An Agile Tool Selection Strategy for Web Testing Tools

Articles

Posted by:
Posted on : 2007/10/24 5:50:00

Selecting a test automation tool has always been a daunting task. Let’s face it, just the thought of automating tests can be daunting! The selection of tools available today, especially open source tools, is positively dazzling. In the past several years, “test-infected” developers, not finding what they need in the vendor tool selections, have created their own tools. Fortunately for the rest of us, many are generous enough to share them as open source. Between open source tools and commercial tools, we have an amazing variety from which to choose.

To avoid that deer-in-the-headlights feeling, consider taking an ‘agile’ approach to selecting web testing tools. Plan an automation strategy before you consider the possible tool solutions. Start simple, and make changes based on your evolving situation. Here are some ideas based on experiences I’ve had with different agile (and not so agile!) development teams. Even if your team doesn’t use agile development practices, you’ll get some useful tips.

Author: Lisa Crispin, http://lisa.crispin.home.att.net/

An Agile Test Automation Strategy

First of all, your team should consider your testing approach. When I say ‘team’, I’m thinking of everyone involved in developing and delivering the software, which in your case might be a virtual team. When do you write tests? Who writes them? How should the test results be delivered? Who needs to be able to look at the test results, and what should they be able to learn from them? What kind of tests need to be automated, and when? Do you have other tedious tasks, such as populating test data or looking through version control system output, that you’d love to automate?

Back in 2003, my current team had no test automation at all, and a buggy legacy web-based J2EE application. We desperately needed to automate our regression tests, since the manual regression tests took the whole team a couple of days to complete, and we were delivering new code to production every two weeks. We had decided to start rewriting the system, developing new features in a new architecture, while maintaining the old code, but this would be impossible without a safety net of tests.

We committed to using to test-driven development for a number of reasons, one being that automated unit tests have the highest return on investment of any automated test. We went a step further, and decided to also use ‘customer-facing’ tests and examples to help drive development. We’ve found that one example is worth pages of narrative requirements! We wanted to be able to write high-level, big-picture test cases before development starts, and then write detailed executable test cases concurrent with development so that when coding is finished, all the tests are passing.

Meanwhile, we required some kind of ‘smoke test’ regression suite for the legacy application, to make sure that critical parts kept working. Due to the old code’s architecture, we decided these would have to be done through the GUI. We wanted all of our tests to run during our continuous build process, which was automated using CruiseControl, so we’d have quick feedback of any regression failures.

Quick and easy-to-read notification of whether tests passed or failed was important to us. Ideally, our build would include these results in an email. In the event of a failure, we wanted to be able to quickly drill down to see the cause.

Platform is an obvious consideration. Our build runs on Linux, and our application was running on Linux, Solaris and Windows at the time. Any test tools that, for example, only ran on Windows did not have much appeal.

Based on all these needs, we started searching for tools. Our whole team takes responsibility for quality and testing, so we all needed to agree on our automation approach and tools. Having programmers, testers, database specialists and system administrators collaborate on test automation leverages a variety of skills to help get the best solutions. I highly recommend taking a ‘whole team’ approach to deciding on a test automation strategy, choosing and implementing tools.

An Agile Tool Selection Strategy

The whole team approach means asking ourselves, “What skills do we have on our team?” Do any team members have extensive experience with particular test tools or types of test tools? What programming and scripting language competencies exist on the team? How much technical expertise do the testers have? How about the business people who might be reviewing or even helping to write tests?

What types of tests are you automating? Unit, integration, functional, security, or do you need to do performance or load testing? How robust do your test scripts need to be, and how much can you spend on maintenance? Are you planning to do data-driven or action keyword type tests where the tests accept a variety of input parameters and have a lot of flexibility? Or are you looking for straightforward, low-maintenance tests? Can you test at a layer below the user interface, or do you have an architecture that makes that difficult? These are all considerations when shopping for a test tool.

With a variety of test needs, consider that you may need a variety of tools. We tried to keep an open mind on what might solve a particular automation problem, and we were willing to experiment. We’d pick a tool to try for a few iterations and see how we liked it. Getting up to speed on tools to the point where you can effectively evaluate them takes time, so be sure to budget plenty of time in your planning.

Our Tool Selection Process

If your team has specific needs that may not easily be met by a generic test tool, and you have the skill set to accomplish it, consider “growing your own”. This way, you get a tool is customized to your needs and integrates well with your application. Many teams have chosen this

route, which is why there are so many excellent open source tools available today. If you ‘home brew’, you still need to consider your test tool requirements. For example, if non-programmers need to specify tests, you’ll have to develop an interface to allow that.

In our case, our financial web-based application didn’t seem all that unique, and we were a tiny team without a lot of bandwidth for tool development. Our management included funds for test tools in the budget, so we looked for an outside solution.

To illustrate how your tool selection process might work, come back in time with me to late 2003 / early 2004 when we started our search. If we started from scratch in 2007, there would be even more options to consider! A couple of great places to start your search for web testing tools are www.softwareqatest.com/qatweb1.html and www.testingfaqs.org, which list both commercial and open source tools, and www.opensourcetesting.org, which provides information about all kinds of open source software testing tools. Another good resource are members of your local testing/QA user group, and testing-related mailing lists such as http://groups.yahoo.com/group/agile-testing.

We had to address our various needs with tools. Unit testing was a no-brainer. With a Java application, JUnit is the obvious choice. The team got started writing unit tests right away, as they would form the foundation of our regression tests. Next on the priority list was GUI testing.

Vendor Tools

We desperately needed to get some automated smoke tests going, and thought maybe we could buy a tool that we could run with. As a tester, I have extensive experience with the capture/playback/scripting type of automation tools. Vendor tools are often a safe choice. They’re generally designed for non-technical users, who can get started fairly easily. They come with user manuals and installation instructions. Training and technical support are available for a fee.

Commercial tools are often integrated with a suite of complementary tools, which can be an advantage. Many vendors offer functional, performance and load test tools. They offer impressively robust features. However, they tend to be targeted towards test organizations, and aren’t very ‘programmer-friendly’. They can be difficult to integrate into a continuous build process. They often have proprietary scripting languages, or are limited to one scripting language such as Javascript.

I had previously used Mercury test tools, so QuickTest Pro was one option to consider among many commercial tools. Other vendor tools we could have considered, or could consider if we were looking today, are TestPartner, Rational Functional Tester, SilkTest, BadBoy, TestMaker and (one I have always wanted to try) LISA. We also tried out Seapine’s QA Wizard, since we used their defect tracking tool TestTrack. At that time, both of those capture/playback tools used proprietary scripting languages. We didn’t want to be limited to only capture/replay, as those scripts can be more work to maintain. The programmers on my team didn’t want to have to learn a new tool or new scripting language. That ruled out all the vendor tools we looked at.

Open Source Tools

We turned to open source tools. Since these were generally written by programmers to satisfy their own requirements, they’re likely to be programmer-friendly. But there are issues with open source tools as well. Support, for example. If you have a question about an open source tool, whom do you ask? Most of the open source tools we considered had mailing lists where users and developers shared information and helped each other. I checked each tool’s mailing list to see if there was lots of activity on a daily or weekly basis, hoping to see a large and active user base. I also tried to find out if users’ issues were addressed by the tool’s developer community, and whether new versions were released frequently with enhancements and fixes. Of course, with open source tools you’re free to add your own fixes and enhancements, but we knew we wouldn’t have the bandwidth to do this at first.

Open source tools have a wide range of learning curves. Some assume programming proficiency. This is fine if everyone using the tool is able to achieve that level of competence. Others are geared to less technical users, such as testers and analysts. Some have user documentation on a par (or even better) with good vendor tools, and others leave the learning more up to the user. Some even have bug tracking systems, and the developers actually fix bugs! Think about the level of support and documentation you will need, and find a tool that provides it. We were looking for a tool that came with a lot of help.

Our GUI Tool Search

One example of a tool we researched, but didn’t try out, was JWebUnit. Since this tool lets you create scripts with Java, it appealed to the programmers on the team. At the time (2003), it didn’t seem to have as many users or as much mailing list activity as other open source test tools. We considered other Java-based tools, such as HtmlUnit, which is widely used. I had used a similar tool, HTTPUnit, before, and had liked it well enough. However, all these Java-based tools were a problem for my severely limited Java coding skills. While we anticipated that the programmers would do a large percentage of automating the customer-facing tests, I needed to write most of the GUI test scripts. I wanted to get a smoke test suite to cover the critical functionality of the legacy system, while the programmers got traction on the unit test side. We needed a programmer-friendly tool, but also a Lisa-friendly tool.

We considered scripting tools such as WATIR and scripting languages such as Ruby. I’d used TCL to write test scripts on a prior team, and I like the flexibility of scripting languages. We didn’t have any Ruby experts on the team, and although I was eager to learn it, the time it would take was an obstacle.

We looked for something that required less OO programming proficiency. I’d heard good things about Selenium, but at the time it had some limiting factor such as being difficult to integrate into our build process. Another tool that would have been a strong contender, but either it wasn’t available yet or I just didn’t know about it, is Jameleon.

After much research, we decided to try Canoo WebTest. This tool, based on HtmlUnit, uses XML to specify tests, and the scripts run via Ant. Since we use CruiseControl for our builds, it was simple to integrate the WebTest scripts with our continuous build. Being used to the features found in commercial tools, WebTest at first seemed a bit simplistic to me. At the time, it didn’t support things like if logic, except by including scripts written in Groovy or other scripting languages. It didn’t look easy to do data-driven tests with WebTest. However, we liked the idea that the tests would be so simple and straightforward, we wouldn’t have to test our test scripts.

WebTest seemed a good choice for creating a smoke test suite.

The programmers were comfortable with WebTest, since they were all familiar with XML. If a test failed, they’d be able to understand the script well enough to debug the problem. They could easily update or write new tests. We decided to try it for a few iterations. Since the programmers were busy with learning how to automate unit tests, it was helpful to have a GUI test tool that was easy for me to learn. I implemented it with some help from our system administrator. We soon had two build processes, one running all the unit tests, and the other running the slower WebTest scripts. It took about eight months to complete enough scripts to cover the major functionality of the application. These scripts have caught many regression bugs, and continue to catch them today. The return on our investment has been awesome.

Extending Our Coverage

Automated GUI test scripts are by far the most fragile and expensive to maintain, although WebTest’s features minimize the need for changes. These scripts were fine for smoke tests, to make sure nothing major or obvious in the application was broken, but we didn’t want to do detailed functional testing this way. Also, we still needed a tool to support our plan to drive development with customer-facing, executable tests and examples.

Once we had traction both at the unit test and GUI test level and could take a breath, we looked for a tool that filled up the big gap in the middle. We looked at FIT and FitNesse (which is essentially FIT using a wiki for the IDE) since they allow a non-programming user to write test cases in a tabular format and programmers to easily write fixtures to automate them. These tools basically replace the UI. They allow you to send test inputs to the code, operate on them, return actual results and compare them automatically with expected results. The results turn red, green or yellow, and we love color coding. Both tools have a large user base and active mailing lists. We liked FitNesse’s wiki component, so we decided to try it.

FitNesse turned out to suit our needs for documenting not only the features we developed, but other information about maintaining the application. We found that it was easy to learn how to define test cases and write the fixtures to automate them. Integrating the test suites into our build process took longer, but was doable. It was easy to write both high level tests to give the big picture, and executable tests to capture the detailed requirements.

Sometimes you get unexpected benefits from tools. We understood that creating FitNesse tests would require work from both a tester, to specify test cases, and a programmer, to automate them. We found that this enforced collaboration enhanced communication within the team. The resulting communication flushed out misunderstandings and wrong assumptions early in the development process. Test tools aren’t just for automation!

Be Open to Experimentation

Several months later, we had good coverage on the GUI side from our smoke test scripts, a safety net of unit tests for new code (plus the benefits of TDD), and used FitNesse tests to help guide development. We still had unfulfilled testing needs.

One difficulty was setting up test data. We had test databases, but creating data to achieve a specific scenario was often a tedious manual task. When we hired a tester with extensive Perl experience who was also interested in learning Ruby, we decided to give WATIR a try. Although we already had a GUI test tool, our new WATIR scripts allowed us to quickly create any combination of data we needed, making it much easier to do serious exploratory testing. No doubt, we could have probably done the same thing with the tools we already had, if we had spent the time researching how to do it. Having someone with skills to get us all up to speed with WATIR scripts made that the path of least resistance, although we had to plan time to learn Ruby. (I enjoyed learning Ruby, and enjoyment is an important factor too!)

Another item missing from our toolbox was a load test tool. When we needed a way to do load testing a couple of years down the road, we started doing some research. I asked the members of the agile-testing Yahoogroup for load test tool suggestions. Recommendations included Httperf, Autobench, OpenWebLoad, Apache Bench, Apache JMeter and The Grinder. After considering factors such as scripting language, learning curve, result content and formatting, recommendations from other users, and compatibility with our application, we budgeted time for trial runs of both JMeter and The Grinder. We had the best results with JMeter, a Java desktop application, and found that a tool called Badboy helped us create the JMeter scripts. We liked the statistics it generated, and how they were displayed. We were productive with JMeter quickly and were happy with our choice. As with all our tools, though, we’re always open to new developments on the tool front.

Even after you have test automation in place, there is so much development of new and existing tools that it pays to keep up with what’s new. Monitor mailing lists such as http://groups.yahoo.com/group/agile-testing, attend local user group meetings, read online and print publications (such as this one!) and attend conferences when possible to catch the latest tool buzz. Recently a couple of our team members saw demos of Selenium, and the agile-testing mailing list had posts from happy Selenium users. It has some potential advantages for us, so when we can budget time for it, we’ll check it out. We don’t intend to throw any existing tools away, since they’re still working for us. But if there’s a better way to automate particular types of tests, we’re open to trying it.

Your needs might change and evolve, as ours did. As team members come and go, your team’s skill set might change. When our WATIR expert move on, my Java programmer teammates bought Ruby books and started learning more about it. However, now there is a Java-based option, WATIJ, that might be more appropriate to our needs (there is also a .NET version, WATIN). Part of Selenium’s appeal is the ability to write the scripts in Java (which the programmers like) or Ruby (which I like). More and more tools, especially the open source ones, support multiple languages and provide more flexibility.

Tools aren’t just for automating regression tests or helping with exploratory testing, either. Scripting languages such as Ruby can automate all kinds of tedious tasks that teams need to do. Get ideas from books such as _Everyday Scripting with Ruby_ by Brian Marick.

Sometimes tools can be combined for even more advantages. Groovy scripts can be integrated into WebTest scripts to provide more flexibility. Selenium tests can be run from FitNesse. Those are just a couple of examples. Don’t limit your thinking to an individual tool’s features.

Successful Tool Implementation

What have our test tools done for our web application development? Just a few months after adopting JUnit, Canoo WebTest and FitNesse, we had a suite of regression tests that gave us a useful safety net, and our defect rate was going down dramatically. Today, three years later, we have increased our test numbers by a factor of ten or more. Our regression suites, running many times per day in two continuous builds, catch bugs at least a few times a week. Our rate of defects introduced during development is down by half. We have time and tools to do robust exploratory testing and load testing. Most importantly, but harder to measure, using our tests to drive development has resulted in features that delighted our customers.

My team put plenty of time into the research and tool adoption I’ve described here. Test automation is a big investment, but carefully done, it returns many times what you put into it. You need enough resources to first define what you need, then investigate your options, then to try out tools. Our team takes two weeks, twice a year, to devote to tasks such as researching new tools, and refactoring tests and code. This may seem like a luxury, but our management knows it helps us keep our technical debt to a minimum, so that we can improve our future productivity.

Depending what people have which skill sets, it may pay to pair people up to research or try out a test tool. A programmer and a tester could team up to try out a scripting language. A system administrator might help determine if a tool can be integrated into the team’s build process. Have brown bag sessions to brainstorm ideas, or start a book club to get ideas from publications.

Remember the ‘whole team’ approach. The team should come to a consensus on what tools to build, to try or to adopt. If a tool isn’t producing expected benefits, the team should decide whether to try a new approach or a different tool. More experienced members of the team can coach their less experienced coworkers to help everyone get up to speed on using the new tool. You may need to involve experts from outside your team to help you succeed with the new tool. People outside your team who need to use the tool, for example, to specify tests themselves, will need your help.

What If Your Team Isn’t Interested?!

“This sounds very nice,” you say, “but my team is so overloaded and busy, nobody has time to think about tools, and they don’t think it’s important to automate tests.” Not everyone has like-thinking team members. Or maybe you’re a tester on an isolated QA team that isn’t getting much support from programmers or other groups.

Don’t despair, just get creative. I once worked in a chaotic company developing a retail internet application. Despite encouragement from the development manager, the programmers automated few unit tests. The company owned licenses for a vendor GUI test tool. We hired a tester with expertise in that tool, who could automate some regression tests and teach others how to use the tool. But the tool couldn’t address all our automation needs.

The web application was written in TCL, so there were several TCL developers. On various mailing lists, I ran across several people using TCL effectively for test automation. I decided to teach myself enough TCL to create test scripts for areas we couldn’t cover with the vendor tool. While the developers weren’t interested in test automation, they were happy to help me with my TCL coding problems. This shows how it pays to leverage the expertise around you. Although far from an

ideal situation, we automated enough regression tests to free up our time for useful exploratory testing.

People, not tools, make projects successful. It doesn’t matter what tools you use or develop, as long as they help you towards your goals. Collaborating to choose and use the right test tools for your team’s situation helps allow all team members to do their best work. That’s the bottom line for test automation.

References

1. www.opensourcetesting.org
2. www.softwareqatest.com/qatweb1.html
3. www.testingfaqs.org/
4. www.io.com/%7wazmo/papers/homebrew_test_automation_200409.pdf
5. fit.c2.com
6. www.fitnesse.org
7. webtest.canoo.com
8. wtr.rubyforge.org
9. www.openqa.org/selenium
10. jameleon.sourceforge.net
11. Jakarta.apache.org/jmeter/usermanual/index.html
12. _Everyday Scripting with Ruby_, Brian Marick, Pragmatic Bookshelf, 2007 Originally published in the Summer 2007 issue of Methods & Tools