Editor in Chief: **Diomidis Spinellis**
Athens University of Economics
and Business, dds@computer.org

State-of-the-Art Software Testing

Diomidis Spinellis

PROGRESS IN SOFTWARE engineering often appears agonizingly slow. Yet when you look back and take stock, you can see that the software we develop today benefits from practices that would have been considered fanciful during the implementation of the system we might aim to replace.

This is the case with software testing. A couple of decades ago, many parts of software were tested only manually or not at all. The integration of testing into development was through a wall over which developers threw the software to dedicated testers. Test coverage analysis and A/B testing were techniques many of us only heard of in college and never saw applied in practice.

The most striking sign of progress is visible in industrial practice, which used to trail academic research at an embarrassing distance but now often leads the way.[1] So here's how to test software like a pro.

Best Practices

First, pair the routines you write with their unit tests. These tests exercise the code in isolation, preventing problems from surfacing during integration. They also promote more modular design, protect you during refactoring, and document how the code you write is supposed to be used. So important are these tests that Michael Feathers considers software lacking them to be legacy code.[2] Adopt a framework, such as one from the xUnit family, for writing and running your unit tests. There are (more than) plenty to choose from; various modern languages, such as Go, Python, Ruby, and Rust, even include unit-testing support as part of their standard library.

Some of you might decide to go even further by adopting test-driven development (TDD): progressing step-by-step by writing a test based on the software's requirements and then implementing the code that implements the test. This development style helps you focus on the requirements from the outset, drives you to design testable software, and ensures that each feature is coupled with its test code. TDD also helps your organization stay honest regarding testing, by minimizing the temptation to skimp on the implementation of tests after the code gets written.

Continue by establishing what Mike Cohn called a *test pyramid*.[3] At the bottom, write plenty of unit tests to ensure that your methods are correct. These are relatively cheap to write, are robust in the face of software changes, and can run very fast. Supplement them with a selective dose of component and integration tests that run below the application's user interface. In modern applications you should be able to write these easily through (for example, REST—Representational State Transfer) service calls. At the pyramid's top, write a few end-to-end tests that exercise the user interface. These can be expensive to write, brittle, and slow, so exercise restraint in what you test here.

Strive to automate all types of tests. This minimizes their cost, simplifies their running, and offers you many opportunities to measure and optimize the process. Automated tests are the machine oil that keeps the development engine running smoothly. As an added bonus, test automation provides more meaningful and stimulating tasks to testers, letting them focus on the tests' quality and process optimization, rather than miring them in the drudgery of manually executed test cases.

Code and its tests tend to decay over time. So, ensure that both are always up to scratch by running your tests during continuous integration (CI). Most CI frameworks support this functionality; all you have to do is configure it. By running tests after each commit, you minimize unpleasant surprises during integration. Code committers get an immediate warning if their code broke their own or somebody else's tests. I've seen that this process, when applied to thoroughly tested code, makes it a lot easier to onboard new developers into a project. With guard railings protecting all parts of the code, the chance of somebody driving over the cliff is minimized.

This brings me to another important practice: test coverage analysis. With this, you want to measure and thoughtfully (rather than blindly[4]) evaluate what code and what percentage of code are covered by tests. Achieving 100 percent coverage is neither easy nor a guarantee of faultless code. However, low or decreasing levels of test coverage are a warning sign that something is amiss. Coupled with automated testing, the measurement of code coverage as part of your CI process with tools such as Coveralls (coveralls.io) can help guide your organization toward a test quality baseline.

When it comes to testing the user experience and usability, automation is more difficult. Nevertheless, there are still methods that can help you a lot. In particular, consider A/B testing, in which you deploy a given feature to only a subset of your user base and compare the two groups' behavior. In services delivered over the web, deploying both versions can be simplified through software option switches, which enable a feature only for specific users. Measuring the two versions' outcomes is also easy; just have your server keep a detailed log of user interactions.

As is always the case in software engineering, the icing on the software development cake entails measurement, evaluation, and improvement. When testing, first examine your test cases' effectiveness. A successful test case is one that catches a bug. For example, testing a class's getters and setters is rarely worthwhile; focus instead on eliminating error-prone boilerplate code with approaches such as those supported by Project Lombok (projectlombok.org).

Two other metrics to examine are the time it takes for test cases to run and their brittleness. Large code

EDITORIAL STAFF

Lead Editor: Meghan O'Dell, m.odell@computer.org
Content Editor: Dennis Taylor
Staff Editors: Lee Garber and Rebecca Torres
Publications Coordinator: software@computer.org
Lead Designer: Jennie Zhu-Mai
Production Editor: Monette Velasco
Webmaster: Brandi Ortega
Multimedia Editor: Erica Hardison
Illustrators: Annie Jiu, Robert Stack, and Alex Torres
Cover Artist: Peter Bollinger
Director, Products & Services: Evan Butterfield
Senior Manager, Editorial Services: Robin Baldwin
Manager, Editorial Content: Carrie Clark
Senior Advertising Coordinator: Debbie Sims, dsims@computer.org

WELCOME NEW ED BOARD MEMBER

We welcome Mik Kersten to our editorial board. He's in charge of our new On DevOps department. On DevOps will examine the discipline of architecture and systems thinking beyond just the code, to encompass all the other artifacts and processes involved in software delivery.

Kersten is the founder and CEO of Tasktop and drives the company's strategic direction and a culture of customer-centric innovation. His research interests focus on value stream architecture. Before Tasktop, Kersten launched a series of open source projects that changed how developers collaborate. As a research scientist at Xerox PARC, he created the first aspect-oriented development environment. He received a PhD in computer science from the University of British Columbia. Kersten has been named a JavaOne Rock Star speaker and one of the IBM developerWorks Java top 10 writers of the decade. He was selected as one of the 2012 Business in Vancouver 40 under 40 and has been a World Technology Awards finalist in the IT Software category. Contact him at mik@tasktop.com or follow @mik_kersten.

bases are often plagued by long testing times, unreliable test results, and other "test smells." You can reduce testing times by having test execution tools intelligently select which test cases to run after a specific commit. Increase test stability by flagging and correcting nondeterministic test cases and implementing a stable staging environment.

Years ago, testing used to be the ugly duckling of software development. Given that nowadays testing can stand eye-to-eye with any other software development process, I can hear you asking, which factors have driven the steady progress in testing over the past decades? My take here is that software code's rising size and complexity, greater demands regarding development speed and agility, and increased heterogeneity and geographic distribution of software teams and their components have forced us to develop and adopt better testing practices. This means that state-of-the-art software testing is now a mandatory part of software development.

References

1. V. Garousi et al., "What Industry Wants from Academia in Software Testing? Hearing Practitioners' Opinions," *Proc. 21st Int'l Conf. Evaluation and Assessment in Software Eng.* (EASE 17), 2017, pp. 65–69.
2. M.C. Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2005.
3. M. Cohn, *Succeeding with Agile: Software Development Using Scrum*, Addison-Wesley Professional, 2009.
4. B. Marick, "How to Misuse Code Coverage," *Proc. 16th Int'l Conf. Testing Computer Software* (ICTCS 99), 1999, pp. 16–18; www.exampler.com/testing-com/writings/coverage.pdf.