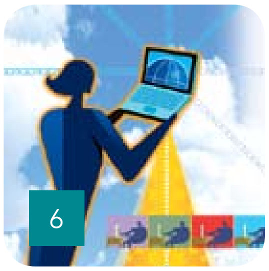
Guide to Developing Better Software

coverity®

an internet.com® Developer eBook

Contents...

Guide to Developing Better Software



This content was adapted from Internet.com's CIO Update, DevX and Developer.com websites. Contributors: Herman Mehling, Ananda Rao Ladi, Clinton Sprauve and Mike Rozlog.

Most Software Bugs Due to Poor Testing Procedures

By Herman Mehling

How painful is it to use your company's software testing system? If your answer is more painful than dealing with a fender-bender, or coping with an equivalent annoyance, you are not alone.

In fact, you are in the majority, according to a survey on software testing done by Osterman Research, and Electric Cloud, a provider of software production management solutions. Fifty-eight percent of developers who responded to the survey said problems in the testing process or infrastructure — not design defects — were the cause of the last major bug found in delivered or deployed software.

When a bug is found in released software, the bottom-line impact on an organization is significant, says Michael Osterman, CEO, Osterman Research. Fifty-six percent of respondents estimated that their last major software bug resulted in an average of $250,000 in lost revenue and 20 developer-hours to correct, the survey found.

Fifty-six percent also reported that bugs discovered late in development almost always affect release dates.

The survey questioned 144 software development professionals from organizations with at least 1,000 employees and 50 developers across a variety of industries. Most organizations surveyed were located in North America.

"As the software we rely on each day continues to grow in complexity, it becomes more essential that bugs are caught and repaired quickly," says Osterman.

But most companies don't do a great job of catching bugs, he noted.

Eighty-eight percent of the surveyed companies still use manual testing to some extent, and rated dealing with their test systems as more painful than dealing with a fender bender.

"Software developers have a long way to go toward full automation and effective test systems," Osterman adds.

Developers who felt their companies give enough time to pre-release testing are less impacted by bugs, and spend less than half as much time resolving bugs compared to other developers — a median of 12 developer-hours compared to 25 developer-hours.

Companies' continued reliance on slow, resource-intensive manual processes prevents them from being as thorough as necessary in their testing, says Electric Cloud CEO Mike Maciag. Fully automated test systems save time and use physical, virtual or cloud resources, while greatly reducing the risk of human error, says Maciag.

However, the survey shows that completely automated software testing environments are still rare, with just 12 percent of software development organizations using fully automated test systems. Almost 10 percent reported that all testing was done manually.

Other key findings of the survey: 46 percent of developers said they don't have as much time to test as they should; 36 percent said their companies don't perform enough pre-release testing; and 53 percent said testing is limited by compute resources. ■

Effective Software Testing: An Approach To Better Products

By Ananda Rao Ladi

The cost of fixing a software defect is cheaper if the defect is detected in the phase in which it was introduced. Yet in many projects of strategic importance, this testing is only an afterthought. Traditionally, testing is associated only with the end product. And, in many IT organizations, the developers double up as testers meaning the complexity and economics of testing are not understood adequately.

All these factors result in spending unreasonable amounts of time and money on application maintenance and rework, which leads to unhappy customers. Taking a tactical or a one-dimensional approach to these challenges seldom succeeds. Therefore, what follows are some of the industry's leading strategies to ensure that testing leads to better products and happy customers:.

Establishing an Independent Testing Practice

Research has shown that testers have a different mindset than developers. The methodology, tools and techniques for testing are quite different from development. It also takes two-to-three years for a tester to be effective.

Testers, by the nature of the job, know more about the products than anyone else in the organization. Learning about the competitive landscape is an absolute must to test a product completely. This business and product knowledge could lead to very valuable inputs for the product management team. The testing practice should only be focused on the quality of the product and not be affected by the time-to-market constraints.

Adoption of Test Automation

Exhaustive testing is almost impossible to perform. At the same time, testing coverage should be adequate to ensure that the product has the required quality. Additionally, 60 to 70 percent of most testing doesn't need manual intervention. Automating this part of testing will result in better coverage and will consume less time.

Ally with an Offshore Service Provider

Testing, especially during the execution, will have a spiky need for manpower. Many IT organizations may not even have an independent testing practice. It is highly domain-knowledge intensive and the knowledge of different tools is absolutely necessary.

Many offshore IT services organizations have, over the years, built very focused testing groups. By giving attention to the special needs of the testers and to the need for the business domain knowledge, these groups are able to provide much better end-to-end testing services. They are able to provide testers with different domain knowledge, having very strong testing background. The key is to select an offshore IT service provider, which has an Independent Testing Practice and is investing continuously on building capability.

Good Process

Since exhaustive testing is almost impossible to achieve, a good process foundation is critical. Processes to determine whether a product is ready for release use either the testing maturity model (TMM) or the test process improvement model (TPI). They are two of the best models for defining the test processes.

Open Source Tools

Effective use of testing tools will result in better predictability, consistency and reduced time. The strategy should cover all the aspects of testing including test management, defect tracking, test execution and reporting. The open source community has developed some of the best tools in this space. They should be given a consideration, in addition to the commercial tools available.

Think of Testing Before Development

The traditional association of testing only with the end product is leading to more defects being detected at a later stage. This leads to additional money and time being spent on fixing problems. The best strategy is to involve senior testers right from the beginning of the product development lifecycle; at the product planning stage. Every milestone right from the beginning must be tested. This includes the product requirements specification, functionality specification and the design specification. ■

Futility-Based Testing: Bad Practices That Destroy Test Automation Efforts

By Clinton Sprauve

Think about it: mainstream, commercial test automation tools have been around since the late 1980s. Open source test automation frameworks and tools like FIT, FitNesse, Selenium, Sahi and Cucumber are commonplace. Every up-and-coming development methodology puts a strong emphasis on testing (i.e., Feature-Driven Development, Test-Driven Development, Behavior Driven Development, INSERT YOUR METHODOLOGY HERE-Driven Development, etc.). Although test automation is no longer considered a nice-to-have, why is it that most development organizations still can't get it right?

The answer is bad practices across the spectrum of development and testing methodologies. Whether it's traditional waterfall or Agile, the community-at-large still doesn't get it. What are we doing wrong?

This article will explore a few of the bad practices and futile habits that prevent many organizations from reaping the true benefits of software test automation.

Assuming Test Automation is So Simple Anyone Can Do It

I mentioned earlier that the current mainstream testing tools have been around since the late '80s. Because of the record-and-playback nature of the tools, it is assumed that anyone that can use a mouse can do test automation. Part of this assumption can be blamed the industry's heavy reliance on subjective criteria (i.e., ease of use or user interface) more than objective criteria of test automation software (i.e., built-in support for the technologies of the application under test, ability to re-factor test code, etc.). In order to sell test automation to the masses, it has to appear so easy that all you have to do is point-and-click your way into ROI heaven. Not true.

Test automation IS software development —period. Assuming that you can send the receptionist to test automation training and build efficiencies into your software development process is ridiculous.

Solution: take the time to build a test automation team of people who understand software development and programming concepts. The testers don't need to be your uber-developers, but they must have the wherewithal to understand, develop and maintain test automation code.

Record and Playback

This relates heavily to the last assumption that anyone can build test automation. Why is record and playback so popular? Because it makes your team members feel productive, even though it's really an exercise in futility. Let me explain.

Let's say you had the mythical ability to record your morning drive to work and replay it. What happens when there is a wreck, or traffic or a four-way stop (you get the idea).

Record and playback lets your team members create throwaway test scripts. The script doesn't work? Don't worry, launch the recorder and start over. Item in the list moved from No. 5 to No. 29? No worries; launch the recorder and start over. And so on and so on...

Again, take the time to build a test automation team of people who understand software development concepts. If automation were that easy, then there wouldn't be a need for test automation specialist.

'Job Security' Test Frameworks

Sometimes there is a downside to having an uber test automation specialist, or having just one person responsible for building and maintaining a test framework for an army of testers. I'm sure many of us are familiar with the term "spaghetti code." What happens when an uber-tester gets hit by a bus or leaves the company? We'll miss him, but who is going to maintain the framework? Many times the test automation is scrapped upon a new release of the application and the test automation efforts starts all over again.

Never rely on a single engineer to develop and maintain anything that another technical team member or new hire can't quickly understand or easily pick up. It is imperative that your framework is simple enough to maintain and well documented so that the company and team doesn't lose the efficiencies created by the framework in the first place.

Kumbya (coom baa ya) Keyword-Driven Testing

OK, some of us "get it." Test automation isn't easy. Here's an idea: let's build a code library of table-based functions/action words so that Ronny, Bobby, Ricky and Mike from accounting can help us automate the testing of our application. Now the entire QA team can help us automate! Really...
Let's examine the pros, cons and assumptions of KDT:

Pros

- Less technical expertise needed to create test automation
- Attempts to involve business analysts (BAs) and subject matter experts (SMEs)in the test automation process
- Automation Engineers do the "heavy lifting"
- Simplifies the link between testing and requirements specification

Cons

- Can actually increase the amount of maintenance for test automation efforts rather than reduce it.

For instance, Suzy from payroll is brought in to test the new accounting app. She's been trained on the framework and how to build tests. She's ready to go. While testing the app she gets an error "object xyz failed to initialize. Shutting down." She has to ping the test automation guru to understand the error. Is this an application problem, or a keyword framework problem? Who has to fix it? When will it be fixed? How many test cases does this affect? Then the crazy cycle starts over again.

- Difficult to manage changes in AUT as well as maintain automation expertise (i.e., who manages the framework when the creator leaves? Remember the" job security" framework?)

- Involves SMEs and BAs and testers in the "wrong way" — the intentions are good, but it's setting a trap for failure.

Let me clarify a key point: keyword-driven testing is NOT a bad thing. The problem is not necessarily how it is implemented but for whom it is implemented. Again, it goes back to my previous assertion: most people assume that test automation is so simple anyone can do it. The entire team does not need to be involved in the test automation process. Keep it simple. Utilize those on the team that have the technical expertise to develop, maintain and execute a keyword-driven framework. Remember, test automation is software development.

If you are going to implement keyword-driven testing, take inventory of the skill set of your team. KDT works best when the entire team has the technical expertise to maintain, execute and enhance the framework.

Take advantage of your nontechnical team members by utilizing their skills as subject matter experts. Allow them to do exploratory testing. Allow them to validate the business rules of the application. Don't set them up for failure by forcing them to be what they are not.

Software test automation is not easy. Building efficiencies into the development process is a difficult undertaking in itself. However, repeating the same mistakes over and over again will keep test automation on the "crazy cycle" of software development. Don't look for the ultimate panacea for test automation, but rather a practical, realistic approach to building a robust and reusable automation library that will provide true ROI. ■

Why Code Coverage Tools Should Be in Every Developer's Toolbox

By Mike Rozlog

Today's developers are under more and more stress to deliver more code in less time. For most developers, they use tools to help produce reusable code. This is a good thing, but with all the code being generated today, how much of the code is instrumented for testing? Forget testing, how well do you know your code?

Most experts state that the test-coverage is around 25 percent across the board and this is after the adoption of great frameworks based on the xUnit approach. But again, the bigger question is how well do you, or how well does your team, know the code? How do you learn about code? I believe one of the great underutilized tools in the developer's arsenal today is code coverage.

What is code coverage? It is a metric that shows how much of the code has been exercised during the execution of the application. This type of tool has been around a long time and in various formats, but a good code coverage tool is essential in writing and understanding great code.

Today's code coverage tools use various approaches to give better understanding of the code being monitored. These include statement coverage; simply put, this is if a statement is executed it reports it as such. Meaning if I have a program that is 100 lines long and 30 of them are executed during the monitoring, it would give me 30 percent coverage. The second type is condition coverage; what part of a conditional statement gets executed. Then there is the 30,000 foot level or high-level blanket coverage; it gives a total top percentage of code executed against the total lines of known code.

This information can be very enlightening. Think about it... you exercise your application with code coverage turned on and the results come back as 35 percent coverage. What's the deal? Did you just run your application with the "happy path" only or did you truly exercise the application? If you did exercise the application, then it appears a great deal of code may not be needed. However, this assumption may be jumping the gun and a little further analysis will be required.

Before we get into some of the things that code coverage can point out, we should take a step back and look at where code coverage can be used. The great news is that code coverage can be used all over the place — from full Q/A testing and full unit testing, to simple developer testing. Each of these areas can benefit from the use of a code coverage tool.

One test that should be run with code coverage is in reference to Q/A. Almost every Q/A department I've ever been exposed to has a Q/A script for the application getting tested. This script may be automated or run by hand, but all Q/A departments have a repeatable script. It is always interesting running code coverage with that Q/A script. Just like the above example of only 35 percent returned, it is always interesting to see the percentage returned after running a Q/A script. It is usually very low, under 60 percent. What does this mean? First it means 40 percent of the application is not getting tested. Remember, usually these scripts are created over time and are used to judge the quality of a product. Normal IT shops don't ship, or release a product until it passes the Q/A script. So getting information back that less than 60 percent of the code is being exercised can be very eye-opening.

These numbers are in line with a study (Wiegers 2002) that showed only about 50 to 60 percent of the code was actually touched in a testing script. Think about what this could mean from a developer's standpoint. What if the company could remove 20 percent of the code that is not used? What would be the cost savings or reduction in overhead if that was possible?

Usually, once this information is known, the Q/A department takes a step back and reviews the areas or functions of the application getting exercised because they are always worried that something will slip through the cracks. Before code coverage, it was always hard to ascertain the percentage of the application getting covered. Using code coverage the task becomes much easier.

Think about how you can focus around the developer and how code coverage can be used with unit testing. If you ask most developers if they do a good job testing their code, they would tell you yes. However in another study (Boris Beizer in Johnson 1994), the average programmer thought they tested around 95 percent of the code they wrote, but on further inspection the amount of actual code was more like 30 percent. There is a large difference between those two numbers. Remember how the "happy path" may be the focus, in most developers' minds, if a procedure is supposed to do a summation of two numbers and the program does that, they are done. But what happens if a letter or character is exposed to the summation process? Most developers only test the first part.

Another thing to keep in mind is the concept of conditionals in code. For each conditional another test has to be written to properly test the other side of the condition. If the testing is only focused on one path, then it is most likely "happy path" testing. One additional tool that should be talked about briefly is using a metric for Cyclomatic Complexity. The Cyclomatic Complexity metric will return the total number of paths though a given method, and the number of paths corresponds to the base number of unit tests needed to test the method. This additional metric is an excellent tool for both unit testing and understanding complexity in code.

Another great side effect of code coverage is around learning code. In the following example (which may ring true to many developers reading this) the boss comes up to you and states that the State Sales Tax calculation is producing odd numbers every so often and that it needs to be fixed immediately.

However, the application in question was just inherited by the developer last week. Where is the State Sales Tax calculation and what parts of the program does it effect? Using code coverage, the developer can exercise the application focusing on the State Sales Tax and figure out all the parts of the application that deal with the State Sales Tax. Not only does this save many cycles of looking through code, but it also gives an excellent pointer to the number of unit tests to be created to validate the new Sales Tax Calculation, and also helps protect against rework with introduction of regression bugs.

These are just a few areas where code coverage can be applied to help build great software. A word of caution, while code coverage is a great tool, it does produce numbers; numbers that can be distorted, numbers that can be misinterpreted, numbers that can lead developers, managers, and companies down a bad path. While code coverage will tell you exactly what lines of code are executed, or what percentage of coverage has been obtained, the one thing code coverage will not do is tell you how good the code is being executed. A developer could have 100 percent coverage, but the code being tested might not work, produce bad results, or just be plain wrong. Always keep that in perspective when using any metric or tool. High coverage rates do not ensure good code.

Once you get the use of code coverage down, it can be used for trending. This is very useful for program managers and companies as a whole. For example, say that starting on week three we had 40 percent total coverage with unit test, week four we had 50 percent of coverage with unit tests, and week five we had 30 percent coverage with unit tests… what gives? Well from week three to four it may have been that the amount of code stayed about the same and more unit tests were created, and between weeks four and five a lot of code was added and very few unit tests were added. It may even mean that some unit tests are being removed. This knowledge can be used to help set goals for the team on the level of coverage they want to achieve. It can also help figure out if problems may be occurring before the developers come back and state they have an issue, allowing for adjustments to be made.

Having knowledge is power, especially when that killer bug comes along and you can point to the amount of unit testing in place, and how the killer bug will not occur again because of the new unit test for that bug. In the same breath, having the knowledge that only 50 percent of the code is getting tested means that the other 50 percent is not, and at a minimum, managers can be made aware of the potential issues or unknowns with the application.

We've looked at the various ways in which code coverage tools work and how they can be used with Unit Testing. This should give a little insight into the many ways in which a good code coverage tool is essential in writing, understanding and implementing good code. ■