4 The four pillars of a good unit test

This chapter covers

- Exploring dichotomies between aspects of a good unit test
- Defining an ideal test
- Understanding the Test Pyramid
- Using black-box and white-box testing

Now we are getting to the heart of the matter. In chapter 1, you saw the properties of a good unit test suite:

- *It is integrated into the development cycle.* You only get value from tests that you actively use; there's no point in writing them otherwise.
- *It targets only the most important parts of your code base.* Not all production code deserves equal attention. It's important to differentiate the heart of the application (its *domain model*) from everything else. This topic is tackled in chapter 7.
- *It provides maximum value with minimum maintenance costs.* To achieve this last attribute, you need to be able to
  - Recognize a valuable test (and, by extension, a test of low value)
  - Write a valuable test

As we discussed in chapter 1, *recognizing a valuable test* and *writing a valuable test* are two separate skills. The latter skill requires the former one, though; so, in this chapter, I'll show how to recognize a valuable test. You'll see a universal frame of reference with which you can analyze any test in the suite. We'll then use this frame of reference to go over some popular unit testing concepts: the Test Pyramid and black-box versus white-box testing.

Buckle up: we are starting out.

4.1 Diving into the four pillars of a good unit test

A good unit test has the following four attributes:

- Protection against regressions
- Resistance to refactoring
- Fast feedback
- Maintainability

These four attributes are foundational. You can use them to analyze any automated test, be it unit, integration, or end-to-end. Every such test exhibits some degree of each attribute. In this section, I define the first two attributes; and in section 4.2, I describe the intrinsic connection between them.

4.1.1 The first pillar: Protection against regressions

Let's start with the first attribute of a good unit test: *protection against regressions*. As you know from chapter 1, a *regression* is a software bug. It's when a feature stops working as intended after some code modification, usually after you roll out new functionality.

Such regressions are annoying (to say the least), but that's not the worst part about them. The worst part is that the more features you develop, the more chances there are that you'll break one of those features with a new release. An unfortunate fact of programming life is that *code is not an asset, it's a liability*. The larger the code base, the more exposure it has to potential bugs. That's why it's crucial to develop a good protection against regressions. Without such protection, you won't be able to sustain the project growth in a long run—you'll be buried under an ever-increasing number of bugs.

To evaluate how well a test scores on the metric of protecting against regressions, you need to take into account the following:

- The amount of code that is executed during the test
- The complexity of that code
- The code's domain significance

Generally, the larger the amount of code that gets executed, the higher the chance that the test will reveal a regression. Of course, assuming that this test has a relevant set of assertions, you don't want to merely execute the code. While it helps to know that this code runs without throwing exceptions, you also need to validate the outcome it produces.

Note that it's not only the amount of code that matters, but also its complexity and domain significance. Code that represents complex business logic is more important than boilerplate code—bugs in business-critical functionality are the most damaging.

On the other hand, it's rarely worthwhile to test trivial code. Such code is short and doesn't contain a substantial amount of business logic. Tests that cover trivial code don't have much of a chance of finding a regression error, because there's not a lot of room for a mistake. An example of trivial code is a single-line property like this:

```
public class User
{
    public string Name { get; set; }
}
```

Furthermore, in addition to your code, the code you didn't write also counts: for example, libraries, frameworks, and any external systems used in the project. That code influences the working of your software almost as much as your own code. For the best protection, the test must include those libraries, frameworks, and external systems in the testing scope, in order to check that the assumptions your software makes about these dependencies are correct.

TIP To maximize the metric of protection against regressions, the test needs to aim at exercising as much code as possible.

4.1.2 The second pillar: Resistance to refactoring

The second attribute of a good unit test is *resistance to refactoring*—the degree to which a test can sustain a refactoring of the underlying application code without turning red (failing).

DEFINITION *Refactoring* means changing existing code without modifying its observable behavior. The intention is usually to improve the code's nonfunctional characteristics: increase readability and reduce complexity. Some examples of refactoring are renaming a method and extracting a piece of code into a new class.

Picture this situation. You developed a new feature, and everything works great. The feature itself is doing its job, and all the tests are passing. Now you decide to clean up the code. You do some refactoring here, a little bit of modification there, and everything looks even better than before. Except one thing—the tests are failing. You look more closely to see exactly what you broke with the refactoring, but it turns out that you didn't break anything. The feature works perfectly, just as before. The problem is that the tests are written in such a way that they turn red with any modification of the underlying code. And they do that regardless of whether you actually break the functionality itself.

This situation is called a *false positive*. A false positive is a false alarm. It's a result indicating that the test fails, although in reality, the functionality it covers works as intended. Such false positives usually take place when you refactor the code—when you modify the implementation but keep the observable behavior intact. Hence the name for this attribute of a good unit test: *resistance to refactoring*.

To evaluate how well a test scores on the metric of resisting to refactoring, you need to look at how many false positives the test generates. The fewer, the better.

Why so much attention on false positives? Because they can have a devastating effect on your entire test suite. As you may recall from chapter 1, the goal of unit testing is to enable sustainable project growth. The mechanism by which the tests enable sustainable growth is that they allow you to add new features and conduct regular refactorings without introducing regressions. There are two specific benefits here:

- *Tests provide an early warning when you break existing functionality.* Thanks to such early warnings, you can fix an issue long before the faulty code is deployed to production, where dealing with it would require a significantly larger amount of effort.
- *You become confident that your code changes won't lead to regressions.* Without such confidence, you will be much more hesitant to refactor and much more likely to leave the code base to deteriorate.

False positives interfere with both of these benefits:

- If tests fail with no good reason, they dilute your ability and willingness to react to problems in code. Over time, you get accustomed to such failures and stop paying as much attention. After a while, you start ignoring legitimate failures, too, allowing them to slip into production.
- On the other hand, when false positives are frequent, you slowly lose trust in the test suite. You no longer perceive it as a reliable safety net—the perception is diminished by false alarms. This lack of trust leads to fewer refactorings, because you try to reduce code changes to a minimum in order to avoid regressions.

A story from the trenches

I once worked on a project with a rich history. The project wasn't too old, maybe two or three years; but during that period of time, management significantly shifted the direction they wanted to go with the project, and development changed direction accordingly. During this change, a problem emerged: the code base accumulated large chunks of leftover code that no one dared to delete or refactor. The company no longer needed the features that code provided, but some parts of it were used in new functionality, so it was impossible to get rid of the old code completely.

The project had good test coverage. But every time someone tried to refactor the old features and separate the bits that were still in use from everything else, the tests failed. And not just the old tests—they had been disabled long ago—but the new tests, too. Some of the failures were legitimate, but most were not—they were false positives.

At first, the developers tried to deal with the test failures. However, since the vast majority of them were false alarms, the situation got to the point where the developers ignored such failures and disabled the failing tests. The prevailing attitude was, "If it's because of that old chunk of code, just disable the test; we'll look at it later."

Everything worked fine for a while—until a major bug slipped into production. One of the tests correctly identified the bug, but no one listened; the test was disabled along with all the others. After that accident, the developers stopped touching the old code entirely.

This story is typical of most projects with brittle tests. First, developers take test failures at face value and deal with them accordingly. After a while, people get tired of tests crying "wolf" all the time and start to ignore them more and more. Eventually, there comes a moment when a bunch of real bugs are released to production because developers ignored the failures along with all the false positives.

You don't want to react to such a situation by ceasing all refactorings, though. The correct response is to re-evaluate the test suite and start reducing its brittleness. I cover this topic in chapter 7.

4.1.3 What causes false positives?

So, what causes false positives? And how can you avoid them?

The number of false positives a test produces is directly related to the way the test is structured. The more the test is coupled to the implementation details of the system under test (SUT), the more false alarms it generates. The only way to reduce the chance of getting a false positive is to decouple the test from those implementation details. You need to make sure the test verifies the end result the SUT delivers: its observable behavior, not the steps it takes to do that. Tests should approach SUT verification from the end user's point of view and check only the outcome meaningful to that end user. Everything else must be disregarded (more on this topic in chapter 5).

The best way to structure a test is to make it tell a story about the problem domain. Should such a test fail, that failure would mean there's a disconnect between the story and the actual application behavior. It's the only type of test failure that benefits you—such failures are always on point and help you quickly understand what went wrong. All other failures are just noise that steer your attention away from things that matter.

Take a look at the following example. In it, the `MessageRenderer` class generates an HTML representation of a message containing a header, a body, and a footer.

Listing 4.1 Generating an HTML representation of a message

```
public class Message
{
    public string Header { get; set; }
    public string Body { get; set; }
    public string Footer { get; set; }
}
```

```
public interface IRenderer
{
    string Render(Message message);
}

public class MessageRenderer : IRenderer
{
    public IReadOnlyList<IRenderer> SubRenderers { get; }

    public MessageRenderer()
    {
        SubRenderers = new List<IRenderer>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
            new FooterRenderer()
        };
    }

    public string Render(Message message)
    {
        return SubRenderers
            .Select(x => x.Render(message))
            .Aggregate("", (str1, str2) => str1 + str2);
    }
}
```

The `MessageRenderer` class contains several sub-renderers to which it delegates the actual work on parts of the message. It then combines the result into an HTML document. The sub-renderers orchestrate the raw text with HTML tags. For example:

```
public class BodyRenderer : IRenderer
{
    public string Render(Message message)
    {
        return $"<b>{message.Body}</b>";
    }
}
```

How can `MessageRenderer` be tested? One possible approach is to analyze the algorithm this class follows.

Listing 4.2 Verifying that `MessageRenderer` has the correct structure

```
[Fact]
public void MessageRenderer_uses_correct_sub_renderers()
{
    var sut = new MessageRenderer();

    IReadOnlyList<IRenderer> renderers = sut.SubRenderers;
    Assert.Equal(3, renderers.Count);
    Assert.IsAssignableFrom<HeaderRenderer>(renderers[0]);
    Assert.IsAssignableFrom<BodyRenderer>(renderers[1]);
    Assert.IsAssignableFrom<FooterRenderer>(renderers[2]);
}
```

This test checks to see if the sub-renderers are all of the expected types and appear in the correct order, which presumes that the way `MessageRenderer` processes messages must also be correct. The test might look good at first, but does it really verify `MessageRenderer`'s observable behavior? What if you rearrange the sub-renderers, or replace one of them with a new one? Will that lead to a bug?

Not necessarily. You could change a sub-renderer's composition in such a way that the resulting HTML document remains the same. For example, you could replace `BodyRenderer` with a `BoldRenderer`, which does the same job as `BodyRenderer`. Or you could get rid of all the sub-renderers and implement the rendering directly in `MessageRenderer`.

Still, the test will turn red if you do any of that, even though the end result won't change. That's because the test couples to the SUT's implementation details and not the outcome the SUT produces. This test inspects the algorithm and expects to see one particular implementation, without any consideration for equally applicable alternative implementations (see figure 4.1).

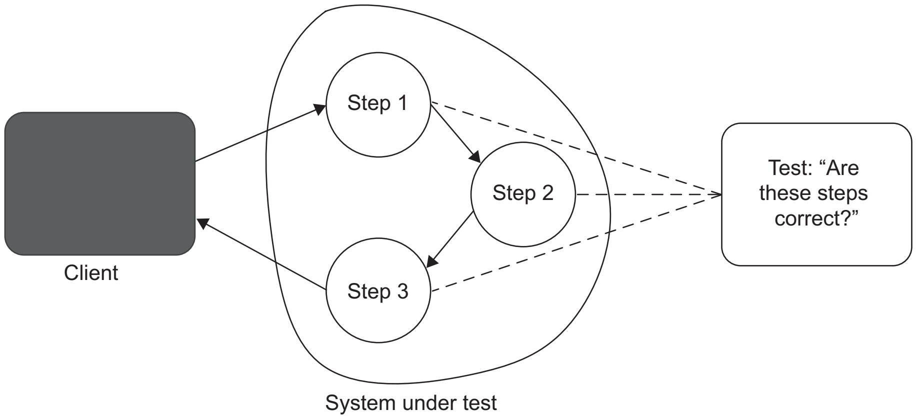

Figure 4.1 A test that couples to the SUT's algorithm. Such a test expects to see one particular implementation (the specific steps the SUT must take to deliver the result) and therefore is brittle. Any refactoring of the SUT's implementation would lead to a test failure.

Any substantial refactoring of the `MessageRenderer` class would lead to a test failure. Mind you, the process of *refactoring* is changing the implementation without affecting the application's observable behavior. And it's precisely because the test is concerned with the implementation details that it turns red every time you change those details.

Therefore, *tests that couple to the SUT's implementation details are not resistant to refactoring.* Such tests exhibit all the shortcomings I described previously:

- They don't provide an early warning in the event of regressions—you simply ignore those warnings due to little relevance.
- They hinder your ability and willingness to refactor. It's no wonder—who would like to refactor, knowing that the tests can't tell which way is up when it comes to finding bugs?

The next listing shows the most egregious example of brittleness in tests that I've ever encountered, in which the test reads the source code of the `MessageRenderer` class and compares it to the "correct" implementation.

Listing 4.3 Verifying the source code of the `MessageRenderer` class

```
[Fact]
public void MessageRenderer_is_implemented_correctly()
{
    string sourceCode = File.ReadAllText(@"[path]\MessageRenderer.cs");

    Assert.Equal(@"
public class MessageRenderer : IRenderer
{
    public IReadOnlyList<<IRenderer> SubRenderers { get; }

    public MessageRenderer()
    {
        SubRenderers = new List<<IRenderer>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
            new FooterRenderer()
        };
    }

    public string Render(Message message) { /* ... */ }
}", sourceCode);
}
```

Of course, this test is just plain ridiculous; it will fail should you modify even the slightest detail in the `MessageRenderer` class. At the same time, it's not that different from the test I brought up earlier. Both insist on a particular implementation without taking into consideration the SUT's observable behavior. And both will turn red each time you change that implementation. Admittedly, though, the test in listing 4.3 will break more often than the one in listing 4.2.

4.1.4 Aim at the end result instead of implementation details

As I mentioned earlier, the only way to avoid brittleness in tests and increase their resistance to refactoring is to decouple them from the SUT's implementation details—keep as much distance as possible between the test and the code's inner workings, and instead aim at verifying the end result. Let's do that: let's refactor the test from listing 4.2 into something much less brittle.

To start off, you need to ask yourself the following question: What is the final outcome you get from `MessageRenderer`? Well, it's the HTML representation of a message. And it's the only thing that makes sense to check, since it's the only observable result you get out of the class. As long as this HTML representation stays the same, there's no need to worry about exactly how it's generated. Such implementation details are irrelevant. The following code is the new version of the test.

Listing 4.4 Verifying the outcome that `MessageRenderer` produces

```
[Fact]
public void Rendering_a_message()
{
    var sut = new MessageRenderer();
    var message = new Message
    {
        Header = "h",
        Body = "b",
        Footer = "f"
    };

    string html = sut.Render(message);

    Assert.Equal("<h1>h</h1><b>b</b><i>f</i>", html);
}
```

This test treats `MessageRenderer` as a black box and is only interested in its observable behavior. As a result, the test is much more resistant to refactoring—it doesn't care what changes you make to the SUT as long as the HTML output remains the same (figure 4.2).

Notice the profound improvement in this test over the original version. It aligns itself with the business needs by verifying the only outcome meaningful to end users—

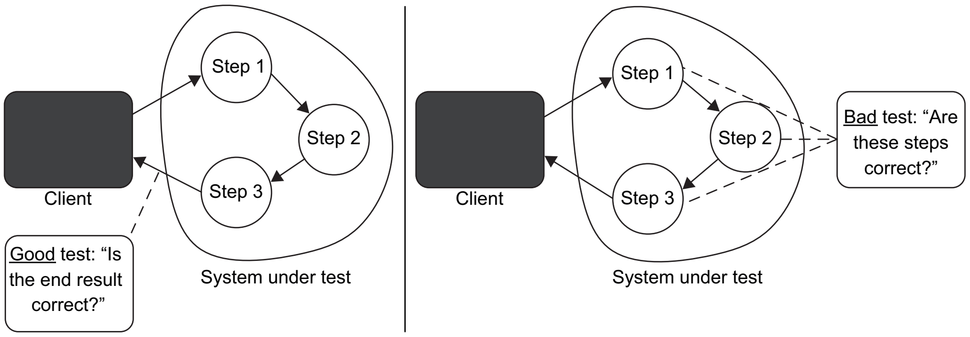

Figure 4.2 The test on the left couples to the SUT's observable behavior as opposed to implementation details. Such a test is resistant to refactoring—it will trigger few, if any, false positives.

how a message is displayed in the browser. Failures of such a test are always on point: they communicate a change in the application behavior that can affect the customer and thus should be brought to the developer's attention. This test will produce few, if any, false positives.

Why *few* and not *none at all*? Because there could still be changes in `MessageRenderer` that would break the test. For example, you could introduce a new parameter in the `Render()` method, causing a compilation error. And technically, such an error counts as a false positive, too. After all, the test isn't failing because of a change in the application's behavior.

But this kind of false positive is easy to fix. Just follow the compiler and add a new parameter to all tests that invoke the `Render()` method. The worse false positives are those that don't lead to compilation errors. Such false positives are the hardest to deal with—they seem as though they point to a legitimate bug and require much more time to investigate.

4.2 The intrinsic connection between the first two attributes

As I mentioned earlier, there's an intrinsic connection between the first two pillars of a good unit test—*protection against regressions* and *resistance to refactoring*. They both contribute to the accuracy of the test suite, though from opposite perspectives. These two attributes also tend to influence the project differently over time: while it's important to have good protection against regressions very soon after the project's initiation, the need for resistance to refactoring is not immediate.

In this section, I talk about

- Maximizing test accuracy
- The importance of false positives and false negatives

4.2.1 Maximizing test accuracy

Let's step back for a second and look at the broader picture with regard to test results. When it comes to code correctness and test results, there are four possible outcomes, as shown in figure 4.3. The test can either pass or fail (the rows of the table). And the functionality itself can be either correct or broken (the table's columns).

The situation when the test passes and the underlying functionality works as intended is a correct inference: the test correctly inferred the state of the system (there are no bugs in it). Another term for this combination of working functionality and a passing test is *true negative*.

Similarly, when the functionality is broken and the test fails, it's also a correct inference. That's because you expect to see the test fail when the functionality is not working properly. That's the whole point of unit testing. The corresponding term for this situation is *true positive*.

But when the test doesn't catch an error, that's a problem. This is the upper-right quadrant, a *false negative*. And this is what the first attribute of a good test—protection

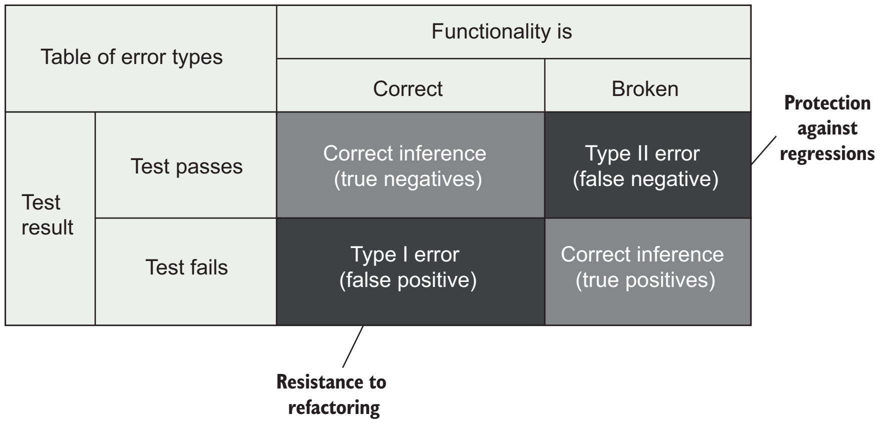

Table of error types		Functionality is	
		Correct	Broken
Test result	Test passes	Correct inference (true negatives)	Type II error (false negative)
	Test fails	Type I error (false positive)	Correct inference (true positives)

Protection against regressions
Resistance to refactoring

Figure 4.3 The relationship between protection against regressions and resistance to refactoring. Protection against regressions guards against false negatives (type II errors). Resistance to refactoring minimizes the number of false positives (type I errors).

against regressions—helps you avoid. Tests with a good protection against regressions help you to minimize the number of *false negatives*—type II errors.

On the other hand, there's a symmetric situation when the functionality is correct but the test still shows a failure. This is a *false positive*, a false alarm. And this is what the second attribute—resistance to refactoring—helps you with.

All these terms (*false positive*, *type I error* and so on) have roots in statistics, but can also be applied to analyzing a test suite. The best way to wrap your head around them is to think of a flu test. A flu test is positive when the person taking the test has the flu. The term *positive* is a bit confusing because there's nothing positive about having the flu. But the test doesn't evaluate the situation as a whole. In the context of testing, *positive* means that some set of conditions is now true. Those are the conditions the creators of the test have set it to react to. In this particular example, it's the presence of the flu. Conversely, the lack of flu renders the flu test *negative*.

Now, when you evaluate how accurate the flu test is, you bring up terms such as *false positive* or *false negative*. The probability of false positives and false negatives tells you how good the flu test is: the lower that probability, the more accurate the test.

This accuracy is what the first two pillars of a good unit test are all about. *Protection against regressions* and *resistance to refactoring* aim at maximizing the accuracy of the test suite. The accuracy metric itself consists of two components:

- How good the test is at indicating the presence of bugs (lack of false negatives, the sphere of *protection against regressions*)
- How good the test is at indicating the absence of bugs (lack of false positives, the sphere of *resistance to refactoring*)

Another way to think of false positives and false negatives is in terms of signal-to-noise ratio. As you can see from the formula in figure 4.4, there are two ways to improve test

$$\text{Test accuracy} = \frac{\text{Signal (number of bugs found)}}{\text{Noise (number of false alarms raised)}}$$

Figure 4.4 A test is accurate insofar as it generates a strong signal (is capable of finding bugs) with as little noise (false alarms) as possible.

accuracy. The first is to increase the numerator, *signal*: that is, make the test better at finding regressions. The second is to reduce the denominator, *noise*: make the test better at not raising false alarms.

Both are critically important. There's no use for a test that isn't capable of finding any bugs, even if it doesn't raise false alarms. Similarly, the test's accuracy goes to zero when it generates a lot of noise, even if it's capable of finding all the bugs in code. These findings are simply lost in the sea of irrelevant information.

4.2.2 The importance of false positives and false negatives: The dynamics

In the short term, false positives are not as bad as false negatives. In the beginning of a project, receiving a wrong warning is not that big a deal as opposed to not being warned at all and running the risk of a bug slipping into production. But as the project grows, false positives start to have an increasingly large effect on the test suite (figure 4.5).

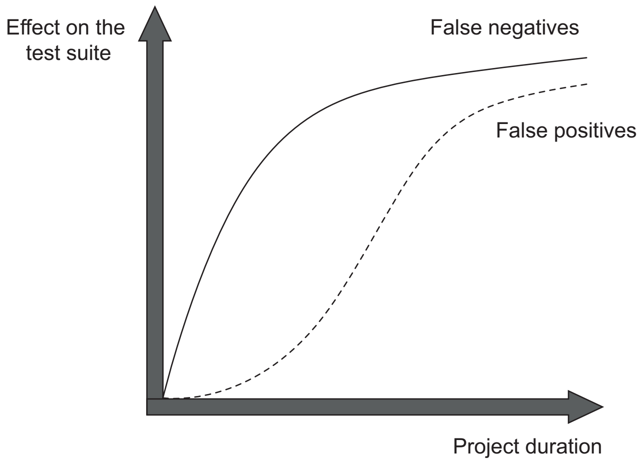

Figure 4.5 False positives (false alarms) don't have as much of a negative effect in the beginning. But they become increasingly important as the project grows—as important as false negatives (unnoticed bugs).

Why are false positives not as important initially? Because the importance of refactoring is also not immediate; it increases gradually over time. You don't need to conduct many code clean-ups in the beginning of the project. Newly written code is often shiny and flawless. It's also still fresh in your memory, so you can easily refactor it even if tests raise false alarms.

But as time goes on, the code base deteriorates. It becomes increasingly complex and disorganized. Thus you have to start conducting regular refactorings in order to mitigate this tendency. Otherwise, the cost of introducing new features eventually becomes prohibitive.

As the need for refactoring increases, the importance of *resistance to refactoring* in tests increases with it. As I explained earlier, you can't refactor when the tests keep crying "wolf" and you keep getting warnings about bugs that don't exist. You quickly lose trust in such tests and stop viewing them as a reliable source of feedback.

Despite the importance of protecting your code against false positives, especially in the later project stages, few developers perceive false positives this way. Most people tend to focus solely on improving the first attribute of a good unit test—protection against regressions, which is not enough to build a valuable, highly accurate test suite that helps sustain project growth.

The reason, of course, is that far fewer projects get to those later stages, mostly because they are small and the development finishes before the project becomes too big. Thus developers face the problem of unnoticed bugs more often than false alarms that swarm the project and hinder all refactoring undertakings. And so, people optimize accordingly. Nevertheless, if you work on a medium to large project, you have to pay equal attention to both false negatives (unnoticed bugs) and false positives (false alarms).

4.3 The third and fourth pillars: Fast feedback and maintainability

In this section, I talk about the two remaining pillars of a good unit test:

- Fast feedback
- Maintainability

As you may remember from chapter 2, fast feedback is an essential property of a unit test. The faster the tests, the more of them you can have in the suite and the more often you can run them.

With tests that run quickly, you can drastically shorten the feedback loop, to the point where the tests begin to warn you about bugs as soon as you break the code, thus reducing the cost of fixing those bugs almost to zero. On the other hand, slow tests delay the feedback and potentially prolong the period during which the bugs remain unnoticed, thus increasing the cost of fixing them. That's because slow tests discourage you from running them often, and therefore lead to wasting more time moving in a wrong direction.

Finally, the fourth pillar of good units tests, the maintainability metric, evaluates maintenance costs. This metric consists of two major components:

- *How hard it is to understand the test*—This component is related to the size of the test. The fewer lines of code in the test, the more readable the test is. It's also easier to change a small test when needed. Of course, that's assuming you don't try to compress the test code artificially just to reduce the line count. The quality of the test code matters as much as the production code. Don't cut corners when writing tests; treat the test code as a first-class citizen.
- *How hard it is to run the test*—If the test works with out-of-process dependencies, you have to spend time keeping those dependencies operational: reboot the database server, resolve network connectivity issues, and so on.

4.4 In search of an ideal test

Here are the four attributes of a good unit test once again:

- Protection against regressions
- Resistance to refactoring
- Fast feedback
- Maintainability

These four attributes, when multiplied together, determine the value of a test. And by *multiplied*, I mean in a mathematical sense; that is, if a test gets zero in one of the attributes, its value turns to zero as well:

```
Value estimate = [0..1] * [0..1] * [0..1] * [0..1]
```

TIP In order to be valuable, the test needs to score at least something in all four categories.

Of course, it's impossible to measure these attributes precisely. There's no code analysis tool you can plug a test into and get the exact numbers. But you can still evaluate the test pretty accurately to see where a test stands with regard to the four attributes. This evaluation, in turn, gives you the test's value estimate, which you can use to decide whether to keep the test in the suite.

Remember, all code, including test code, is a liability. Set a fairly high threshold for the minimum required value, and only allow tests in the suite if they meet this threshold. A small number of highly valuable tests will do a much better job sustaining project growth than a large number of mediocre tests.

I'll show some examples shortly. For now, let's examine whether it's possible to create an ideal test.

4.4.1 Is it possible to create an ideal test?

An ideal test is a test that scores the maximum in all four attributes. If you take the minimum and maximum values as `0` and `1` for each of the attributes, an ideal test must get `1` in all of them.

Unfortunately, it's impossible to create such an ideal test. The reason is that the first three attributes—*protection against regressions*, *resistance to refactoring*, and *fast feedback*—are mutually exclusive. It's impossible to maximize them all: you have to sacrifice one of the three in order to max out the remaining two.

Moreover, because of the multiplication principle (see the calculation of the value estimate in the previous section), it's even trickier to keep the balance. You can't just forgo one of the attributes in order to focus on the others. As I mentioned previously, a test that scores zero in one of the four categories is worthless. Therefore, you have to maximize these attributes in such a way that none of them is diminished too much. Let's look at some examples of tests that aim at maximizing two out of three attributes at the expense of the third and, as a result, have a value that's close to zero.

4.4.2 Extreme case #1: End-to-end tests

The first example is end-to-end tests. As you may remember from chapter 2, end-to-end tests look at the system from the end user's perspective. They normally go through all of the system's components, including the UI, database, and external applications.

Since end-to-end tests exercise a lot of code, they provide the best protection against regressions. In fact, of all types of tests, end-to-end tests exercise the most code—both your code and the code you didn't write but use in the project, such as external libraries, frameworks, and third-party applications.

End-to-end tests are also immune to false positives and thus have a good resistance to refactoring. A refactoring, if done correctly, doesn't change the system's observable behavior and therefore doesn't affect the end-to-end tests. That's another advantage of such tests: they don't impose any particular implementation. The only thing end-to-end tests look at is how a feature behaves from the end user's point of view. They are as removed from implementation details as tests could possibly be.

However, despite these benefits, end-to-end tests have a major drawback: they are slow. Any system that relies solely on such tests would have a hard time getting rapid feedback. And that is a deal-breaker for many development teams. This is why it's pretty much impossible to cover your code base with only end-to-end tests.

Figure 4.6 shows where end-to-end tests stand with regard to the first three unit testing metrics. Such tests provide great protection against both regression errors and false positives, but lack speed.

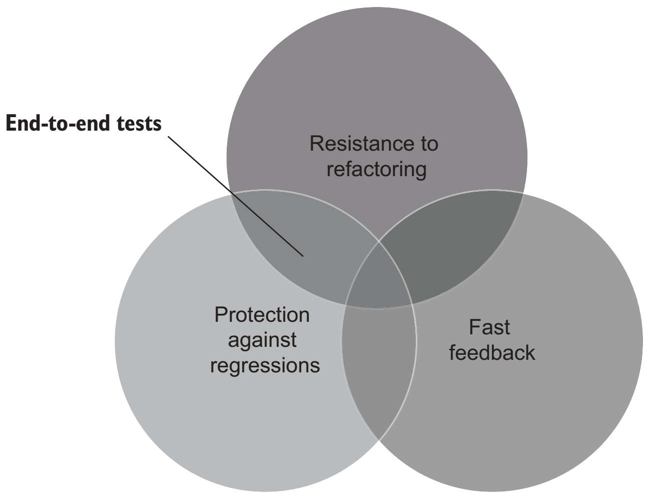

Figure 4.6 End-to-end tests provide great protection against both regression errors and false positives, but they fail at the metric of fast feedback.

4.4.3 Extreme case #2: Trivial tests

Another example of maximizing two out of three attributes at the expense of the third is a *trivial test*. Such tests cover a simple piece of code, something that is unlikely to break because it's too trivial, as shown in the following listing.

Listing 4.5 Trivial test covering a simple piece of code

```
public class User
{
    public string Name { get; set; }    <- One-liners like this are unlikely to contain bugs.
}

[Fact]
public void Test()
{
    var sut = new User();

    sut.Name = "John Smith";

    Assert.Equal("John Smith", sut.Name);
}
```

Unlike end-to-end tests, trivial tests do provide fast feedback—they run very quickly. They also have a fairly low chance of producing a false positive, so they have good resistance to refactoring. Trivial tests are unlikely to reveal any regressions, though, because there's not much room for a mistake in the underlying code.

Trivial tests taken to an extreme result in *tautology* tests. They don't test anything because they are set up in such a way that they always pass or contain semantically meaningless assertions.

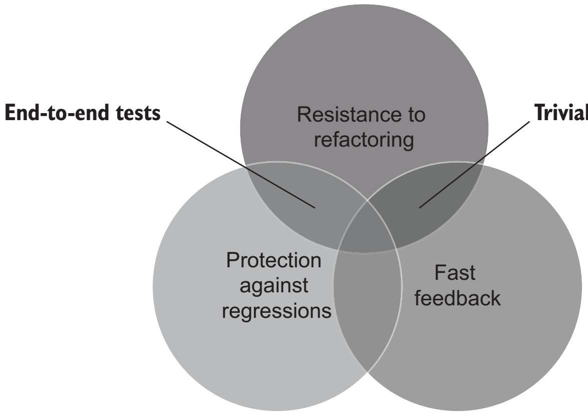

Figure 4.7 Trivial tests have good resistance to refactoring, and they provide fast feedback, but such tests don't protect you from regressions.

Figure 4.7 shows where trivial tests stand. They have good resistance to refactoring and provide fast feedback, but they don't protect you from regressions.

4.4.4 Extreme case #3: Brittle tests

Similarly, it's pretty easy to write a test that runs fast and has a good chance of catching a regression but does so with a lot of false positives. Such a test is called a *brittle test*: it can't withstand a refactoring and will turn red regardless of whether the underlying functionality is broken.

You already saw an example of a brittle test in listing 4.2. Here's another one.

Listing 4.6 Test verifying which SQL statement is executed

```
public class UserRepository
{
    public User GetById(int id)
    {
        /* ... */
    }

    public string LastExecutedSqlStatement { get; set; }
}

[Fact]
public void GetById_executes_correct_SQL_code()
{
    var sut = new UserRepository();

    User user = sut.GetById(5);

    Assert.Equal(
        "SELECT * FROM dbo.[User] WHERE UserID = 5",
        sut.LastExecutedSqlStatement);
}
```

This test makes sure the `UserRepository` class generates a correct SQL statement when fetching a user from the database. Can this test catch a bug? It can. For example, a developer can mess up the SQL code generation and mistakenly use `ID` instead of `UserID`, and the test will point that out by raising a failure. But does this test have good resistance to refactoring? Absolutely not. Here are different variations of the SQL statement that lead to the same result:

```
SELECT * FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.User WHERE UserID = 5
SELECT UserID, Name, Email FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.[User] WHERE UserID = @UserID
```

The test in listing 4.6 will turn red if you change the SQL script to any of these variations, even though the functionality itself will remain operational. This is once again an example of coupling the test to the SUT's internal implementation details. The test is focusing on *hows* instead of *whats* and thus ingrains the SUT's implementation details, preventing any further refactoring.

Figure 4.8 shows that brittle tests fall into the third bucket. Such tests run fast and provide good protection against regressions but have little resistance to refactoring.

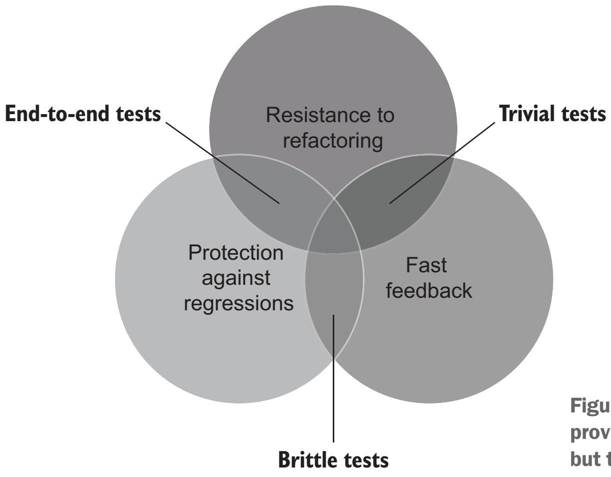

Figure 4.8 Brittle tests run fast and they provide good protection against regressions, but they have little resistance to refactoring.

4.4.5 In search of an ideal test: The results

The first three attributes of a good unit test (*protection against regressions*, *resistance to refactoring*, and *fast feedback*) are mutually exclusive. While it's quite easy to come up with a test that maximizes two out of these three attributes, you can only do that at the expense of the third. Still, such a test would have a close-to-zero value due to the multiplication rule. Unfortunately, it's impossible to create an ideal test that has a perfect score in all three attributes (figure 4.9).

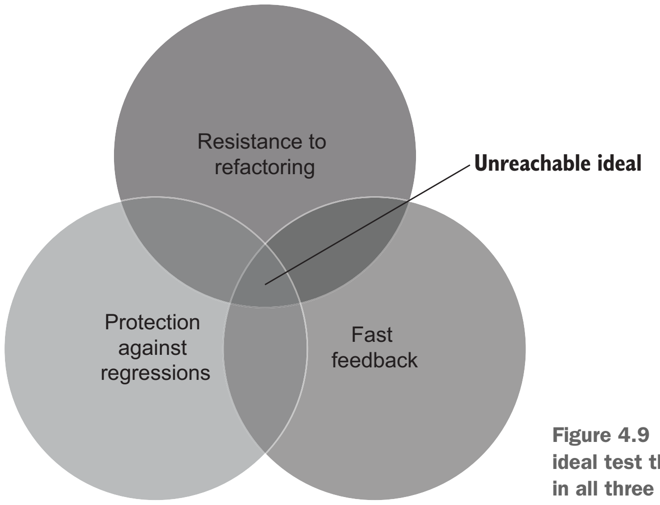

Figure 4.9 It's impossible to create an ideal test that would have a perfect score in all three attributes.

The fourth attribute, *maintainability*, is not correlated to the first three, with the exception of end-to-end tests. End-to-end tests are normally larger in size because of the necessity to set up all the dependencies such tests reach out to. They also require additional effort to keep those dependencies operational. Hence end-to-end tests tend to be more expensive in terms of maintenance costs.

It's hard to keep a balance between the attributes of a good test. A test can't have the maximum score in each of the first three categories, and you also have to keep an eye on the maintainability aspect so the test remains reasonably short and simple. Therefore, you have to make trade-offs. Moreover, you should make those trade-offs in such a way that no particular attribute turns to zero. The sacrifices have to be partial and strategic.

What should those sacrifices look like? Because of the mutual exclusiveness of *protection against regressions*, *resistance to refactoring*, and *fast feedback*, you may think that the best strategy is to concede a little bit of each: just enough to make room for all three attributes.

In reality, though, *resistance to refactoring* is non-negotiable. You should aim at gaining as much of it as you can, provided that your tests remain reasonably quick and you don't resort to the exclusive use of end-to-end tests. The trade-off, then, comes down to the choice between how good your tests are at pointing out bugs and how fast they do that: that is, between *protection against regressions* and *fast feedback*. You can view this choice as a slider that can be freely moved between *protection against regressions* and *fast feedback*. The more you gain in one attribute, the more you lose on the other (see figure 4.10).

The reason *resistance to refactoring* is non-negotiable is that whether a test possesses this attribute is mostly a binary choice: the test either has resistance to refactoring or it doesn't. There are almost no intermediate stages in between. Thus you can't concede

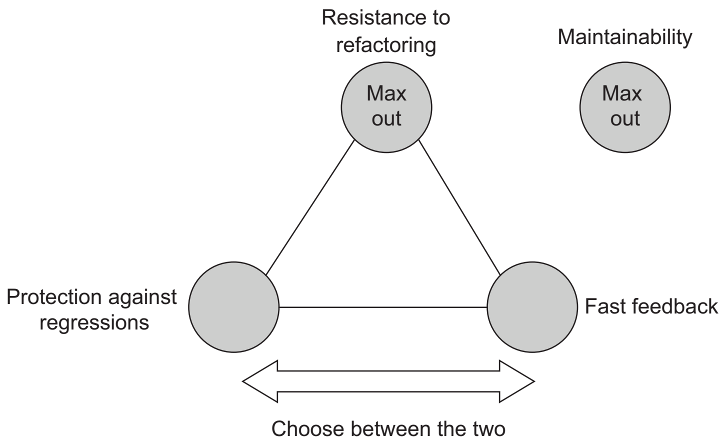

Figure 4.10 The best tests exhibit maximum maintainability and resistance to refactoring; always try to max out these two attributes. The trade-off comes down to the choice between protection against regressions and fast feedback.

just a little *resistance to refactoring*: you'll have to lose it all. On the other hand, the metrics of *protection against regressions* and *fast feedback* are more malleable. You will see in the next section what kind of trade-offs are possible when you choose one over the other.

TIP Eradicating brittleness (false positives) in tests is the first priority on the path to a robust test suite.

The CAP theorem

The trade-off between the first three attributes of a good unit test is similar to the CAP theorem. The CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two of the following three guarantees:

- *Consistency*, which means every read receives the most recent write or an error.
- *Availability*, which means every request receives a response (apart from outages that affect all nodes in the system).
- *Partition tolerance*, which means the system continues to operate despite network partitioning (losing connection between network nodes).

The similarity is two-fold:

- First, there is the *two-out-of-three* trade-off.
- Second, the *partition tolerance* component in large-scale distributed systems is also non-negotiable. A large application such as, for example, the Amazon website can't operate on a single machine. The option of preferring *consistency* and *availability* at the expense of *partition tolerance* simply isn't on the table—Amazon has too much data to store on a single server, however big that server is.

The choice, then, also boils down to a trade-off between *consistency* and *availability*. In some parts of the system, it's preferable to concede a little consistency to gain more availability. For example, when displaying a product catalog, it's generally fine if some parts of the catalog are out of date. *Availability* is of higher priority in this scenario. On the other hand, when updating a product description, *consistency* is more important than *availability*: network nodes must have a consensus on what the most recent version of that description is, in order to avoid merge conflicts.

4.5 Exploring well-known test automation concepts

The four attributes of a good unit test shown earlier are foundational. All existing, well-known test automation concepts can be traced back to these four attributes. In this section, we'll look at two such concepts: the Test Pyramid and white-box versus black-box testing.

4.5.1 Breaking down the Test Pyramid

The *Test Pyramid* is a concept that advocates for a certain ratio of different types of tests in the test suite (figure 4.11):

- Unit tests
- Integration tests
- End-to-end tests

The Test Pyramid is often represented visually as a pyramid with those three types of tests in it. The width of the pyramid layers refers to the prevalence of a particular type

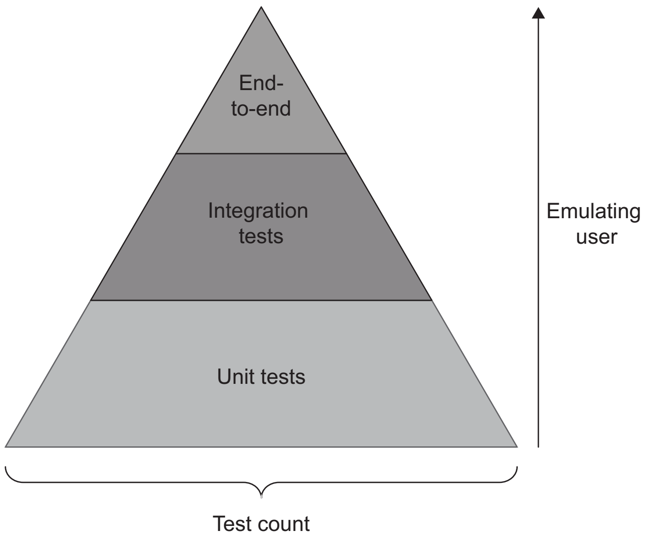

Figure 4.11 The Test Pyramid advocates for a certain ratio of unit, integration, and end-to-end tests.

of test in the suite. The wider the layer, the greater the test count. The height of the layer is a measure of how close these tests are to emulating the end user's behavior. End-to-end tests are at the top—they are the closest to imitating the user experience. Different types of tests in the pyramid make different choices in the trade-off between *fast feedback* and *protection against regressions*. Tests in higher pyramid layers favor *protection against regressions*, while lower layers emphasize execution speed (figure 4.12).

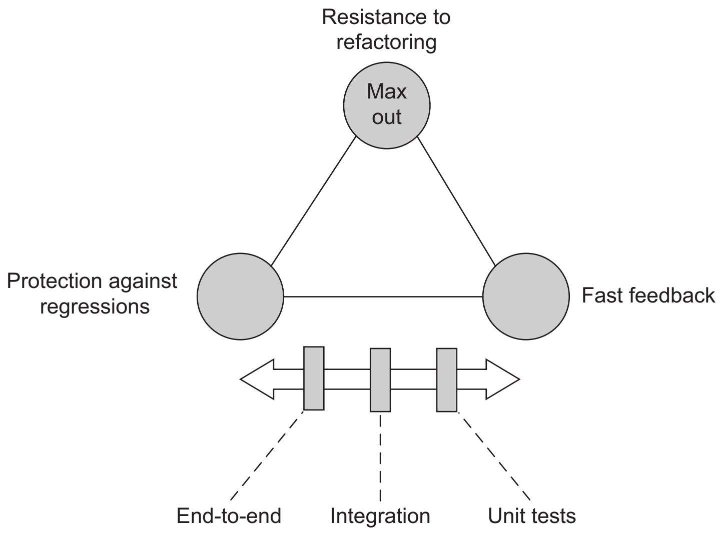

Figure 4.12 Different types of tests in the pyramid make different choices between fast feedback and protection against regressions. End-to-end tests favor protection against regressions, unit tests emphasize fast feedback, and integration tests lie in the middle.

Notice that neither layer gives up *resistance to refactoring*. Naturally, end-to-end and integration tests score higher on this metric than unit tests, but only as a side effect of being more detached from the production code. Still, even unit tests should not concede *resistance to refactoring*. All tests should aim at producing as few false positives as possible, even when working directly with the production code. (How to do that is the topic of the next chapter.)

The exact mix between types of tests will be different for each team and project. But in general, it should retain the pyramid shape: end-to-end tests should be the minority; unit tests, the majority; and integration tests somewhere in the middle.

The reason end-to-end tests are the minority is, again, the multiplication rule described in section 4.4. End-to-end tests score extremely low on the metric of *fast feedback*. They also lack *maintainability*: they tend to be larger in size and require additional effort to maintain the involved out-of-process dependencies. Thus, end-to-end tests only make sense when applied to the most critical functionality—features in which you don't ever want to see any bugs—and only when you can't get the same degree of protection with unit or integration tests. The use of end-to-end tests for anything else shouldn't pass your minimum required value threshold. Unit tests are usually more balanced, and hence you normally have many more of them.

There are exceptions to the Test Pyramid. For example, if all your application does is basic create, read, update, and delete (CRUD) operations with very few business rules or any other complexity, your test "pyramid" will most likely look like a rectangle with an equal number of unit and integration tests and no end-to-end tests.

Unit tests are less useful in a setting without algorithmic or business complexity—they quickly descend into trivial tests. At the same time, integration tests retain their value—it's still important to verify how code, however simple it is, works in integration with other subsystems, such as the database. As a result, you may end up with fewer unit tests and more integration tests. In the most trivial examples, the number of integration tests may even be greater than the number of unit tests.

Another exception to the Test Pyramid is an API that reaches out to a single out-of-process dependency—say, a database. Having more end-to-end tests may be a viable option for such an application. Since there's no user interface, end-to-end tests will run reasonably fast. The maintenance costs won't be too high, either, because you only work with the single external dependency, the database. Basically, end-to-end tests are indistinguishable from integration tests in this environment. The only thing that differs is the entry point: end-to-end tests require the application to be hosted somewhere to fully emulate the end user, while integration tests normally host the application in the same process. We'll get back to the Test Pyramid in chapter 8, when we'll be talking about integration testing.

4.5.2 Choosing between black-box and white-box testing

The other well-known test automation concept is black-box versus white-box testing. In this section, I show when to use each of the two approaches:

- *Black-box testing* is a method of software testing that examines the functionality of a system without knowing its internal structure. Such testing is normally built around specifications and requirements: *what* the application is supposed to do, rather than *how* it does it.
- *White-box testing* is the opposite of that. It's a method of testing that verifies the application's inner workings. The tests are derived from the source code, not requirements or specifications.

There are pros and cons to both of these methods. White-box testing tends to be more thorough. By analyzing the source code, you can uncover a lot of errors that you may miss when relying solely on external specifications. On the other hand, tests resulting from white-box testing are often brittle, as they tend to tightly couple to the specific implementation of the code under test. Such tests produce many false positives and thus fall short on the metric of *resistance to refactoring*. They also often can't be traced back to a behavior that is meaningful to a business person, which is a strong sign that these tests are fragile and don't add much value. Black-box testing provides the opposite set of pros and cons (table 4.1).

Table 4.1 The pros and cons of white-box and black-box testing

	Protection against regressions	Resistance to refactoring
White-box testing	Good	Bad
Black-box testing	Bad	Good

As you may remember from section 4.4.5, you can't compromise on *resistance to refactoring*: a test either possesses *resistance to refactoring* or it doesn't. Therefore, *choose black-box testing over white-box testing by default.* Make all tests—be they unit, integration, or end-to-end—view the system as a black box and verify behavior meaningful to the problem domain. If you can't trace a test back to a business requirement, it's an indication of the test's brittleness. Either restructure or delete this test; don't let it into the suite as-is. The only exception is when the test covers utility code with high algorithmic complexity (more on this in chapter 7).

Note that even though black-box testing is preferable when *writing tests*, you can still use the white-box method when *analyzing* the tests. *Use code coverage tools to see which code branches are not exercised, but then turn around and test them as if you know nothing about the code's internal structure.* Such a combination of the white-box and black-box methods works best.

Summary

- A good unit test has four foundational attributes that you can use to analyze any automated test, whether unit, integration, or end-to-end:
  - Protection against regressions
  - Resistance to refactoring
  - Fast feedback
  - Maintainability
- *Protection against regressions* is a measure of how good the test is at indicating the presence of bugs (regressions). The more code the test executes (both your code and the code of libraries and frameworks used in the project), the higher the chance this test will reveal a bug.
- *Resistance to refactoring* is the degree to which a test can sustain application code refactoring without producing a false positive.
- A false positive is a false alarm—a result indicating that the test fails, whereas the functionality it covers works as intended. False positives can have a devastating effect on the test suite:
  - They dilute your ability and willingness to react to problems in code, because you get accustomed to false alarms and stop paying attention to them.

  - They diminish your perception of tests as a reliable safety net and lead to losing trust in the test suite.
- False positives are a result of tight coupling between tests and the internal implementation details of the system under test. To avoid such coupling, the test must verify the end result the SUT produces, not the steps it took to do that.
- *Protection against regressions* and *resistance to refactoring* contribute to test accuracy. A test is accurate insofar as it generates a strong signal (is capable of finding bugs, the sphere of *protection against regressions*) with as little noise (false positives) as possible (the sphere of *resistance to refactoring*).
- False positives don't have as much of a negative effect in the beginning of the project, but they become increasingly important as the project grows: as important as false negatives (unnoticed bugs).
- *Fast feedback* is a measure of how quickly the test executes.
- *Maintainability* consists of two components:
  - How hard it is to understand the test. The smaller the test, the more readable it is.
  - How hard it is to run the test. The fewer out-of-process dependencies the test reaches out to, the easier it is to keep them operational.
- A test's value estimate is the product of scores the test gets in each of the four attributes. If the test gets zero in one of the attributes, its value turns to zero as well.
- It's impossible to create a test that gets the maximum score in all four attributes, because the first three—*protection against regressions*, *resistance to refactoring*, and *fast feedback*—are mutually exclusive. The test can only maximize two out of the three.
- *Resistance to refactoring* is non-negotiable because whether a test possess this attribute is mostly a binary choice: the test either has resistance to refactoring or it doesn't. The trade-off between the attributes comes down to the choice between *protection against regressions* and *fast feedback*.
- The Test Pyramid advocates for a certain ratio of unit, integration, and end-to-end tests: end-to-end tests should be in the minority, unit tests in the majority, and integration tests somewhere in the middle.
- Different types of tests in the pyramid make different choices between *fast feedback* and *protection against regressions*. End-to-end tests favor *protection against regressions*, while unit tests favor *fast feedback*.
- Use the black-box testing method when *writing* tests. Use the white-box method when *analyzing* the tests.