The Necessity For -- And the Evolution Of -- Software Testing

By Michael Zlatkovsky

CS 390: Software Engineering

October 29, 2008
In almost any domain, complexity grows exponentially with the size of the input. Consider, for example, the classic handshake problem: "How many handshakes are possible in a room with n people". With only two people in the room, the number of handshakes is 1; with five people, the number of handshakes is 10; with 20 people, the number of handshakes is 190. As the size of the input increases, the complexity increases at a much more rapid rate.

So it is with software. As the article "Software Testing 3.0" by LogiGear [1] explains, when computers were still in their early stages and most code was short and specialized for a given task, the potential for undetected faults was fairly low. Once written, the program either predictably worked or did not work at all: thus, just a few runs of the program could suffice for "debugging" the software. But as computers have gotten more storage capacity, as software has become more inter-connected (and reliant on other software/frameworks), and as programs have become more generalized, keeping track of all of the potential sources of error has become an increasingly complex task. Testing -- and, specifically, automated testing tools -- therefore has become an increasingly important task in the software development lifecycle, and only now is rising to the prominence and sophistication that it deserves.

In the early days of programming, testing was misunderstood. Software was written either completely without tests or with very minimal testing, and it resembled more of an afterthought than a true stage in software development. Testing did not have its own budget, received almost no attention from upper management, and lacked the appropriate testing and debugging tools. When testing did occur, often it was done by lesser-skilled programmers, who were deemed to be too inexperienced to write good code themselves, but who could be recruited for the seemingly dull and mundane testing phase. In such cases, testing was often put off to the last minute, done hurriedly, and generally was considered to be a waste of time.
The article describes this undeveloped early stage of testing as "Software Testing 1.0". This stage was marked by a general lack of testing, complete with a lack of tools and a lack of managerial supervision whenever tests did occur. Fortunately, the effects of such lax testing methodology were not too devastating for early software development, since the code itself was often simple enough and self-sufficient enough that a last-minute test could easily catch the majority of the bugs. As computers gained more power and code started to get more generalized and more re-used, however, the undetected cracks in the code became all too evident. It was at this stage that software testing transitioned from 1.0 to 2.0.

Whereas actual testing was a rare and unorthodox occurrence in Software Testing 1.0, Software Testing 2.0 saw an acceptance in the notion that testing MUST be an integral part of software development. Unfortunately, the mere fact that testing became "integral" did not easily specify where software testing fit in an organization, and how it was to be performed. For example, some organizations continued to assign testing to the lesser-skilled programmers, while others made it a mandatory part of any developer's schedule. Likewise, some organizations required daily or weekly testing, whereas other organizations viewed testing as a last-minute activity. Meanwhile, an explosion of testing tools became available to the programmers, though without a clear understanding or a direction for testing, tool selection became a time-consuming and ultimately useless endeavor. Still, programmers had at least come to embrace the concept of testing, if not any successful implementation thereof. Upper management, however, remained detached from the testing process, curtailing the potential benefits that would come from earnest software testing.

The net result of Software Testing 1.0 and 2.0 was a large increase in software testing but only some increase in software quality. To some organization managers, this was a very
troubling fact; after all, programmers who could otherwise have been working on other projects had instead directed their efforts to software testing, yet there were hardly significant gains in the software quality or in the programmers' productivity. Some saw this unreturned investment on testing as a reason NOT to test: a testament to the futility of the whole software testing endeavor. What such managers did not realize is that "without a clear understanding of the importance of testing, its place within the development process and its valuable contributions to the organization, testing hopes can be dashed upon the rocky shores of misguided direction, funding, and expectations" [1]. Improperly applied testing could indeed be a waste of an organization's time and money; but such improperly applied testing spoke badly not of Testing as a whole, but rather of the inappropriate application thereof.

The current state of testing is somewhere between 2.0 and something slightly higher. In response to this, LogiGear suggests a Software Testing 3.0 state -- a state of testing that is within reach of many organizations, and that, if properly applied, will finally provide those organizations with the elusive benefits that were missing from Software Testing 1.0 or 2.0. The Software Testing 3.0 state views software testing as a strategic initiative to which organizations must truly devote their full energy. No longer can programmers be left to decide for themselves how to carry out tests, when to carry out tests, and what projects can or cannot be postponed in lieu of testing. Rather, management must actively be involved in finding the appropriate testing methodology and enforcing it throughout the organization. The organization must also realize that software testing is its own separate component of a project development lifecycle, and cannot be pushed around or lumped (and forgotten) with other stages.

Haphazard testing, while certainly better than no testing at all, leads to a lot of overlap of some test cases, and little to no coverage of other parts. So long as programmers test code at
their own leisure, it is impossible to centralize the software testing, nor is it possible to ensure adequate coverage; at the same time, a lot of valuable coding time gets wasted on redundant tests. Also, while a good programmer and a good tester must share some skills in common, it is unrealistic to assume that a programmer can be as good at testing code as someone who is specifically trained in the testing code, and vice-versa. After all, "Development is a creative process of providing new functionality and features. Testing is a destructive process where testers actually try to 'break' the software so that customers do not" [1].

Therefore, an organization that is ready to embrace Software Testing 3.0 must dedicate a separate team to handle software testing (though certainly not just recruit sub-quality programmers to do the testing, as in Software Testing 1.0). To be sure, developers must still perform preliminary tests on their own code, but they should not be held responsible to catch all of the bugs or to ensure 100% integration with others' code. Dedicating a separate testing team allows the organization to centralize its tests, train the selected individuals in the art of software testing, and also not succumb to a common code-attachment ego-problem that invariably arises when programmers must find faults with their own precious code. A separate testing team can also ensure that the development team will be less likely to "ram sub-standard code ... to meet delivery promises without a red flag being waved to senior management" [1]. That is, separating development and testing not only creates a specialization of labor (and hence higher productivity in each group), but also enforces a certain checks-and-balances system so that developers can be more accountable for the quality of their code.

Another advantage to centralized testing is that testing can then be made more visible throughout an organization. Testing provides valuable data about an organization's performance, such as the number of bugs caught (and uncaught) and the testing breadth (the number -- and
coverage -- of the performed tests). Displaying this information throughout the organization can help management focus on the areas that need improvement (such as better staffing or more training in certain areas of the organization), as well as provide some feedback and assurance about the quality of the released software.

With earnest and rigorous software testing comes knowledge and power: the knowledge about the current state of the software (such as how close it is to being done, how well it handles unusual cases, etc), and the power to change current code or to integrate formerly-written code and be sure that appropriate testing procedure are in place to catch any bugs and incompatibilities. Though learning the appropriate testing techniques, re-structuring an organization, and getting used to the culture change can all be cumbersome processes that initially might even detract from an organization's productivity, a software company's long-term success depends on adequate testing procedures. Not adequately testing code can lead to long-term customer dissatisfaction, inability to reuse older code (for fear of incompatibilities), and wasted programmer's time (who might do overlapping tests, or who cannot be sure that a piece of code that had been handed down to them can be confidently used).

Instituting proper testing techniques is not cheap: Software Testing 3.0 requires a separate testing group, the appropriate software tools, and a fair amount of training. But while an organization might have troubles affording the cost of this change, how could it afford not to implement the necessary changes? The only way that a software company can succeed in the long run is to write good software; yet, no matter how good the programmers are, good software must be tested. Therefore, if software testing is essential for good software, and if better testing techniques leads to better (or more efficiently tested) software, optimizing software testing is the key to an organization's long-term success.
References: