Agile Product & Project Management Advisory Service
Executive Update Vol. 11, No. 24

Starting Agile Adoption:

Part III — Advantages and Pitfalls of Unit Testing

by Steve Berczuk

Automated unit testing is an essential engineering practice for successful agile software development. A related practice, test-driven (or test-first) development (TDD), takes the idea of unit testing further, mandating the writing of tests before production code as a way of ensuring good, testable design. While the benefits of automated testing seem clear, teams struggle with making the writing of unit tests routine and effective. This *Executive Update*, the last in a three-part series,[1] will explain the challenges teams encounter when adopting unit testing and provide some suggestions about how to use testing to move your agile adoption process forward.

WHAT IS UNIT TESTING?

Unit tests, written by developers, are automated and can be described as follows:[2]

- Fast, so that the overhead of running them frequently is low
- Independent, so that you can evaluate pieces of functionality independently
- Repeatable in any environment, ideally not dependent on external resources
- Self-validating, so that you can quickly determine whether something broke
- Timely, or written about the same time that the production code undergoing testing is written (more on this in a bit)

Unit tests could also be referred to as "developer tests."[3] Unit tests will be executed before every developer commits and as part of the integration build. Having unit tests, you enable some essential agile practices:

- **Continually shippable code.** Unit tests allow you to spot potential problems sooner, even before changes are committed to the repository.
- **Refactoring.** Unit tests allow you to quickly judge the impact of a change, identifying when an attempted refactoring changes functionality rather than structure.
- **Responsiveness.** Since you can quickly identify the consequences of a code for adding a new feature change, you can feel more confident in your ability to change functionality.

Developer-written tests supplement more traditional exploratory testing. Unit tests are said to encourage flexible design and limited coupling, as more loosely coupled code is easier to test in isolation. Some research[4, 5] challenges the claims that TDD always provides such benefits as better design, more robust testing, and better test coverage, but writing tests first does make it less likely that you will write code that is not testable.

Regardless of whether you want to establish a test-first or a test-early approach, more testing earlier can have many benefits. For the rest of this *Update*, early testing means "before committing production code changes to the repository." In many cases, this may lead to a test-driven approach, but the goal is to improve test coverage so that you catch issues earlier. Regardless of the approach you choose, it's important to periodically evaluate what you are doing, its impact, and how to improve. To realize the benefits, you need to understand the costs and how to overcome some technical and cultural challenges.

CHALLENGES

Some of the challenges to adopting unit testing include:

- **Testing has a cost.** Like all engineering activities, effective testing is a balance between cost, benefit, and risk. Some tests are fragile in the face of change and difficult to maintain. Some tests seem almost too trivial to write. Deciding how to balance spending effort in unit rather than integration testing can be tricky. Often, the benefits of reduced rework outweigh the cost of testing. Before evaluating the costs of unit testing you need to follow the practice for a time and gather data.

- **Legacy code is hard to test.** Many teams adopting agile methods are starting with an existing code base. In these cases, it's hard to start testing because of coupling. For this reason, Michael Feathers defines legacy code as code without unit tests.[6]
- **The role of QA changes.** In some teams, testing is considered a "secondary" skill that's best left to someone other than developers, and writing good unit tests is a skill that people on your team need to learn.

The technical issues that make testing difficult, combined with the challenges of changing from a QA-only test culture, can lead to a desire to defer writing tests once problems develop, with the result that the team never sees the benefits of unit testing. Don't underestimate the cultural impact of changing to a test-oriented culture. Assume that testing is possible. In the next sections, I will address each of these points in detail.

The Cost of Tests

It's important to acknowledge that writing code with tests will take longer than writing code without them. This is not unreasonable since a good unit test means that the you will be less likely to find problems later with the code under test and thus catch regressions sooner.

Unit tests take time to run; making builds (and thus source-code changes) take longer. While unit tests need not meet the same performance criteria as production code, one should think through whether the test is doing the right thing, and in the right way.

Start with the idea that tests add value. At the end of each iteration, identify slow-running and slow-to-implement tests and discuss what the best approach is.

Fragile Tests

In addition to the cost of implementing tests, maintaining tests has a cost. When you make a change to code and a unit test fails, there are two possibilities:

1. Your code caused a contract to be violated, and your code needs to change.
2. You are defining new functionality, and the test needs to change.

In the latter case, you might consider whether the cost of maintaining the tests is too high. Ordinarily, keeping tests up to date should not have a significant cost.

In some cases, the cost of changing a test might exceed the cost of the code change by a large margin. This is typical for code that works with user interface components. While it's easy to write a test validating that you have sent a flag that tells the server to enable a button and change the color to red, it might be more difficult to determine how to verify that the button actually exists and has the correct color in the user interface. In these cases, it may take significant length of time to write an automated test to verify a quick code change.

For these cases, it is still good practice to start with the idea of erring on the side of writing an automated test. After an iteration or two, you can evaluate whether the value the tests added was worth the cost.

If you find yourself with tests that take significantly more time to keep up to date than the original code change, consider the following:

- Are you using the right testing approach or tool? For example, if you are testing XML or HTML generation, making assertions about structure is more robust than string comparisons against a master.
- What's the cost of an error? Will the application fail in a significant and unexpected way, or is the result purely cosmetic?

In the end, unit tests can, and should, add value, but adding tests solely for the sake of increasing test count can be counterproductive.

Trivial Tests and Testing the Trivial

When adopting unit testing, team members might go to an extreme and test every method they write, including trivial methods, such as getters and setters. Team members might skip a potentially useful test because it's too hard to write. Or they might skip "trivial" tests because the code under test could be verified "by inspection." Deciding what to test and how to test is a skill that takes time to learn.

Adopting unit testing, like any new skill, is about both learning skills and adopting habits. When you first start a project, you may not be sure what tests will provide the most value for effort. Most programmers have spent a significant length of time tracking down an "obvious" problem. These experiences should provide insight into the value of erring on the side of thorough testing.

Start by writing tests for any code that is more complicated than a simple assignment. Even tests that do seemingly "dumb" things, such as validating that a configuration file has no syntax errors, can save your team time by detecting errors during build time — before changes were committed to the repository — rather than after the application is deployed. When creating the unit test seems to be taking significant effort, consider whether the fault lies with the testing strategy or the design. At the end of each iteration, review testing practices, evaluate the benefits and costs of tests that you are writing, and track test coverage.

While unit testing has costs, more often than not the benefits outweigh these costs. Until you learn the balance, err on the side of encouraging tests. It's easier to remove tests once the team is in the habit of testing, than it is to recover from the effects of missing tests.

Integration Testing

While it is possible that having an assembly of components that pass all of their unit tests work perfectly when they are first put together, this does not always happen. Either the unit tests are not as thorough as they could have been, or putting the system together adds complexity that could not be tested in a unit test context. Integration testing adds value, and you need to understand the relative value of investing in unit testing as compared to integration testing.

TDD may result in a false sense of security and therefore cause more failures at the acceptance test level,[7] as illustrated by a story from a time when I worked on a team that had fully embraced TDD. While using pair programming to fix a problem, we wrote the test and the code. All the tests ran successfully. My colleague resisted the idea of running the application, insisting that having running tests was sufficient proof that all was well. It turns out that our change had inadvertently disabled one part of the user interface. This was something we would have seen immediately had we launched the application, but our testing framework could not easily discover it. The belief that the TDD approach could ensure quality without validating that the tests were "good" can lead to a decrease in quality.

Unit tests are not a replacement for integration and exploratory testing, but they can become drivers for more unit testing. A failed integration test can point to a component that needs more unit testing, and components that are tricky to unit-test can point to areas to focus more top-down testing efforts.

Legacy Code

Legacy code is often monolithic, and setting up an test environment can have significant overhead. For this reason, Feathers defines legacy code as code without tests.[8] When working on new code, you have the option to use an approach such as TDD to make sure you have testable code. Many projects that are adopting agile are working with an existing code base, and the challenges in writing tests for legacy code can frustrate the most excited adopter of unit testing.

When working with legacy code, understand that the costs of writing tests for it will be higher than for writing tests for new code. Testing legacy code may require starting with tests that are better suited to integration or refactoring the code into smaller, more testable units. While the refactoring takes time, the benefit at the end will be more adaptable, reliable, less fragile code that others will be more willing and able to update.

Culture Change

Engineers are quick to address technical challenges. Cultural challenges are a more difficult barrier to introducing change. Since introducing developer testing will increase the (apparent) time to deliver functionality, especially at the beginning, when team members are learning new habits, you need to create an environment where it's safe to take the time to test. I say "apparent" because by having developers take more time to test and validate in an automated way, you will reduce regressions and cycle times from initial implementation to acceptance. By focusing on testing earlier, you may identify requirements risks, as tests will highlight inconsistencies or roadblocks to implementing a feature that might not be obvious with higher-level discussions.

As developers adopt more testing, the role of QA can expand by:

- Adapting a more code-centric approach to testing and writing more automated integration level tests
- Helping with requirements specification, as discussed in Part II
- Doing more interesting exploratory testing, rather than looking for issues easily found at the code level

Even in a TDD environment, testers are needed. As Forrest Shull et al. point out: "TDD does not replace skillful testers but it does free them to find serious bugs in areas related to end to end scenarios and nonfunctional system characteristics."[9]

GETTING STARTED

Like many change-adoption processes, it's better to do things than to look for reasons not to do things, so that you can form the habit. While a TDD approach can instill good discipline and practices, there may be significant cultural obstacles to implementing a TDD practice, and it may be practically impossible on teams working with a significant legacy code base. Start with an approach that asks developers to do the following:

- Consider what each coding activity will accomplish. This will be the basis of a test.
- Before committing code, evaluate how they would test this functionality, and if such a test would add value relative to the effort. If the developer decides that a test is not useful, confirm why they decided not to. This way the default position is to test rather than not test, as is usual.
- Require that each commit to the source repository have a test associated with it, or a clear statement of why they decided not to test. This provides an audit trail that can be used as data in retrospectives to help teams improve their practices.
- The important thing is to create a culture where the quest is to decide what to test, and to err on the side of testing, and evaluate your testing process at the end of each iteration. As with the rest of your agile adoption process, the iteration review and retrospective process is essential for keeping this technique on track.[10, 11]

CONCLUSIONS

While it's hard to argue against early testing, adopting developer testing has challenges both cultural and technical. The technical challenges can be solved in a fairly straightforward manner. To overcome the organizational challenges, you need to provide an environment where developers can explore testing techniques, and review and adapt based on experience, not supposition. Manns and Rising provide guidance on changing cultures.[12]

Automated developer tests can be an important tool to help your transition to agile be more effective. But such tests are only part of your transition to an agile approach to development. Exploratory testing and manual smoke testing have a role to play, too.

ENDNOTES

[1]Berczuk, Steve. "Starting Agile Adoption: Part I — Quality Assurance." Cutter Consortium Agile Product & Project Management *Executive Update*, Vol. 11, No. 16, 2010; Berczuk, Steve. "Starting Agile Adoption: Part II — Avoiding Common Pitfalls of Planning." Cutter Consortium Agile Product & Project Management *Executive Update*, Vol. 11, No. 21, 2010.

[2]Martin, R.C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[3]Rainsberger, J.B., and S. Stirling. *JUnit Recipes: Practical Methods for Programmer Testing*. Manning, 2005.

[4]Siniaalto, Maria, and Pekka Abrahamsson. "A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage." *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, IEEE Computer Society, 2007.

[5]Janzen, David, and Hossein Saiedian. "Does Test-Driven Development Really Improve Software Design Quality?" *IEEE Software*, Vol. 25, No. 2, 2008, pp. 77-84.

[6]Feathers, Michael C. *Working Effectively With Legacy Code*. Prentice Hall Professional Technical Reference, 2004.

[7]Siniaalto. See 3.

[8]Feathers. See 5.

[9]Shull, Forrest et al. "What Do We Know about Test-Driven Development?" *IEEE Software*, November/December 2010, Vol. 27, No. 6, pp. 16-19.

[10]For guidelines about lightweight retrospective techniques, see: Derby, Esther, and Diana Larsen. *Agile Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, 2006.

[11]For techniques for keeping meetings focused, see: Tabaka, Jean. *Collaboration Explained: Facilitation Skills for Software Project Leaders*. Addison-Wesley, 2006.

[12]Manns, Mary Lynn, and Linda Rising. *Fearless Change: Patterns for Introducing New Ideas*. Boston, Addison-Wesley, 2005.

ABOUT THE AUTHOR

Steve Berczuk is an engineer and ScrumMaster at Humedica, where he's helping to build next-generation clinical informatics applications based on software as a service (SaaS). The author of *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, he is a recognized expert in software configuration management and agile software development. Mr. Berczuk is passionate about helping teams work effectively to produce quality software. He has a master's degree in operations research from Stanford University, a bachelor's degree in electrical engineering from MIT, and is a Certified Practicing ScrumMaster. He can be reached at steve@berczuk.com.