

Pursue Better Software, Not Absolution for Defective Products

Avoiding the “Four Deadly Sins of Software Development”
Is a Good Start

Should software producers be absolved from any problems caused by their defective products? Some software makers think so, and their allies in state and federal governments would like this absolution to be written into law.

The Uniform Computer Information Transactions Act (UCITA), which has already been passed in the Maryland and Virginia legislatures, says in part that vendors of software cannot be held liable for defects in the products they produce.

The Software Engineering Institute (SEI) strongly opposes UCITA, as do prestigious professional societies, including the Institute for Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM). (For more information about UCITA, see <http://www.badssoftware.com>.)

False Assumptions

We at the SEI believe that the rationale for granting special treatment to the software industry relies on four false assumptions:

1. High-quality software cannot be produced. This is demonstrably incorrect.
2. It costs too much to produce defect-free software. This is also demonstrably incorrect.
3. Customers do not want software that is defect free. This is the same argument that U.S. auto manufacturers used before Japanese cars began to take a big bite out of the U.S. market share—because people did value reliability when they had a choice. Why shouldn't the same be true for software? As the software market matures, will India, with its growing number of Capability Maturity Model® Level 5 companies, become the source of software that people will prefer to buy?
4. *Software is a vital industry that should be protected.* We agree that the software industry is vital. But it is not in the national interest to protect an industry that persists in using undisciplined and poor-quality practices, particularly when high-quality software can be produced at an acceptable cost (see assumptions 1 and 2).

Dispelling Myths About Software Quality

Two recent publications have sought to dispel the myth that high quality is impossible by presenting specific guidelines and techniques for enhancing software systems:

- “How to Eliminate the Ten Most Critical Internet Security Threats: The Experts’ Consensus,” published by the SANS Institute at <http://www.sans.org/topten.htm>. The list, compiled by representatives from industry, government, and academia, including the SEI’s CERT® Coordination Center (CERT/CC), can help administrators harden their systems against the handful of software vulnerabilities that account for the majority of successful attacks. The SANS Institute Web site also includes a step-by-step tutorial.
- “Software Defect Reduction Top 10 List,” by Barry Boehm and Victor R. Basili, published in the January 2001 issue of the IEEE journal *Computer*. Boehm and Basili provide an update of their 1987 article “Industrial Metrics Top 10 List” with techniques that can help developers reduce flaws in their code.

The Four Deadly Sins of Software Development

In this issue we present our own list, albeit one with a somewhat broader perspective. We hope the software community will consider, and build upon, the following Four Deadly Sins of Software Development.

Software Defect Reduction Top 10 List

By Barry Boehm and Victor R. Basili

from IEEE Computer, January 2001, p. 135-137.

1. Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.
2. Current software projects spend about 40 to 50 percent of their effort on avoidable rework.
3. About 80 percent of avoidable rework comes from 20 percent of the defects.
4. About 80 percent of the defects come from 20 percent of the modules, and about half the modules are defect free.
5. About 90 percent of the downtime comes from, at most, 10 percent of the defects.
6. Peer reviews catch 60 percent of the defects.
7. Perspective-based reviews catch 35 percent more defects than nondirected reviews.
8. Disciplined personal practices can reduce defect introduction rates by up to 75 percent.
9. All other things being equal, it costs 50 percent more per source instruction to develop high-dependability software products than to develop low-dependability software products. However, the investment is more than worth it if the project involves significant operations and maintenance costs.
10. About 40 to 50 percent of user programs contain nontrivial defects.

The First Deadly Sin: Sloth *Who Needs Discipline?*

A large number of defects are introduced into a typical software product from the detailed design phase through the unit test phase. Finding and fixing defects through testing, which is the usual method, can consume a great deal of time—often up to half the total development time. Furthermore, testing only uncovers some of the defects; 50% is generally considered to be a good number in the software industry. So even after organizations spend considerable resources in software testing, the product that comes out of testing is not defect free. To get defect-free products out of test, organizations must put essentially defect-free products into test.

Boehm and Basili point out that “disciplined personal practices can reduce defect introduction rates up to 75%.” By applying the principles of the Personal Software ProcessSM (PSPSM) and the Team Software ProcessSM (TSPSM), developed at the SEI, or the [Cleanroom software development process](#), originally developed by Harlan Mills, organizations can greatly reduce the number of defects present when the software reaches the testing phase, reduce the number of defects introduced into the software, and can achieve near-zero defects in delivered software.

The PSP method, for example, helps engineers improve by bringing discipline to the way they develop software. The TSP approach is designed to transition those disciplined methods into an organization, by helping its members form and manage high-performance teams. Together, PSP and TSP offer organizations the ability to produce very high quality software, to minimize schedule variance, and to improve cycle times.

In the PSP classes taught at the SEI and by SEI-licensed instructors, engineers learn sound design and verification practices, as well as structured and measured review procedures. SEI data shows that at the start of PSP training, students inject 10 defects in every 100 lines of code they write. Using PSP techniques, this number drops to about 0.7 defects per 100 lines of code. This is data from classroom projects, but real-world analysis shows similar results.

PSP- and TSP-trained engineers also catch defects earlier in the process, says the SEI’s Noopur Davis. “Because early defect removal requires much less effort than late defect removal, and because there are fewer defects to find and fix, the cycle time decreases and quality increases.” In other words, these practices can help engineers overcome another of Boehm and Basili’s findings: “Current software projects spend about 40 to 50% of their effort on avoidable rework.”

This approach differs significantly from the method for finding defects through testing, which generally involves a time-consuming process of looking at the symptoms of a problem—say, a system crash—and working backwards. Davis says, “You have to ask, ‘how could I have arrived

at this point in the program?’ and you have to look at all the paths you could have taken to get there. Very often, the symptom has very little to do with the cause of the problem.”

The Second Deadly Sin: Complacency

The World Will Cooperate with My Expectations

This sin can be the most lethal when developers are so complacent that they fail to defend against hostile input. Eighty-five percent of all security problems can be placed in the category of “failing to defend against hostile input,” says Shawn Hernan of the CERT/CC. What makes this sin especially egregious, he says, is that in many cases the defense would be easy to achieve: simply avoid software development mistakes that have already been identified and are well known, or make the necessary repairs, such as those recommended by the CERT/CC at its Web site (<http://www.cert.org>) and those listed on the SANS Institute’s top 10 list. “Most problems involve the same mistakes committed over and over,” Hernan says.

The training that programmers receive is partly to blame, Hernan says. When software developers first learn to program, they are told to focus on functionality and performance—for example, how to make a program do what it’s supposed to do faster. “But little attention is paid to making sure a program doesn’t do what it’s not supposed to do, when its assumptions are violated.”

Programmers also unwittingly open the door to hostile input when they try to extend the functionality of one program into a domain for which it was not intended. “You want to reuse everything that you can, but you have to be careful how you do it,” Hernan says. “You must understand the assumptions of the developer and packager before you reuse something in a system that has security requirements.”

One example of failing to defend against hostile input is the CGI program “phf,” a directory service that was once included with popular server software. The program provided a Web-based form that was designed to allow users to get legitimate information from a directory, such as telephone numbers (“ph” stands for “phone” and “f” stands for “form”). The problem was that the easy-to-use form would construct a UNIX command line and execute it exactly as if it were typed into an actual command line—in other words, it gave users of the Web form command-line access to the server. Instead of simply typing in a name on the form to get a telephone number, an intruder could plug in, for example, “\$ph smith; cat /etc/passwd” and gain access to the password file on the server. “The problem comes in using a system command over the Web. The system command is way too powerful for this purpose,” Hernan says.

The implications of allowing hostile input into a computer system are myriad and extreme, and can include a compromise of an entire system. In the case of phf, the CERT/CC received hundreds of reports of hostile users exploiting the security flaw to modify Web sites and compromise computers.

The CERT/CC advises writers of software code to include checks of all input for size and type to make sure it is appropriate. This prevents the use of special characters, which was one of the problems with phf. The CERT/CC also advises code writers to limit inputs to the allocated space, which helps prevent buffer overflows that could occur when an intruder attempts to store more data in a buffer than it can handle, resulting in the loss of service on a computer or system. The SEI technical report [rlogin\(1\): The Untold Story](#), provides an analysis of software defects that could result in buffer overflows from one well-known case, and some general guidelines and mitigation strategies for preventing similar defects.

Hernan points out that developers should be especially cautious when dealing with Web-based programs, which should be considered “privileged” because they can grant extraordinary access to users. “Anything that operates over the network is a privileged program. So, anything that operates over the Web is a privileged program. If you extend a privilege to people everywhere, then you give them the ability to run anything on your system.”

The Third Deadly Sin: Meagerness *Who Needs an Architecture?*

“An explicit software architecture is one of the most important software engineering artifacts to create, analyze, and maintain,” says the SEI’s Mark Klein. Yet, developers often leave the architecture invisible, or “it is covered and permuted by the sands of change.”

Even when the principles of an intended architecture are laid out in advance, it is difficult to remain faithful to that architecture as design and implementation proceeds. This is particularly true in cases where the intended architecture is not completely specified, documented, or disseminated to all of the project members. In the SEI’s experience, well specified, documented, disseminated, and controlled architectures are the exception. This problem is exacerbated during maintenance and evolutionary development, as the developed product begins to drift from the architecture that was initially intended.

When the architecture of a software system is invisible, it means no one has intellectual control of the system, the design of the system is very difficult to communicate to others, and the behavior of the system is hard to predict. As a result, changes are difficult and costly, and often result in unforeseen side effects.

Suppose that a developer inserts a new middleware layer of software so that distribution and porting of the system become much easier. Without an explicit architecture to refer to, that developer might fail to realize that this change negatively affects the performance of the timing-critical aspects of the system.

An invisible software architecture also represents a missed opportunity. The ability to identify the architecture of an existing system that has successfully met its quality goals fosters reuse of the architecture in systems with similar goals. Indeed, architectural reuse is the cornerstone practice of software product line development.

Developers must build systems with explicit architectures in mind, and must remain faithful to those intended architectures. At the very least, significant changes to a software system must be explicitly recorded. By using techniques such as the Architecture Tradeoff Analysis MethodSM (ATAMSM) developers can analyze the tradeoffs inherent in architectural design, so that they can understand how various quality attributes—such as modifiability and reliability—interact. They can then weigh alternatives in light of system requirements.

For systems that are already built, all is not lost. Software architectures can be reconstructed through the acquisition of architectural constraints and patterns that capture the fundamental elements of the architecture. Commercial and research tools are available that provide the basic mechanisms for extracting and fusing multiple views of a software program or system. (The SEI has collected several of these tools and techniques into the “Dali” workbench.) Using such tools can allow for the reconstruction of a system’s architecture even when there is a complete lack of pre-existing architectural information, as might be the case when the system being analyzed is particularly old, and thus there are no longer any designers or developers who can relate architectural information.

The Fourth Deadly Sin: Ignorance

What I Don't Know Doesn't Matter

Here, designers develop software systems without showing proper regard for the intended purpose of software components. This problem appears more frequently today as designers increasingly assemble systems by integrating commercial off-the-shelf (COTS) components. To do this properly, designers must fully understand the components they are using. The final system will often fail to meet expectations if designers do not thoroughly investigate the components they select and the tradeoff decisions that must be made and quantified in terms of cost and risk.

The SEI is developing methods that designers can use to make objective evaluations of components and then competently use those components when assembling systems. The goal is to provide owners and users with the system they think they are getting.

“A designer’s perceived reality of a product is built up over time and could come from any number of sources.” says the SEI’s Scott Hissam. “These sources might include product literature, articles in trade magazines, or hearsay.” All of these sources suggest to designers what the product can do, such as providing distributed transaction services, load balancing, fault tolerance, or seamless integration of X to Y.

When system designers' perceptions of a component don't match the reality of the product, those designers can expect to run into problems when they try to integrate the component into a system. For example, the product could turn out to have unanticipated security vulnerabilities. Or a designer could find that two components that purport to have the capability to seamlessly integrate do not do so. This might result in a great deal of additional work to make the components function together, or it could mean selecting other components for the overall system, or the overall system, when built, may not meet performance, security, or dependability requirements.

Eliminating Mismatches

To gain the insight needed to make better decisions about component selection, designers must work through their expectations iteratively and learn precisely what the product does, what it requires, and how it behaves. Designers should ask questions such as: "What does the product need from me?" "What does the product need from other products in my environment?" "How does the product behave when I try to use it?" "What resources does it use?" Some intangible aspects of a product might not appear on the license agreement or in the product specification, such as the threading model, latencies, and security (or lack thereof). Such information can often only be gained through actual interaction with the product.

Designers often have "a slightly skewed vision of what the specifications are for products because there is such a huge implied environment that people never document," says Bill Fithen of the SEI's Survivable Systems initiative. "They just assume that, for example, 'UNIX' is a satisfactory definition of the environment in which the program is going to run. Perhaps 90% of the errors that we see fall into that category: the designers did not consider what the environment really looked like. They considered some aspect of it but not all of it."

A pervasive example of undocumented assumptions can be found in the TCP/IP protocols—the suite of communications protocols used to connect hosts on the Internet. TCP/IP was developed by the U.S. Department of Defense to connect a number different networks designed by different vendors into a "network of networks." One of the assumptions made by the originators of the protocol was that parties on both ends of the communication actually wanted to communicate. It never occurred to them that on one end of the communication the goal could be, for example, to gather sensitive data from the system on the other end.

Conclusion

We believe that the practice of software engineering is sufficiently mature to enable the routine production of near-zero-defect software. Our Web site (www.sei.cmu.edu) provides data and case studies to prove this (see, for example,

<http://www.sei.cmu.edu/tsp/results.html>). As consumers of commercial software, we should demand and expect the highest possible quality.

As the U.S. Department of Defense comes increasingly to rely on the commercial software industry to provide components in its systems, it is important to our national defense that software developers be held to standards of quality and accountability that are in effect in other industries.

Ironically, it is in the U.S. software industry's best interests to produce better software, as the SEI's Watts Humphrey has written in a recent column in *news@sei interactive* (http://interactive.sei.cmu.edu/news@sei/columns/watts_new/2001/1q01/watts-1q01.htm). Improved software quality "would mean increased opportunities for computing systems and increased demand for the suppliers' products," Humphrey says.

What's more, continuing to ignore quality—and even to seek legal protection for defective software through such mechanisms as UCITA—will ultimately invite competition from other nations, whose software makers will learn to make superior products and could eventually replace the United States as the industry leader.

We welcome your feedback on this article and invite you, as members of the software-development community, to help us identify more sins of software-development. We plan to update and add to this list at least annually and perhaps more often, depending on your response. Please send your comments and suggestions to interactive@sei.cmu.edu and join us in the quest for the highest possible software quality.

Copyright © 2001 Carnegie Mellon University

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

® CMM, Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, and CERT Coordination Center are registered in the U.S. Patent Trademark Office.

SM ATAM; Architecture Tradeoff Analysis Method; CMMI; CMM Integration; CURE; IDEAL; Interim Profile; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Personal Software Process; PSP; SCAMPI; SCAMPI Lead Assessor; SCE; Team Software Process; and TSP are service marks of Carnegie Mellon University.

TM Simplex is a trademark of Carnegie Mellon University.