 8% to 25%	40%
Duration of system test (days/KLOC)	0.5 0.2 to 0.8	NA[5]
Failure COQ	17% 4% to 38%	50%

Table 2: Quality

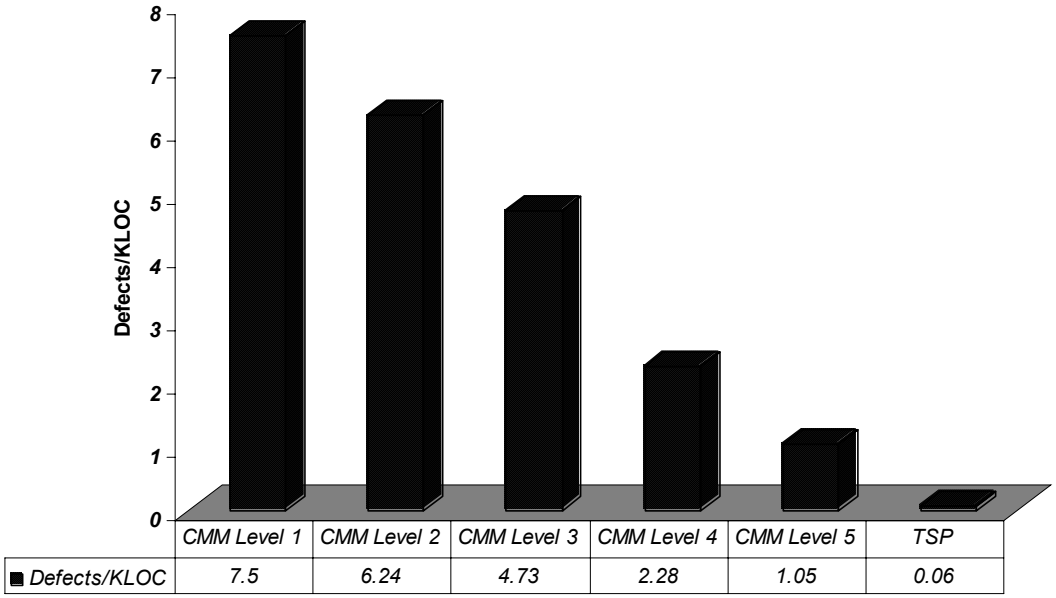


Figure 6: Average Defect Density of Delivered Software

Figure 6 shows the quality of delivered software classified by CMM Level [Jones 00], compared to the TSP teams presented in this report. These data show that TSP teams produced software an order of magnitude higher in quality than projects from organizations rated at CMM Level 5.

---

[5] This data was not available.

Some organizations reported the benefits of the TSP compared to previous projects (Table 3). They reported an average of 8 times reduction in system test defect density when using the TSP. System test duration was reduced an average of 4 times with the TSP: for example, a TSP project spending 0.5 days/KLOC in system test would have been spending 2.0 days/KLOC prior to using the TSP.

Measure	**TSP Projects** ***Average*** ***Range***
System test defect reduction	8 times 4 times to 10 times
System test duration reduction	4 times 2 times to 8 times

Table 3: Reductions In System Test Defects and System Test Duration

3.2.3 Quality is Free

A frequent concern expressed about disciplined methods is the perceived adverse impact on productivity. The data in Table 4 show that TSP projects improve their productivity and at the same time reduce their failure COQ (percentage of total effort spent in failure activities) and their total COQ (percentage of total effort spent in failure and appraisal activities). The main reason for this increase in productivity is the reduced time spent in test because of higher quality products being delivered into test, as shown in Table 2.

Measure	**Average**
Productivity improvement	78%
Failure COQ reduction	58%
Total COQ reduction	30%

Table 4: Improvements in Productivity and Cost Of Quality

3.3 Summarized Project Data

The results summarized in this section are remarkable when compared to typical software projects. The Standish Group reported in 1999 that 74% of all projects were not successful.[6] The Standish group also reported in 1996 that unsuccessful projects accounted for over half

[6] "CHAOS: A Recipe for Success. Project Resolution: The 5-Year View." The Standish Group International, Inc., 1999.

(53%) of total spending on software projects.[7] And in 1994, the same group reported that for the unsuccessful projects, the average cost overrun was 189% and the average time overrun was 222%. Typical projects spend 40% to 60% of total project time on test, and typical defect densities of delivered products range from 1 to 10 defects/KLOC [Humphrey 02].

---

[7] "CHAOS '97 – The Changing Tide." A Standish Group Research Note. The Standish Group International, Inc., 1997.

4 The Team Software Process for Secure Software Development

The Team Software Process for Secure Software Development (TSP-Secure) builds on the TSP by adding secure development practices to the planning, measurement, and quality management practices provided by the TSP.

The problem with producing secure software is that although high quality is a pre-requisite, it is not enough. Testing is not enough, inspections and reviews are not enough, use of tools is not enough, design principles are not enough, and risk management is not enough. First, there is a need for a process that combines all of the above in a planned, managed, and measured framework. The process must use the best software engineering practices that produce near defect-free software, best security practices, best management practices, all supported by a measurement framework. Second, there is a need for security and software engineering education for software developers.

The research objectives of TSP-Secure are to reduce or eliminate software vulnerabilities that result from software design and implementation defects, and to provide the capability to predict the likelihood of latent vulnerabilities in delivered software.

Areas of exploration include vulnerability analysis by defect type, operational process for secure software production, predictive process metrics and checkpoints, quality management practices for secure programming, design patterns for common vulnerabilities, verification techniques, and removing vulnerabilities in legacy software.

TSP-Secure incorporates the following security practices into the TSP: education on common causes of vulnerabilities, intrusion aware design, state machine design and verification, secure inspections and reviews, code analysis tools, security risk analysis and management, and secure testing practices.

TSP-Secure is still under development, but an initial proof-of-concept pilot produced encouraging results. A team of eight developers produced an application with 30,000 new and changed LOC. No security coding defects were found during system test, corporate security audits, or in several months of use since the product was released.

Phase	Post code complete defects
Integration Test	4
System Test	10
User Acceptance Test	3
Security code defects	0
Total Defects	17

Table 5: TSP-Secure Pilot Results

References

All URLs are valid as of the publication date of this report.

[Ciurczak 02] Ciurczak, John. "Team Software Process (TSP) Experiences in the Foreign Exchange Market." *SEPG 2002* (CD-ROM). Phoenix, AZ, February 18-21, 2002. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

[Davis 01] Davis, Noopur; Humphrey, Watts; & McHale, Jim. "Using the TSP to Accelerate CMM-Based Software Process Improvement." *SEPG 2001: Focusing on the Delta* (CD-ROM). New Orleans, LA, March 12-15, 2001. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001.

[Ferguson 99] Ferguson, P.; Leman, G; Perini, P; Renner, S.; & Seshagiri, G. *Software Process Improvement Works!* (CMU/SEI-99-TR-027, ADA371804). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. <http://www.sei.cmu.edu /publications/documents/99.reports/99tr027/99tr027abstract.html>.

[Hayes 97] Hayes, W. & Over, J. W. *The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers.* (CMU/SEI-97-TR-001, ADA335543). Pittsburgh, PA: The Software Engineering Institute, Carnegie Mellon University, 1997. <http://www.sei.cmu.edu/publications/documents/97.reports /97tr001/97tr001abstract.html>.

[Humphrey 95] Humphrey, Watts S. *A Discipline for Software Engineering.* Reading, MA: Addison-Wesley, 1995.

[Humphrey 00] Humphrey, W. *The Personal Software Process℠ (PSP℠)* (CMU/SEI-2000-TR-022, ADA387268). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00tr022.html>.

[Humphrey 02] Humphrey, Watts S. *Winning with Software: An Executive Strategy.* Reading, MA: Addison-Wesley, 2002.

[Janiszewski 01] Janiszewski, Steve & Myers, Chuck. "Making Haste Deliberately." *SEPG 2001: Focusing on the Delta* (CD-ROM). New Orleans, LA, March 12-15, 2001. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001.

[Jones 95a] Jones, Capers. *Patterns of Software Systems Failure and Success.* Boston, MA: International Thomson Computer Press, 1995.

[Jones 95b] Jones, Capers. *Backfiring: Converting Lines of Code to Function Points. IEEE Computer 28*, 11 (November 1995): 87-88.

[Jones 96] Jones, Capers. *Applied Software Measurement*. New York, NY: McGraw-Hill 1996.

[Jones 00] Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Reading, MA: Addison-Wesley, 2000.

[McAndrews 00] McAndrews, D. *The Team Software Process℠: An Overview and Preliminary Results of Using Disciplined Practices* (CMU/SEI-2000-TR-015, ADA387260). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00tr015.html>.

[Morgan 93] Morgan, Ben B., Jr.; Salas, Eduardo; & Glickman, Albert S. "An Analysis of Team Evolution and Maturation." *Journal of General Psychology 120*, 3: 277-291.

[Narayanan 02] Narayanan, Sridhar. "People – Process Synergy." *SEPG 2002* (CD-ROM). Phoenix, AZ, February 18-21, 2002. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

[Pracchia 03] Pracchia, Lisa & Hefley, Bill. "Accelerating SW-CMM Progress Using the TSP." *SEPG 2003: Assuring Stability in a Global Enterprise* (CD-ROM). Boston, MA, February 24-27, 2003. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.

[Riall 02] Riall, Cary & Pavlik, Rich. "Integrating PSP, TSP, and Six Sigma." *SEPG 2002* (CD-ROM). Phoenix, AZ, February 18-21, 2002. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

[Schwalb 03] Schwalb, Jeff & Hodgins, Brad. "Team Software Process for Maintenance Projects." *SEPG 2003: Assuring Stability in a Global Enterprise* (CD-ROM). Boston, MA, February 24-27, 2003. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.

[SEL 93] Condon, S.; Regardie, M.; Stark, M.; & Waligora, S. *Cost and Schedule Estimation Study Report* (Software Engineering Laboratory Series SEL-93-002). Greenbelt, MD: NASA Goddard Space Flight Center, 1993.

[Serrano 03] Serrano, Miguel A. & Montes de Oca, Carlos. "Using TSP in an Outsourcing Environment." *SEPG 2003: Assuring Stability in a Global Enterprise* (CD-ROM). Boston, MA, February 24-27, 2003. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.

[Sheshagiri 02] Sheshagiri, Girish. "It's Hard To Believe Unless You Do It." *SEPG 2002* (CD-ROM). Phoenix, AZ, February 18-21, 2002. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

[Webb 02] Webb, Dave. "Managing Risk with the Team Software Process." *SEPG 2002* (CD-ROM). Phoenix, AZ, February 18-21, 2002. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

Volume II
Software Process Subgroup
Task Force on Security across the Software Development Lifecycle
National Cyber Security Summit
March 2004

Edited by Samuel T. Redwine, Jr. and Noopur Davis