The Quest for Perfect Software

Testing and analyzing in the attempt to reduce software bugs.

Written by James Yoder

Dr. Deborah Hwang
CS 390
October 22, 2008
Perfect software is the dream of many programmers. Software tools and programming languages are getting better and making it easier to write complicated programs. Unfortunately, humans are fallible creatures. Whether it is critical software for the military, a commercial operating system sold throughout the world, or a freeware computer application, programmers make mistakes in their code. Unlike mistakes in other career fields however, these bugs often go undetected until they cause thousands of dollars in damage and require even more time and money to fix when they could have been detected and fixed early in development. Although they cannot solve communication organization issues, new tools and techniques such as automatic unit testing, static code analysis, and contract-based testing are helping to improve the quality of code and detect bugs and security holes early in development.

As software grows larger and more complex, the industry is being buried in a deluge of bugs. Recent software faults have caused Toyota Priuses to stall at highway speeds, California’s 911 system to crash, airport radars to malfunction, and warning sirens at a New York nuclear plant to fail. [8] Software conflicts caused electronic voting machines to drop votes from 11 counties in Ohio, resulting in lawsuits from the state. [6] The average computer program is no better. According to Capers Jones of Software Productivity Research, there are on average about five faults per feature, and only 85% of these are fixed before the product is shipped. [1] Arden Bement, director of the National Institute of Standards and Technology (NIST) explained the breadth of the problem: “The impact of software errors is enormous because virtually every business in the United States now depends on software for the development, production, distribution, and after-sales support of products and services.” [7] Security holes are another problem with software. Although they are invisible to the average user, malicious people can use them to subvert the security of the computer. Security holes are harder to detect, and often go ignored until after they are exploited by a virus or computer criminal. Buggy software is prevalent and, sadly, is often considered to be normal by many people. But software faults are not just an inconvenience; they also cost the public dearly in time and money.
Faults themselves have a high cost to both customers and the software makers themselves. According to a NIST study, software errors cost the U.S. economy around $59.5 billion dollars annually. [7] And the economic costs keep increasing. As programs get more complex and become more necessary in daily life, software errors have more of an impact and can cause more problems. The study also found that more than a third of the costs (around $22.2 billion) could have been prevented simply by using better testing techniques and discovering software faults earlier in the process. Discovering faults before shipping saves an enormous amount of time, money, and the fixes have fewer regression errors than attempting to deploy patches after the program has been shipped. Although this is taught in software engineering classes, some companies have to learn about the increasing cost of fixing bugs the hard way.

Microsoft had to learn about the importance of software testing the hard way. After releasing Windows XP, they began to work on Windows Vista like they had always done; each programmer started working on their own section of code and wove it together every so often, which often took a week. [3] Unfortunately, they had focused on features and ignored existing and newly added bugs, leaving those to patch later. [3] Bugs were difficult to fix and were largely ignored until later because the programmers would rather write new features than fix elusive errors. After bringing in two brilliant consultants who were experienced in reviving stalling projects, they set up a new system. They set up automated testing to analyze the code before it was integrated into the code base. “If a feature had too many bugs, software "gates" rejected it from being used in Longhorn. If engineers had too many outstanding bugs they were tossed in "bug jail" and banned from writing new code.” [3] The new system prevented very buggy code from making its way into the project and penalized programmers who spent too much time writing new code and not enough time fixing the bugs that they caused. In the end, the time and effort spent paid off. The time to rebuild the operating system from the code was significantly shorted. The bug reports that should have numbered in the tens of thousands only numbered in the thousands. [3] Although they still have further to go, Microsoft is on their way to making a more quality operating system and provide another example of the need for automatic testing.
Automatic unit testing is changing the way testing is done. Although software writers have written programs that use the computer’s power to make things simpler in other fields of work, only recently have they begun to use the computer to make testing easier on themselves. Programmers design test cases, often before they even begin to code the module, and have the computer run the tests and compare the results. These can be performed nightly to ensure that the code always works and that problems are fixed immediately instead of letting them slip into the finished product. Designing the tests before the module is built also helps prevent the module from drifting away from the specifications, the same problem that ultimately destroyed the NASA Mars Climate Orbiter. [4] In addition, automatic tests can perform regression testing, tests that make sure that fixing a fault does not break another section of code. When a problem is fixed, a test case can be made that fails if the problem is reintroduced. Automatic unit testing is becoming a key part of bug detection and is being incorporated into many programmers toolsets, but new tools are being developed that can detect buggy code before it is even run.

Static analysis tools can analyze code and detect bugs without having to run any test cases. Although no tool can reliably indicate the absence of bugs, static code analyzers can indicate the quality of the code, possible coding errors, and assertions that can fail. [1] Some analyzers just check the style of the code to ensure that it follows the company’s style guidelines and coding rules. These checkers help keep the style of source code submitted uniform and easier to modify, but do not help as much with detecting errors unless the error itself contradicts a coding rule. Others provide coding metrics and highlight areas of code that may need reviewing, refactoring, or more unit tests. Coding metric analyzers are helpful for finding problem areas in large projects, but like style checkers, they cannot find specific errors. Unlike style and metric analyzers, deep code checkers can scan source code and ferret out specific bugs, sometimes even automatically fixing them. [1] When combined with the other analyzers, these can be extremely helpful in proactively finding and fixing actual faults in the code. Static code analysis is a powerful way of detecting faults, but the extent of testing it can do is limited. Dynamic run-time testing is still required to ensure that
the code works in use. But dynamic testing is getting easier thanks to a new development, contract-based testing.

Contract-based testing is an experimental way of providing both automatic testing and verifying the correctness of code. Each module defines a contract as a part of the code which describes necessary preconditions, post-conditions, exceptions, etc. Contracts enable an automatic testing program to create test values and the desired oracles and check for unfulfilled preconditions. With contract-based testing, making test cases is an automatic part of testing, easing the difficulty of creating test cases and saving time. Testing involves just running the testing program which reports any faults that it found. Unfortunately, contract-based testing is not the panacea that it appears to be. The contracts have to be both complete and correct. Complete in that they specify all needed preconditions and can describe all possible changes that it can do. In an open environment, this is often very difficult to do. In addition, the contract has to correctly represent the specifications. Not only that, but the specifications and requirements that the contracts are based on must be correct as well. Contract-based testers, such as AutoTest [5], are going to make testing much simpler than it is today, but it cannot overcome faults in communication and organization that often overshadow the actual faults in the code.

Communication with other programmers is a key ingredient of software engineering that often goes ignored. Jonathan Pincus, an expert on software reliability stated that “the key issues [in programming] relate to people and the way they communicate and organize themselves.” [1] Project management plays a big role in software reliability. Programmers must be able communicate well with their superiors, fellow programmers, and project leaders to be productive. According to an ACM article, programmers who are outgoing and extroverted perform better than programmers who are introverted. [2] The article speculated that it was because extroverted people communicate better with their team members and were able to make group decisions better than introverts. IBM’s Grady Booch “once tracked 50 developers for 24 hours, and found that only 30% of their time was spent coding—the rest was spent talking to other
members of their team.” [1] Testing helps improve the quality of software, but it fails miserably in ensuring that programs are designed well and that programmers communicate efficiently.

Programming is a complicated art. Unlike many professions, problems are numerous and can go undetected for years. Proper testing with both static code checking and dynamic run-time testing is vital in detecting and fixing software faults. Studies show that problems can be greatly reduced and nearly bug-free code can be written if programmers put aside their scorn for testing and work hard to ferret out the bugs that are in every piece of code that they write. Contract-based testing is proving to greatly ease the burden of testing and produce more thorough test cases, but even it has limitations that prevent it becoming a replacement for manual testing. In addition, inefficient programmer communication and bad program designs can subvert even the most diligent testing and make an inferior product. Testing is not a silver bullet, but when used in context with good software engineering, testing can aid us on our quest for perfect software.
Bibliography


