Lowering the Entry Barrier to Test Automation

Summary: Software process experts have long expounding the benefits of test automation in software development, yet few implement and those often fail to gain the expected benefit. With testing vital to the quality and success of a software project, TopCoder's Blake Tolbert shares tips and techniques on implementing a process that can drive large returns from relatively small upfront investment—creating an environment where developers learn from each other and strive to innovate their processes.

While working primarily with Fortune 1000 enterprises, a glaring majority of companies I have assisted do not have a testing strategy at all, and the rest achieve little of the benefit expected.

With the rise of outsourced, community built, and distributed developed software, testing is even more vital to the quality and overall success of a software project. It's clear that adoption of effective test automation at the enterprise is necessary, but implementation and adoption is difficult. In the following discussion, I'll explore and address the barriers to entry and how to overcome them, by sharing hands-on experience and advice from establishing and working with a wide variety of successful testing environments. By looking at the common strategies that led to success, we identify four core strategic areas to address for test automation.

By focusing in upon the Tools, Process, Culture, and Measurement, each of which must be aggressively implemented to overcome the test automation barrier to entry, you'll also see how adoption of each helps implementation of the others, in a complementary, intertwined fashion. Before going into further detail, lets define the areas I refer to in order to avoid any confusion:

- **Tools** – Writing unit tests has never been easier if given the correct tools. Test automation frameworks exist for virtually every development environment and continuous integration applications can analyze tests without any human intervention.
- **Process** – Testing must be ingrained in the overall development process. This might sound harsh but developers will not write tests if not required to do so as part of a process.
- **Culture** – Test writing is viewed negatively by most developers. It is easy to view tests as utilizing extra time that could be spent writing real code. Common ways to improve the test automation culture are discussed in this article.
- **Measurement** – The other strategic areas of test automation depend on or assist in the gathering of concrete measurements. The most common measurements are critical to overall success, as we'll discuss below.

If an organization is lacking in any of these four areas, test automation is likely to end up as an afterthought, as we continue to see time and time again. Now that we're on the same page, we can explore an overall test automation implementation strategy.

Culture is Critical

Organizations must create an environment that promotes testing. Developers must be willing to write tests and other stakeholders should understand the benefits of automated testing. Simple steps to enable this include:

- **Publish test metrics** – Valuable test metrics should be easy to gather given the correct tools and process. An automated weekly email with code coverage statistics, lines of test code, and other project specific metrics can be a source of pride for a development team and keep product sponsors up-to-date.
- **Set code coverage goals** – Setting clear code coverage goals provides guidance for developers. Saying "all committed code must have 80% code coverage unless otherwise approved" is much more effective than "write tests." The testing tools and source code repositories available for most platforms can automate this workflow.
- **Review test strategy** – Monthly or quarterly test automation strategy meetings provide a forum for feedback and process improvement.
- **Confidence** – The above steps create confidence throughout the organization which further promotes the culture. Once this cycle is started the results can be fantastic.

Testing Automation Starts with Process

Development organizations often provide testing tools to developers without a mandate to write tests, and then wonder why those developers are not creating extra work for themselves. Tools seldom make a difference without establishing the right process and approach in which to use them. Before implementing your organization's testing process, a few key topics to consider are:

- **Test-driven development (TDD)** – Test driven development involves writing unit tests, watching them fail, and then implementing the source code required to make the test pass. While many organizations do not utilize TDD, I believe every developer should experience it at least once. Among other benefits, TDD makes developers contemplate the purpose of their source code before they write it.
- **Non-traditional development organizations** – With the rise of offshore, community, and distributed development, test automation process is even more critical. In these environments, agreements on how tests will be written and how test success will be measured should be determined before development begins. Code coverage and number of tests can also be a valuable progress indicator.

Leverage Existing Tools

The last few years have provided an explosion in the number and quality of software tools available to assist with automated testing. With such a wide array of tools available, I'll touch upon the primary families of test automation software to point readers in the right direction, but it's important to compare test automation tools as they apply to your organization's personal needs and quality strategy. And while these tools have

traditionally been separate solutions, recent products such as Microsoft® Visual Studio Team System are offering more comprehensive bundled solutions.

- **Unit testing framework** – This is the actual software library that allows test to be easily written and executed. Popular test frameworks include JUnit for Java™, and NUnit and MbUnit for .NET.
- **Code coverage** – A code coverage tool will execute unit tests and determine which lines of application source code are exercised ("covered"). Popular tools include Cobertura for Java and NCover for .NET.
- **Build tools** – This provides one "script" to build the source code, execute the tests, calculate code coverage, and various other tasks. Popular build tools include Apache Ant for Java and NAnt for .NET.
- **Source code repository** – This is where all source code is cataloged. Popular repositories include Subversion® and Microsoft Team Foundation Server.
- **Continuous integration** – These tools continuously build and test the source code from the source code repository. Combined with a build script these can provide periodic reporting of test metrics.
- **Miscellaneous** – The above tools represent the core for an automated testing strategy. Auxiliary tools exist to further ease the automated testing leap (such as integrated development environments, mock libraries, and code analysis tools).

Metrics lead to Measurement

Test automation metrics can help teams set goals and measure progress. Most successful test organizations track and publish test automation metrics, which inspire pride and competition among their teams. Most of the build tools mentioned above support a task to create a daily report and automatically email it to team members. As you consider what metrics to keep in order to drive measurement and analysis of testing, consider the following, most common measurements:

- **Source code coverage** – This is the percentage of application code that is exercised by automated tests. Most organization set a code coverage goal for all code of 80 to 90 percent.
- **Total number of tests or total lines of test source code** – These absolute tracking metrics such as KLOCs (bugs per thousand lines of code) can provide a measure of progress when tracked over time.
- **Velocity** – When the total number of tests or lines of test source code is tracked over a period of time a team can measure their "velocity" – the number of tests written per day, and/or week. This is an especially useful metric for agile teams that need to estimate work done per iteration.
- **Test code to application code ratio** – The ratio of the absolute lines of source code in tests versus the application is another popular goal. A higher ratio can be misleading but, in general, it provides confidence that the application is sufficiently tested.

Identifying Where Automation Benefits Most

While the experts agree that automated testing is an effective practice, it's important to remember that it's not a silver bullet for success. To ensure a better result, consider the following issues before your organization starts its test automation plan:

First, writing tests is more time consuming up front. While it's been shown that this is outweighed by future reductions in bug fixing and maintenance, it's important to note that even with the help of automation tools, the original development will take longer.

In addition, automated tests do not eliminate the need for manual testing. Some tests are not easily automatable (or can't be automated at all). For instance, simulating a network outage. For these reasons, it is usually impossible to achieve 100% code coverage, so include manual regression tests as usual in your application project plans.

Finally, a passing automated test does not guarantee that the software is behaving correctly or that the code is well written. Software engineering practices such as verifying requirements, gathering user feedback, and code reviews are still vital to a project's success.

Making It Happen

Despite the overwhelming agreement of software process experts and the evidence that automated unit testing significantly increases software quality, a large percentage enterprise software organizations still do not implement automated tests and their projects continue to be behind schedule and bug-prone. While some organizations consciously make this choice, others simply cannot convince developers that the reward is worth the extra time involved. I believe (and is more and more the thought of the software community) that this is negligent on the part of these organizations.

Using the tools and techniques shown in the article, development groups can create leverage and see large returns for a relatively small upfront investment. The ideas and tools here are just a base of what can turn a team that is always putting out fires into a well-oiled machine. Once ideas like automated testing and continuous integration are implemented teams usually want to embrace more forward-thinking development practices such as those touched on in this article. That is a side benefit -- an environment where developers learn from each other and the community and strive to innovate their processes.

About the Author

In his role as Project Manager for TopCoder, Blake Tolbert is responsible for specification, implementation, and delivery of enterprise-class Microsoft .NET and Java software applications. TopCoder (www.TopCoder.com) is the world's largest competitive software development community with over 185,000 developers representing more than 200 countries.