Unit Testing Best Practices: How to Get the Most Out of Your Test Automation

Take the smallest components of your application to task with proper unit testing best practices and methods.

Brian McGlauflin, Jun. 25, 19 (https://blog.parasoft.com/unit-testing-best-practices-getting-the-most-out-of-your-test-automation)

Unit testing is a well-known practice, but there's lots of room for improvement! In this post, the most effective unit testing best practices, including approaches for maximizing your automation tools along the way. We will also discuss code coverage, mocking dependencies, and overall testing strategies.

What is Unit Testing?

Unit testing is the practice of testing individual units or components of an application, in order to validate that each of those units is working properly. Generally, a unit should be a small part of the application — in Java, it is often a single class. Note that I am not strictly defining "unit" here, and it is up to the developer to decide the scope of tested code for each test.

People sometimes contrast the term "unit testing" with "integration testing" or "end-to-end testing." The difference is that generally, unit testing is done to validate the behavior of an individual testable unit, whereas integration tests are validating the behavior of multiple components together, or the application as a whole. Like I said, the definition for what constitutes a "unit" is not strictly defined, and it's up to you to decide the scope for each test.

Why Unit Test?

Unit testing is a proven technique for ensuring software quality, with plenty of benefits. Here are (more than) a few great reasons to unit test:

- Unit testing **validates** that each piece of your software not only works properly today, but continues to work in the future, providing a solid foundation for future development.
- Unit testing **identifies defects at early stages** of the production process, which reduces the costs of fixing them in later stages of the development cycle.
- Unit-tested code is generally **safer to refactor**, since tests can be re-run quickly to validate that behavior has not changed.
- Writing unit tests forces developers to consider how well the production code is designed in order to make it **suitable for unit testing**, and makes developers look at their code from a **different perspective**, encouraging them to consider corner cases and error conditions in their implementation.

- Including unit tests in the **code review process** can reveal how the modified or new code is supposed to work. Plus, reviewers can confirm whether the tests are good ones or not.

It's unfortunate that all too often, developers either don't write unit tests at all, don't write *enough* tests, or they don't *maintain* them. I understand — unit tests can sometimes be tricky to write, or time-consuming to maintain. Sometimes there's a deadline to meet, and it feels like writing tests will make us miss that deadline. But not writing enough unit tests or not writing good unit tests is a risky trap to fall into.

So please consider my following best-practice recommendations on how to write clean, maintainable, automated tests that give you all the benefits of unit testing, with a minimum amount of time and effort.

Unit Testing Best Practices

Let's look at some best practices for building, running, and maintaining unit tests, to achieve the best results.

Unit Tests Should Be Trustworthy

The test must fail if the code is broken and only if the code is broken. If it doesn't, we cannot trust what the test results are telling us.

Unit Tests Should Be Maintainable and Readable

When production code changes, tests often need to be updated, and possibly debugged as well. So, it must be easy to read and understand the test, not only for whoever wrote it, but for other developers as well. Always organize and name your tests for clarity and readability.

Unit Tests Should Verify a Single-Use Case

Good tests validate one thing and one thing only, which means that typically, they validate a single use-case. Tests that follow this best practice are simpler and more understandable, and that is good for maintainability and debugging. Tests that validate more than one thing can easily become complex and time-consuming to maintain. Don't let this happen.

Another best practice is to use a minimal number of assertions. Some people recommend just one assertion per test (this may be a little too restrictive); the idea is to focus on validating only what is needed for the use-case you are testing.

Unit Tests Should Be Isolated

Tests should be runnable on any machine, in any order, without affecting each other. If possible, tests should have no dependencies on environmental factors or global/external state. Tests that have these dependencies are harder to run and usually unstable, making them harder to debug and fix, and end up costing more time than they save (see **trustworthy**, above).

Martin Fowler, a few years ago, wrote about "solitary" vs. "sociable" code, to describe dependency usage in application code, and how tests need to be designed accordingly. In his article, "solitary" code doesn't depend on other units (it's more self-contained), whereas "sociable" code does interact with other components. If the application code is solitary, then the test is simple, but for sociable code under test, you can either build a "solitary" or "sociable"

test. A "sociable test" would rely on real dependencies in order to validate behavior, whereas a "solitary test" isolates the code under test from dependencies. You can use mocks to isolate the code under test and build a "solitary" test for "sociable" code. We'll look at how to do that below.

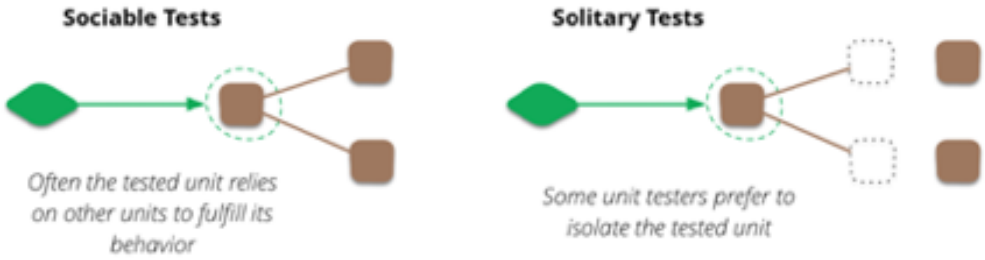


Figure 1: *Sociable vs Solitary Tests. Source: Martin Fowler, 2014, "Unit Test"*

In general, using mocks for dependencies makes our life easier as testers, because we can generate "solitary tests" for sociable code. A sociable test for complex code may require a lot of setup and may violate the principles of being isolated and repeatable. But since the mock is created and configured in the test, it is self-contained, and we have more control over the behavior of dependencies. Plus, we can test more code paths. For instance, I can return custom values or throw exceptions from the mock, in order to cover boundary or error conditions.

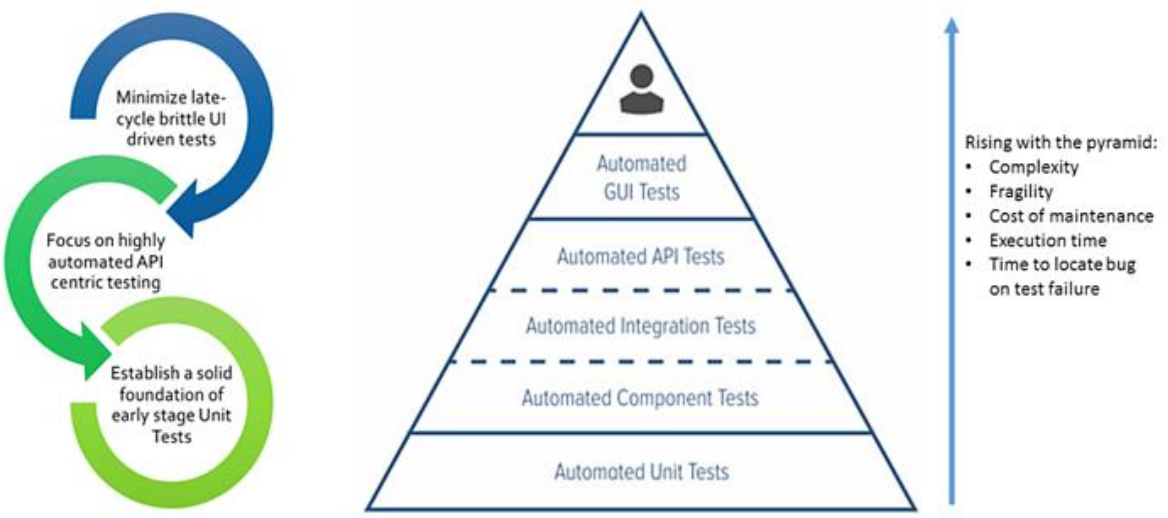
Unit Tests Should Be Automated

Make sure tests are being run in an automated process. This can be daily, or every hour, or in a Continuous Integration or Delivery process. The reports need to be accessible to and reviewed by everyone on the team. As a team, talk about which metrics you care about: code coverage, modified code coverage, number of tests being run, performance, etc.

A lot can be learned by looking at these numbers, and a big shift in those numbers often indicates regressions that can be addressed immediately.

Use a Good Mixture of Unit and Integration Tests

Michael Cohn's book, *Succeeding with Agile: Software Development Using Scrum*, addresses this using a testing pyramid model (see illustration in the image below). This is a commonly used model to describe the ideal distribution of testing resources. The idea is that as you go up in the pyramid, tests are usually more complex to build, more fragile, slower to run, and slower to debug. Lower levels are more isolated and more integrated, faster, and simpler to build and debug. Therefore, automated unit tests should make up the bulk of your tests.



Unit tests should validate all of the details, the corner cases and boundary conditions, etc. Component, integration, UI, and functional tests should be used more sparingly, to validate the behavior of the APIs or application as a whole. Manual tests should be a minimal percentage of the overall pyramid structure, but are still useful for release acceptance and exploratory testing. This model provides organizations with a high level of automation and test coverage, so that they can scale up their testing efforts and keep the costs associated with building, running, and maintaining tests at a minimum.

Unit Tests Should Be Executed Within an Organized Test Practice

In order to drive the success of your testing at all levels, and make the unit testing process scalable and sustainable, you will need some additional practices in place. First of all, this means **writing unit tests as you write your application code**. Some organizations write the tests before the application code (test-driven or behavior-driven programming). The important thing is that tests go hand-in-hand with the application code. The tests and application code should even be reviewed together in the code review process. Reviews help you understand the code being written (because they can see the expected behavior) and improve tests too!

Writing tests along with code isn't just for new behavior or planned changes, it's critical for bug fixes too. **Every bug you fix should have a test that verifies the bug is fixed**. This ensures that the bug stays fixed in the future.

Adopt a zero-tolerance policy for failing tests. If your team is ignoring test results, then why have tests at all? Test failures should indicate real issues...so address those issues right away, before they waste QA's time, or worse, they get into the released product.

The longer it takes to address failures, the more time and money those failures will ultimately cost your organization. So, run tests during refactoring, run tests right before you commit code, and don't let a task be considered "done" until the tests are passing too.

Finally, **maintain those tests**. As I said before, if you're not keeping those tests up-to-date when the application changes, they lose their value. Especially if they are failing, failing tests

are costing time and money to investigate each time they fail. Refactor the tests as needed, when the code changes.

As you can see, maximizing your returns on money and time invested in your unit tests requires some investment in applying best practices. But in the end, the rewards are worth the initial investment.

What About Code Coverage?

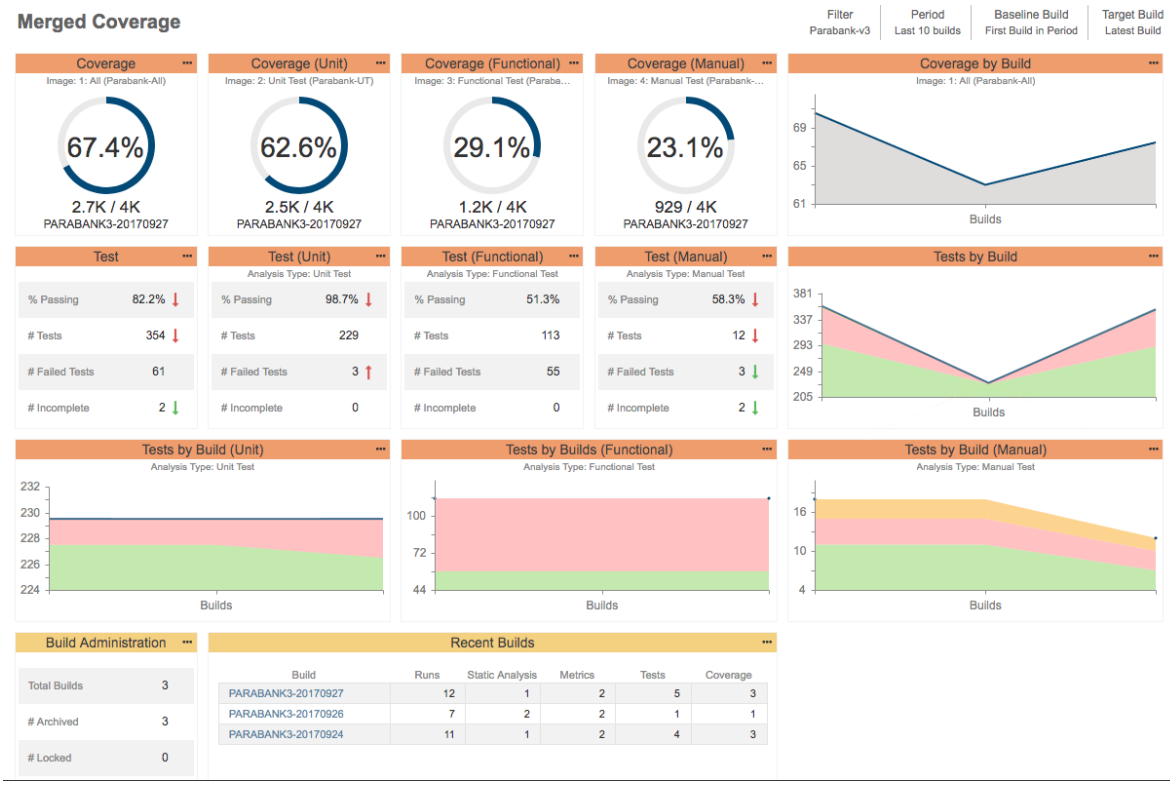
In general, code coverage is a measurement of how much of the production code is executed while your automated tests are running. By running a suite of tests and looking at code coverage data, you can get a general sense of how much of your application is being tested.

There are many kinds of code coverage — the most common ones are line coverage and branch coverage. Most tools focus on line coverage, which just tells you if a specific line was covered. Branch is more granular, as it tells you if *each path through the code* is covered.

Code coverage is an important metric, but remember that increasing it is a means to an end. It's great for finding gaps in testing, but it's not the only thing to focus on. Be careful not to spend too much effort trying to achieve 100% coverage — it may not even be possible or feasible, and really the quality of your tests is the important thing. That being said, achieving at least 60% coverage for your projects is a good starting point, and 80% or more is a good goal to set. Obviously, it's up to you to decide what that goal should be.

It's also valuable if you have automated tools that not only measure code coverage but also keep track how much modified code is being covered by tests, because this can provide visibility into whether enough tests are being written along with changes in production code.

See here an example code coverage report from Parasoft's reporting and analytics hub, that you can navigate through if you are using Parasoft Jtest for your unit testing:



Another thing to keep in mind is that, when writing new tests, be careful of focusing on line coverage alone, as single lines of code can result in multiple code paths, so make sure your tests validate these code paths. Line coverage is a useful quick indicator, but it isn't the only thing to look for.

The most obvious way to increase coverage is simply to add more tests for more code paths, and more use-cases of the method under test. A powerful way to increase coverage is to use parameterized tests. For Junit4, there was the built-in Junit4 Parameterized functionality and 3rd-party libraries like JunitParams. Junit5 has built-in parameterization.

Finally, if you aren't already tracking test coverage, I highly recommend you start. There's plenty of tools out there that can help. Start by measuring your current coverage numbers, then set goals for where it should be, address important gaps first, and then work from there.

Summary

Although unit testing is a proven technique for ensuring software quality, it's still considered a burden to developers and many teams are still struggling with it. In order to get the most out of testing and automated testing tools, tests must be trustworthy, maintainable, readable, self-contained, and be used to verify a single use case. Automation is key to making unit testing workable and scalable.

In addition, software teams need to practice good testing techniques, such as writing and reviewing tests alongside application code, maintaining tests, and ensuring that failed tests are tracked and remediated immediately. Adopting these unit testing best practices can quickly improve your unit testing outcomes.