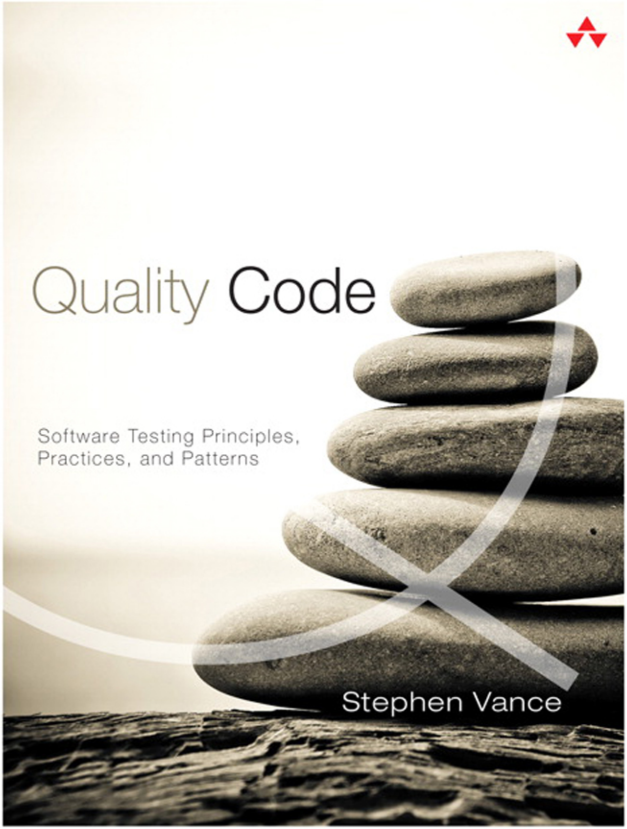

Quality Code

Software Testing Principles, Practices, and Patterns

Stephen Vance

Quality Code

are less likely to incur coupling. Small tests are easier to diagnose when they fail. Need I say more?

Separate Your Concerns

Many of us think of separation of concerns when we design and code software but forget about it when we test. A common source of complexity in tests is a failure to separate the concerns.

For example, say you are creating a date-conversion library, two of the methods of which parse date strings into the underlying representation and format it into a display string. You could write a test that verifies the result by a round-trip, as in Listing 5-4.

Listing 5-4: *Testing date parsing and formatting through round-trip conversion in JavaScript with Jasmine*

```
describe('DateLib', function() {
  it('converts the date back into itself', function() {
    var expectedDate = '17 Apr 2008 10:00 +1000';

    var actualDate = DateLib.format(
        DateLib.parse(expectedDate));

    expect(actualDate).toBe(expectedDate);
  });
});
```

However, you are actually testing two independent pieces of functionality: the parsing and the formatting. Just because you can test them together does not mean you should. Instead, test them separately as in Listing 5-5.

Listing 5-5: *Testing date parsing and formatting from Listing 5-4 as separate concerns*

```
describe('DateLib', function() {
  it('parses a date properly', function() {
    var expectedTimestamp = 1208390400000;

    var actualDate = DateLib.parse('17 Apr 2008 10:00 +1000');
    expect(actualDate.getTime()).toBe(expectedTimestamp);
  });

  it('formats a timestamp properly for GMT', function() {
    var expectedDateString = 'Thu, 17 Apr 2008 00:00:00 GMT';
    var inputDate = new DateLib.Date(1208390400000);

    expect(inputDate.toGMTString()).toBe(expectedDateString);
  });
});
```

By testing your functionality separately, you verify each operation independently and catch the case in which one bug compensates for another. You also more directly express the way in which each piece of functionality should be used.

Use Unique Values

The readability and usefulness of tests are enhanced when the values used to construct fixtures are mutually distinct. Within a test, unique values reinforce the differences between various assertions and help highlight when the implementation has miswired parameters or properties. Between iterations of a data-driven test, distinct values help to highlight the failing test case more clearly. The same effect applies between tests, in which case unique values also help to identify the effects of shared state.

Assertion failures generally provide good feedback on the values involved and the location in the code. However, some assertions, like `assertTrue`, do not display their contributing expression values. Assertions in loops will share a location for multiple data values. Custom assertions and assertions in fixtures or utility methods will show deeper stack traces than you would normally expect. In each of these cases, unique values help to localize the failure more quickly.

Data-driven tests magnify the shared location property of assertions in loops. The entire test is executed repeatedly rather than just the few lines in the loop. Unique values within a row help to distinguish the deviant property. Unique values within a column isolate the test cases regardless of which assertion fails.

Although best practice guides us away from tests that interact with each other, various shared resources ranging from static variables to singletons to files or databases can cause crossover, especially when tests are run in parallel. Unique values allow us to more easily notice when a value shows up in the wrong test or context.

Keep It Simple: Remove Code

You can simplify your tests and reduce your testing burden by deleting code. If the code does not exist, you neither have to test it nor have to maintain it. Remove unused code, commented code, code that does more than it needs to, and code that is more complex than it needs to be. Deleting code is a great way to reduce your costs.

Also, remove redundant tests. Even better, do not write them to begin with. Extra tests do not add to your validation, but they take time to run and attention to maintain. Just imagine how much time could be wasted trying to figure out what makes the redundant test different from the other tests in the same area.

If you use code coverage to guide your testing, removing code also increases your coverage. We usually think of increasing coverage by increasing the number of tests. But coverage is a percentage computed by dividing the number of features[4] exercised by the number of features that exist. We have two degrees of freedom to control the coverage, including reducing the total number of features. In the case of redundant tests, your coverage will not increase but it will not decrease, either.

Don't Test the Framework

You are more likely to succeed by leveraging the work of others than by duplicating their efforts. While you may find it fun to write your own <fill in your own pet project>, chances are good that someone else with greater expertise or experience has already written it, and hundreds or thousands of people are using it. Or it may be available as part of the standard features or libraries of your language of choice.

Frameworks, libraries, plug-ins, code generators, and their kin increasingly let us work at a higher and higher level of abstraction. Whether built in, commercial, or open source, we incorporate them into our development ecosystem and trust them to work correctly.

Commercial software comes with support and maintenance as well as—despite the license disclaimers—some expectation of fitness for

4. I refer to "features" here because coverage is not just about lines or statements. A thorough testing regimen may also consider branches, conditions, loop iterations, def-use chains, or other measurable features of the code.

purpose. The same is usually true of our system libraries, often including the backing design and support of a standards body in their design and specification. Open-source software comes with user communities and often includes test suites as part of their build and installation.

Generally, you do not need to test third-party code. I have lost count of the number of times I have seen developers, myself included, quickly jump to the conclusion that the compiler or library had a bug because it could not have been our mistake, only to realize after a long side trip that we should look at our own code first. Sure, sometimes shortcomings in the availability or quality of documentation contribute. Once in awhile, the bug really is in the third-party code. However, most of the problems are ours.

Someone else has done the work to guarantee the quality of the software. Let's not duplicate their efforts.

Don't Test Generated Code

Note that generated code also falls into this category. The benefits of generating source code have proven themselves for decades. Early compilers were assembly code generators. Many compilers are generated from descriptions of their lexemes and grammars. XML parsers, regular expression generators, and CORBA, SOAP, and REST interfaces all use code generation in many implementations. Trends toward domain-specific languages (DSLs) and higher-level constructs frequently employ generators, at least in their early incarnations. So why wouldn't we test that code?

First of all, code generation usually derives from a solid theoretical or experiential foundation. The generators implement empirically or algorithmically correct approaches to solve the problem. While this does not guarantee a correct implementation, it provides a solid basis for more proven solutions.

Second, generated code often solves problems for which the authors or the industry have not come up with generalized implementations. This means that the generated code is usually repetitious and is thus tedious to test.

Finally, I have found that much generated code ignores many good design principles, making it difficult to test. Many of the theory-based generators come from procedural heritages and do not share the same level of obsession with small functions that we see in current programming paradigms. And the code is simply not designed for human maintenance.

Test that you have used the generator to correctly achieve the purpose for your system. If you are using a compiler generator, make sure the generated compiler accepts the language you are trying to implement. For a remote API, test the business purpose of the API via the generated stubs.

Sometimes Test the Framework

To every rule, there are exceptions. When we talked about frameworks, we talked about things like trust, having tests, guarantees, and correctness. Sometimes, those conditions are not true. Other times, you cannot afford to assume they are true. Let's look at some times in which you need to test the framework.

Imagine you are part of a team working on a large software system, perhaps a product consisting of millions of lines of code, hundreds of developers, and using hundreds of third-party packages across the entire code base. In any given period of time between releases, perhaps dozens of those packages change and may need to be upgraded. How do you update the packages safely without breaking your system? You can rely on the tests you have written for your own use of the packages in part, but in other cases, particularly those in which you have chosen to stub or mock out the framework functionality (e.g., a service access library) or where it participates outside of the scope of your tests (like a dependency injection framework might, for example), your tests may not be sufficient to qualify the new version.

An open-source package without tests, especially one that is young, infrequently maintained, or with a history of incompatibility, also may not meet the standards of trust you need to include it without verification. Writing your own tests may be the most prudent course of action to ensure it meets your needs. You may even consider contributing your tests back to the project.

Frameworks with high-risk dependencies for your project probably also deserve greater attention to correctness. A simple framework used pervasively affects all parts of your system. A framework used comprehensively requires all parts of the framework to behave as expected.

In the end, testing third-party code entails a risk assessment of whether the cost of testing it justifies the benefits of reuse against the risk of failures. Consider it, and judge wisely.

Part II

Testing and Testability Patterns

With philosophy and principles in place, now it is time to dive into code, the essence of our craft. Part II starts with setting the basic foundation of consistency upon which to build your tests. It continues through increasingly sophisticated topics using examples from a variety of programming languages. It finishes with a discussion of how to test concurrency.