PARASOFT®
Perfecting Software

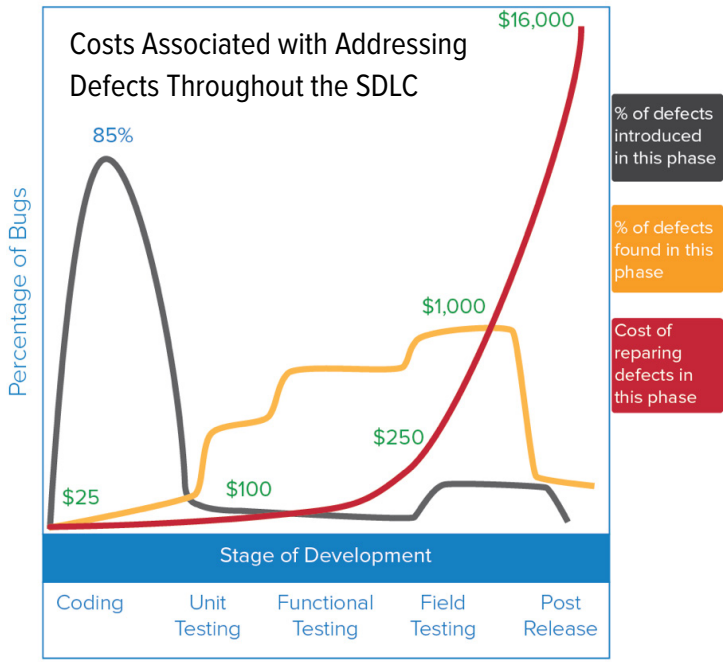
Top 5 Unit Testing Best Practices

Technical Whitepaper

Executive Summary

Software developers employ unit testing to individually scrutinize very small parts of an application ("units"), to test if they are operating properly. Unit tests are proven development testing activities for building quality into software, but only work as intended if they are written correctly, used consistently, and maintained as the code evolves.

Upfront time and effort is required to reap the benefits of unit testing, which saves overall development time and effort over the life of the application. In many organizations, though, the argument for dedicating resources to unit testing upfront is outweighed by the need to finish the project on time and on budget. This approach is unnecessarily risky, as it assumes that defects found in QA or in the field are acceptable.



Source: Applied Software Measurement, Capers/Jones, 1996

In the long run, as shown in the chart to the left, organizations will benefit from spending a little extra time during the coding stage, as opposed to a lot of extra time in the integration phase.

In this paper, we'll explore simple unit testing best practices for optimizing your development efforts. You'll learn how to:

- Create actionable tests for the present and the future.
- Ensure that unit testing is adopted across the team.
- Help the team avoid unit testing pitfalls.

Unit testing helps prevent costly field-reported bugs by finding defects as early as possible.

1. Design Tests for Precision

Committing to running unit tests is a great start, but the next step is to resist the temptation to hammer them out without planning how tests will target the code and scale as it evolves. Tests require strategy to be effective, otherwise you may see a large amount of noise, or worse, tests that fail to flag defective code.

Avoid tests that produce a failure if a unit that isn't the target of the test fails. For instance, if the function you're testing depends on a different function, you will need to write a separate test targeting the dependent function. This is deceptively simple because you also want to make sure that your test is focused on the important aspects of the function. For example, if the unit under test runs calculations, your test needs to make sure that the unit correctly calculates; don't be tempted to include variable input consistency—that's a separate test.

Remember these two simple rules when creating your unit test: First, what does it mean when the test passes? And second, what does it mean when the test fails? If you cannot answer that question then you need to rethink or redesign the test.

Writing and running a good unit test takes time and should be factored in during the planning process. Security vulnerabilities, memory errors, and poorly structured code could be lurking in your application, and it's far less expensive to find and remediate them during the development process than in QA. That goes double for defects found in the field.

2. Stop Writing Tests from the Hip

One of biggest reasons for failing to consistently run unit tests is that they become noisier as the code evolves, which is usually a sign that the tests have not been maintained. Ignoring or sporadically maintaining unit test suites is chronically justified by arguing that rewriting the tests will put the team behind schedule. This is symptomatic of a "temporary-test" mindset in which the developer subscribes to the I-just-need-to-test-this-one-thing-right-now approach. From the temporary-test perspective, the developer is more likely to write unit tests that can't be maintained.

It's good that the developer is even taking time to conduct unit tests at all, but he or she must also take the extra time to think about how to structure the test so that it can be maintained to keep up with code as it evolves.

Think of writing code like writing a technical document – if you rely on a spell checker to find spelling errors in this type of document, it will probably return a slew of false positives. But suppose you added the technical terms in your document to the spell checker library. You would successfully reduce the noise and be able to confidently finish writing your document. And if you continued to maintain your spell checker dictionary, it would be easier to find real problems when you wrote a new document with new technical terminology.

Similarly, developers must do a little planning and maintenance in order to make sure that their tests are able to update as the software evolves.

Don't Forget the Data

Data used in test assertions can also affect maintainability. Unit tests are tightly coupled to the code, which helps you control the data because the expectations are defined alongside the test. This is a double-edged sword, however, because the cost of having the data available means that you must also keep the data up to date.

3. Learn the Art of the Assertion

Writing assertions is actually very easy, which is why the number of assertions increases as the test suites grows. When you start writing test suites, it's easy to assert everything and throw it into the code. Consider where the assertions are going and ask the following questions:

- Does this matter? Is it important?
- Does it help me distinguish an actual failure?
- Will I be able to maintain it in the long term?

Writing a technically-correct assertion does not guarantee that the data will be properly processed. The assertion needs to test the function based on your intentions, otherwise the test is just going to create noise. The logic of your assertions must remain consistent with the application, so you should assert something that connects to the logic of the application.

Developers have a range of possible assertions available: assert null, assert true/false, assert contains. Breaking assertions down to a true/false condition, though, simplifies unit tests. In addition, writing assertions that check values by range, as opposed to equivalents, provides better scalability and maintainability over the long term.

Lastly, it is helpful to write tests that aid developers in understanding what failed so that they have an idea of how to address the failure. In the following example, an assertion returns a string that let's you know what went wrong:

```
public Integer getNumber() {
   Scanner input = new Scanner( System.in);
   System.out.print( "Enter a number between 0 and 10: ");
   int number = input.nextInt();
   // assert that the absolute value is >=0
   assert ( number >=0 && number <=10 ) : "bad number: "+number;
   return number
}
```

When the person examining the failure is armed with useful information, there is a greater chance that the problem will be investigated and fixed.

4. Automate Regular Test Execution

The simplest way to ensure that unit tests are executed is to integrate automated testing into your development process. This way, developers are never in a position to decide whether or not they should run a test.

An additional benefit is that the tests will prevent integration problems. If developer A's code passes local unit tests, and developer B's code passes local tests, there still may be an issue when the software builds. These issues will be missed if there isn't a policy implemented for running continuous testing to help you find and correct integration problems.

Automating your testing means complete integration:

- No humans should be required for input or review.
- Tests must be independently and consistently repeatable.
- Tests must not have dependencies.
- Tests must be able to run over and over again on supported configurations.

Preparing Your Test Suite for Automation

Removing noisy tests before automating them will decrease the chances of legitimate test failures being dismissed as false positives. Inevitably, there is a certain percentage of noise that most developers are comfortable accepting, but it is still important to strive for quiet test suites.

If your noisy test suites cause you to scan through lines of code to identify legitimate defects, you should consider the following process for getting rid of noise:

1. Run your test suite (make they have assertions).
2. Turn off, but don't delete, any test that fails. If time permits, scan through the disabled tests to identify any you might want to go back and fix.
3. Run the test suite again.
4. Repeat step 2.
5. Run the tests on a different system, including a different OS or non-standard directory if possible.
6. Repeat step 2.
7. Run the tests on a different date (the actual next day or change the clock on your computer).
8. Repeat step 2; if your application has time dependencies, then you may want to keep them.
9. Run coverage analysis and write new (better) tests to fill in any gaps.

Tests that you turned off may be useful for guidance as you fill in the gaps, but it's more often better to start fresh because you will (hopefully) become a better programmer over time and will likely be able to create better tests. You should also consider running coverage analysis before and after cleaning out your suite to see how removing tests has affected your coverage. You may find that you need to fix and replace some of the tests, or you may find that turning off all those tests had no effect on the coverage results.

Letting Go

It can be extremely difficult for a developer to turn away from tests he or she wrote—especially if they still work. But in many cases, it's the right thing to do. Brilliant authors learn to delete favorite scenes because they don't add to the plot; captivating movies are made more entertaining by removing unnecessary exchanges; developers must learn to do the same. (If you are having a hard time letting go, use the two rules identified in Best Practice 1.)

5. Adopt a Unit Testing Policy

These testing best practices are technical in nature, but all of them are underlined by the need for human diligence. Developers are faced with making coding and testing decisions every day that go beyond the software, also affecting the business. These decisions determine the safety, security, performance, and reliability of the software that drives the business, giving developers power to introduce or minimize business risks.

The key to reining in these risks is to align software development activities with your business goals. This can be achieved through policy-driven development, which ensures that developers deliver software that matches expectations set by the business.

Policy-driven development must include:

1. Clearly defining expectations and documenting them in understandable policies.
2. Training engineers on the business objectives driving those policies.
3. Monitoring policy adherence in an automated, unobtrusive way.
4. Assoiating non-functional requirements with objective metrics, including areas like performance, security, and reliability.

We recommend implementing a policy that states, for example, that each function, method, or class, has at least one unit test associated with it. You can even determine a policy for how to create test suites. The following examples of unit testing policies may be a good place to start:

- All functions must have a unit with an assertion.
- Assertions must include a pass/fail condition.
- Test suites must run every time a change set is committed.
- Always create a test for defects found in the field.
- All tests must be verified by at least one peer.

Although peer reviewing tests hasn't yet become a regular practice, the process provides tremendous value, helping make sure that tests aren't noisy and are testing what they are supposed to test.

Policy vs. Guidelines

The main reason organizations fail to implement software development practices that they know will have a positive impact is that they rely on guidelines to govern software development rather than policies.

Guidelines describe suggested behavior, whereas policies define expected behavior. For example, "wash your hands after using the restroom" or "look both ways before crossing the street" are guidelines. They're great suggestions, but they cannot be fully enforced. For more information about defining expected behavior, read our whitepaper about implementing policy-driven development.

Conclusion

Unit testing is an invaluable tool for ensuring the quality, security, safety and reliability of your software. If your organization is adopting (or has already adopted) a development testing strategy that includes unit tests, you're part of the way there. But in order to reap the full benefits of unit testing, developers should:

- Take time to plan and peer review their tests
- Write maintainable tests
- Use assertions to control data
- Automate regular, continuous test execution
- Elevate unit testing from recommendation to development policy

The organization, furthermore, must recognize that following unit testing best practices requires an upfront investment in terms of development resources. This investment is necessary to ensure that developers are comfortable with the unit testing policy so it can be properly executed.

About Parasoft

Parasoft helps organizations perfect today's highly-connected applications by automating time-consuming testing tasks and providing management with intelligent analytics necessary to focus on what matters. Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software, by integrating static and runtime analysis; unit, functional, and API testing; and service virtualization. With developer testing tools, manager reporting/analytics, and executive dashboarding, Parasoft supports software organizations with the innovative tools they need to successfully develop and deploy applications in the embedded, enterprise, and IoT markets, all while enabling today's most strategic development initiatives — agile, continuous testing, DevOps, and security.

Parasoft Corporation
101 E Huntington Drive
Monrovia, CA 91016 USA
Sales: 1-888-305-0041
Int'l: +1-626-256-3680