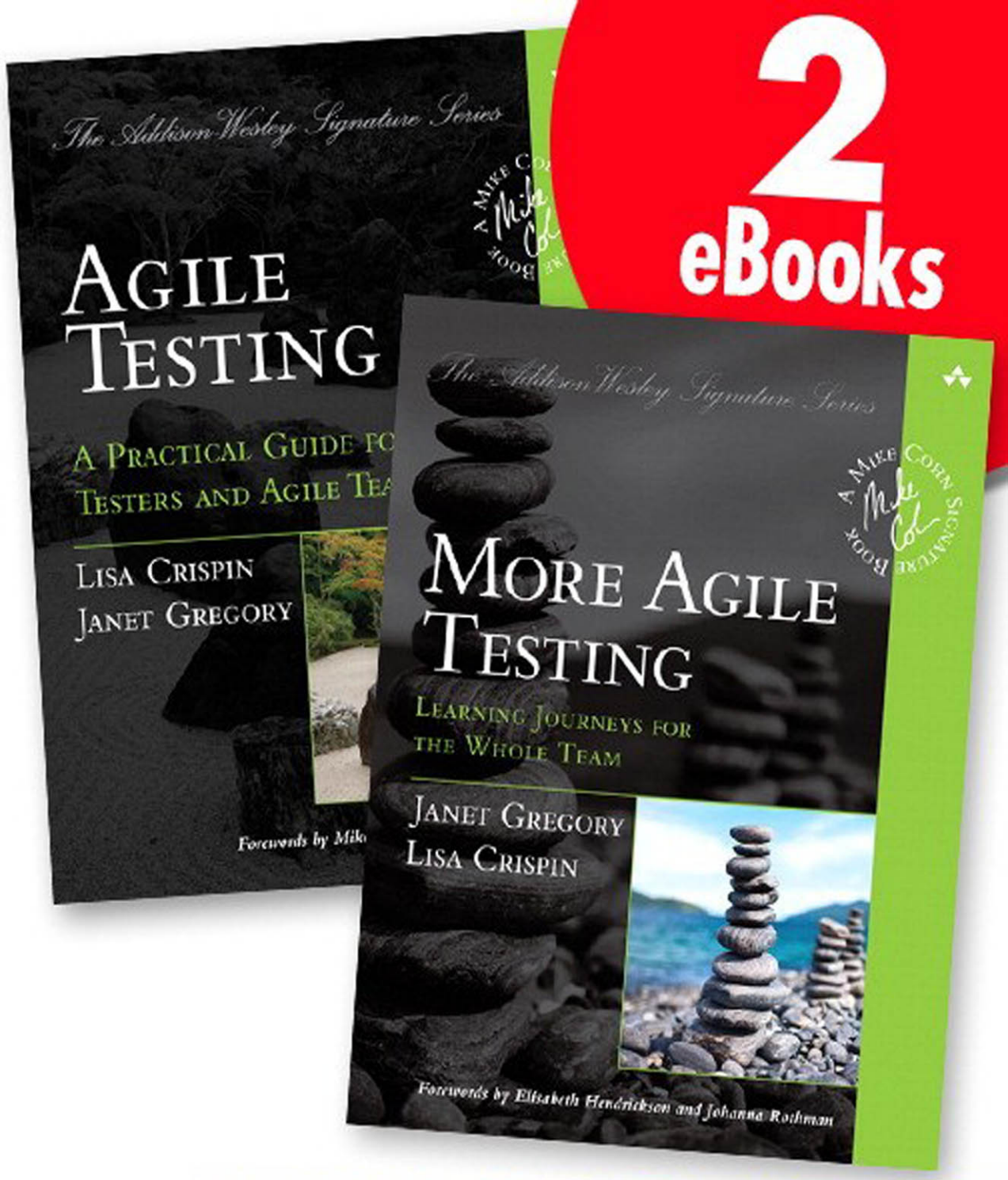

THE AGILE TESTING COLLECTION

The Agile Testing Collection

Agile Testing: A Practical Guide for Testers and Agile Teams

More Agile Testing: Learning Journeys for the Whole Team

Janet Gregory
Lisa Crispin

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

See the bibliography for a link to Patrick Wilson-Welsh's discussion of "flipping the test pyramid" right-side up.

may start out with only a few bricks. The fixtures that automate functional tests in the middle layer are easy to write if the system is designed with those tests in mind, so the sticks might pile up faster than the bricks. As teams master TDD and unit test automation, the bottom layer starts to grow. When they get traction, a team using TDD will quickly build out the brick foundation of the test pyramid.

The testing pyramid is a good place to start looking at how test automation can help an agile team. Programmers tend to focus on the bottom of the pyramid, and they need plenty of time and training to get over the "hump of pain" and get to the point where TDD is natural and quick. In traditional teams, testers usually have no choice but to automate tests at the GUI level. The whole-team approach used by agile teams means that testers pair with programmers and help them get better at writing tests, which in turn solidifies that brick foundation layer of the pyramid. Because tests drive development, the whole team is always designing for maximum testability, and the pyramid can grow to the right shape.

Programmers pair with testers to automate functional-level tests, filling out the middle layer. For example, a tester and customer may prepare a 400-row spreadsheet of test cases for a web services application. The programmer can help figure out a way to automate those tests. Different team members may have expertise in areas such as generating test data or using tools such as Excel macros, and all that knowledge spreads around the team. Working together, the team finds the best combinations of tools, test cases, and test data.

Involving the programmers in finding cost-effective ways to automate the top-level GUI tests has multiple benefits. These efforts may give programmers a better understanding of the system's "big picture," and testers can learn how to create more pliable, less straw-like GUI tests.

The more a team can work together and share knowledge, the stronger the team, the application, and the tests will become. The Big Bad Wolf won't stand a chance. Let's start by looking at what kind of tests we can automate and then at what we shouldn't even try.

WHAT CAN WE AUTOMATE?

Most types of testing you can think of benefit from automation. Manual unit tests don't go far toward preventing regression failures, because performing a suite of manual tests before every check-in just isn't practical. You can't design code test-first through manual unit tests either. When programmers

can't run tests quickly at the touch of a button, they may not be motivated enough to run tests at all. We could manually test that different units of code work together correctly, but automated component tests are a much more effective safety net.

Manual exploratory testing is an effective way to find functional defects, but if we don't have enough automated business-facing regression tests, we probably spend all of our time madly trying to keep up with manual regression testing. Let's talk about all of the different kinds of testing that can be done well with automation.

To run automated tests, you need some kind of automated framework that allows programmers to check in code often, run tests on that code, and create deployable files. Let's consider this first.

Continuous Integration, Builds, and Deploys

See Chapter 7, "Technology-Facing Tests that Support the Team," for examples of build automation tools.

Any tedious or repetitive task involved in developing software is a candidate for automation. We've talked about the importance of an automated build process. You can't build your automated test pyramid without this. Your team needs the immediate feedback from the unit-level tests to stay on track. Getting automated build emails listing every change checked in is a big help to testers because they know when a build is ready to test without having to bother the programmers.

Peril: Waiting for Tuesday's Build

In a traditional environment, it is normal for testers to wait for a stable build, even if that means waiting until next Tuesday. In an agile environment, if testers don't keep up with the developers, the stories get tested late in the game. If the developers don't get the feedback, such as suggestions and bugs, the testers can lose credibility with the developers. Bugs won't be discovered until the developers are already on another story and do not want to be interrupted to fix them until later.

Bugs pile up, and automation suffers because it can't be completed. Velocity is affected because a story cannot be marked "done" until it is tested. This makes it harder to plan the next iteration. At the end of the release cycle, your story testing runs into the end game and you may not have a successful release. At the very least, you will have a stressful release.

An automated deployment process also speeds up testing and reduces errors. In fact, the day Janet was editing this chapter, she messed up the deployment because it was a manual process. It was pretty simple, but she was new to the project and moved the file to the wrong place. Getting an automated deployment process in place went on Janet's list of things to get done right away. Lisa's team implemented its continuous integration and build framework first thing, and found it fairly easy and quick to do, although it requires continual care and feeding. Other teams, especially those with large, complex systems, face much bigger hurdles.

We've talked with teams who had build times of two hours or more. This meant that a programmer would have to wait for two hours after checking in code to get validation that his check-in didn't break any preexisting functionality. That is a long time to wait.

Most agile teams find an ongoing build longer than eight to ten minutes to be unworkable. Even 15 minutes is much too long to wait for feedback, because check-ins will start stacking up, and testers will wait a long time to get the latest, greatest build. Can you imagine how the developers working with a build that takes two hours feel as they approach the end of an iteration or release cycle? If they break any functionality, they'll have to wait two more hours to learn whether or not they had fixed it.

Many times, long builds are the result of accessing the database or trying to test through the interface. Thousands of tests running against a large codebase can tax the resources of the machine running the build. Do some profiling of your tests and see where the bottleneck is. For example, if it is the database access that is causing most of the problems, try mocking out the real database and use an in-memory one instead. Configure the build process to distribute tests across several machines. See if different software could help manage resources better. Bring in experts from outside your team to help if needed.

The key to speeding up a continuous integration and build process is to take one small step at a time. Introduce changes one at a time so that you can measure each success separately and know you are on the right track. To start with, you may want to simply remove the most costly (in terms of time) tests to run nightly instead of on every build.

A fast-running continuous integration and build process gives the greatest ROI of any automation effort. It's the first thing every team needs to automate.

When it's in place, the team has a way to get quick feedback from the automated tests. Next, we look at different types of tests that should be automated.

See the bibliography for links to build automation tools and books with more information about improving the build process.

Unit and Component Tests

We can't overemphasize the importance of automating the unit tests. If your programmers are using TDD as a mechanism to write their tests, then they are not only creating a great regression suite, but they are using them to design high-quality, robust code. If your team is not automating unit tests, its chances of long-term success are slim. Make unit-level test automation and continuous integration your first priority.

Chapter 7, "Technology-Facing Tests that Support the Team," goes into detail about some of the tools that can be used.

API or Web Services Testing

Testing an API or web services application is easiest using some form of automation. Janet has been on teams that have successfully used Ruby to read in a spreadsheet with all of the permutations and combinations of input variables and compare the outputs with the expected results stored in the spreadsheets. These data-driven tests are easy to write and maintain.

One customer of Janet's used Ruby's IRB (Interactive Ruby Shell) feature to test the web services for acceptance tests. The team was willing to share its scripts with the customer team, but the business testers preferred to watch to see what happened if inputs were changed on the fly. Running tests interactively in a semiautomated manner allowed that.

Testing behind the GUI

See Chapter 9, "Toolkit for Business-Facing Tests that Support the Team," for specific tool examples.

Testing behind the GUI is easier to automate than testing the GUI itself. Because the tests aren't affected by changes to the presentation layer and work on more stable business logic code, they're more stable. Tools for this type of testing typically provide for writing tests in a declarative format, using tables or spreadsheets. The fixtures that get the production code to operate on the test inputs and return the results can generally be written quickly. This is a prime area for writing business-facing tests, understandable to both customers and developers that drive development.

Testing the GUI

Even a thin GUI with little or no business logic needs to be tested. The fast pace of agile development, delivering new functionality each iteration, mandates some automated regression tests at the GUI level for most projects.

Tool selection is key for successful GUI automation. The automated scripts need to be flexible and easy to maintain. Janet has used Ruby and Watir very successfully when the framework was developed using good coding practices, just as if it were a production application. Time was put into developing the libraries so that there was not a lot of rework or duplication in the code, and changes needed could be made in one place. Making the code easy to maintain increased the ROI on these tests.

See Chapter 9, "Toolkit for Business-Facing Tests that Support the Team," for examples of GUI test frameworks.

A point about testability here—make sure the programmers name their objects or assign IDs to them. If they rely on system-generated identifiers, then every time a new object is added to the page, the IDs will change, requiring changes to the tests.

Keep the tests to just the actual interface. Check things like making sure the buttons really work and do what they are supposed to. Don't try to try to test business functionality. Other types of tests that can be automated easily are link checkers. There is no need for someone to manually go through every link on every page to make sure they hit the right page. Look for the low-hanging fruit, automate the things that are simple to automate first, and you'll have more time for the bigger challenges.

Load Tests

See Chapter 11, "Critiquing the Product Using Technology-Facing Tests," for examples of load test automation tools.

Some types of testing can't be done without automation. Manual load tests aren't usually feasible or accurate, although we've all tried it at one time or another. Performance testing requires both monitoring tools and a way to drive actions in the system under test. You can't generate a high-volume attack to verify whether a website can be hacked or can handle a large load without some tool framework.

Comparisons

Read more about source code management tools and IDEs in Chapter 7, "Technology-Facing Tests that Support the Team."

Visually checking an ASCII file output by a system process is much easier if you first parse the file and display it in a human-readable format. A script to compare output files to make sure no unintentional changes were made is a lot faster and more accurate than trying to compare them manually. File comparison tools abound, ranging from the free diff to proprietary tools such as WinDiff. Source code management tools, and IDEs have their own built-in comparison tools. These are essential items in every tester's toolbox. Don't forget about creating scripts for comparing database tables when doing testing for your data warehouse or data migration projects.

Repetitive Tasks

As we work with our customers to better understand the business and learn what's valuable to them, we might see opportunities to automate some of their tasks. Lisa's company needed to mail several forms with a cover letter to all of their clients. The programmers could not only generate the forms but could also concatenate them with the cover letter and greatly speed up the mailing effort. Lisa's fellow tester, Mike Busse, wrote a spreadsheet macro to do complex calculations for allocating funds that the retirement plan administrators had been doing manually. A lot of manual checklists can be replaced with an automated script. Automation isn't just for testing.

Data Creation or Setup

Another useful area for automation is data creation or setup. If you are constantly setting up your data, automate the process. Often, we need to repeat something multiple times to be able to recreate a bug. If that can be automated, you will be guaranteed to have the same results each time.

Lisa's Story

Many of our test schemas, including the ones used for automated regression suites, use canonical data. This canonical or "seed" data was originally taken from production. Some tables in the database, such as lookup tables, don't change, so they never need to be refreshed with a new copy. Other tables, such as those containing retirement plan, employee, and transaction information, need to start from Ground Zero whenever a regression suite runs.

Our database developer wrote a stored procedure to refresh each test schema from the "seed" schema. We testers may specify the tables we want refreshed in a special table called REFRESH_TABLE_LIST. We have an ant target for each test schema to run the stored procedure that refreshes the data. The automated builds use this target, but we use it ourselves whenever we want to clean up our test schema and start over.

Many of our regression tests create their own data on top of the "seed" data. Our Watir tests create all of the data they need and include logic that makes them rerunnable no matter what data is present. For example, the script that tests an employee requesting a loan from his or her retirement plan first cancels any existing loans so a new one can be taken out.

FitNesse tests that test the database layer also create their own data. We use a special schema where we have removed most constraints, so we don't have to add every column of every table. The tests only add the data that's pertinent to the functionality being tested. Each test tears down the data it created, so subsequent tests aren't affected, and each test is independent and rerunnable.

—Lisa
