# 15

# Clean Tests

Clean tests are readable, fast, isolated, repeatable, self-verifying, timely, and designed. Readability is perhaps even more important in unit tests than it is in production code. What makes tests readable? The same things that make all code readable: clarity, simplicity, and density of expression. In a test, you want to say a lot with as few expressions as reasonable.

Consider this code from FitNesse. These three tests are difficult to understand and can certainly be improved. First, there is a terrible amount of duplicate code in the repeated calls to `addPage` and `assertSubString`. More importantly, this code is just loaded with details that interfere with the expressiveness of the test.

```
public void testGetPageHierarchyAsXml() throws Except
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.Chi
  crawler.addPage(root, PathParser.parse("PageTwo"));
  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder()
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType())
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHierarchyAsXmlDoesntContainSym
throws Exception {
  WikiPage pageOne =
    crawler.addPage(root, PathParser.parse("PageOne")
  crawler.addPage(root, PathParser.parse("PageOne.Chi
  crawler.addPage(root, PathParser.parse("PageTwo"));

  PageData data = pageOne.getData();
  WikiPageProperties properties = data.getProperties(
  WikiPageProperty symLinks =
    properties.set(SymbolicPage.PROPERTY_NAME);
  symLinks.set("SymPage", "PageTwo");
  pageOne.commit(data);

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder()
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType())
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
  assertNotSubString("SymPage", xml);
}

public void testGetDataAsHtml() throws Exception {
  crawler.addPage(root, PathParser.parse("TestPageOne

  request.setResource("TestPageOne");
  request.addInput("type", "data");
  Responder responder = new SerializedPageResponder()
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType())
  assertSubString("test page", xml);
  assertSubString("<Test", xml);
}
```

For example, look at the `PathParser` calls. They transform strings into `PagePath` instances used by the `crawlers`. This transformation is completely irrelevant to the test at hand and serves only to obfuscate the intent. The details surrounding the creation of the responder and the gathering and casting of the response are also just noise. Then there's the ham-handed way that the request URL is built from a resource and an argument. (I helped write this code 20 years ago, so I feel free to roundly criticize it.)

In the end, this code was not designed to be read. The poor reader is inundated with a swarm of details that must be understood before the tests make any real sense.

Now consider the improved tests below. These tests do the exact same thing, but they have been refactored into a much cleaner and more explanatory form.

```
  public void testGetPageHierarchyAsXml() throws Except
    makePages("PageOne", "PageOne.ChildOne", "PageTwo")

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
      "<name>PageOne</name>",
      "<name>PageTwo</name>",
      "<name>ChildOne</name>");
```

```
}

public void
testSymbolicLinksAreNotInXmlPageHierarchy() throws Ex
  WikiPage page = makePage("PageOne");
  makePages("PageOne.ChildOne", "PageTwo");

  addLinkTo(page, "PageTwo", "SymPage");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>",
    "<name>PageTwo</name>",
    "<name>ChildOne</name>"
  );
  assertResponseDoesNotContain("SymPage");
}

public void testGetDataAsXml() throws Exception {
  makePageWithContent("TestPageOne", "test page");

  submitRequest("TestPageOne", "type:data");

  assertResponseIsXML();
  assertResponseContains("test page", "<Test");
}
```

The Arrange/Act/Assert (AAA)[1] pattern is made obvious by the structure of these tests. Each of the tests is clearly split into three parts. The first part (*Arrange*) builds up the test data, the second part (*Act*) operates on that test data, and the third part (*Assert*) checks that the operation yielded the expected results.

---

1. https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/

Notice that the vast majority of annoying detail has been eliminated. The tests get right to the point and use only the data types and functions that they truly need. Anyone who reads these tests should be able to work out what they do very quickly, without being misled or overwhelmed by details.

## Domain-Specific Testing Language

The tests above demonstrate the technique of building a domain-specific language for your tests. Rather than using the APIs that programmers use to manipulate the system, we build up a set of functions and utilities that make use of those APIs and that make the tests more convenient to write and easier to read. These functions and utilities become a specialized API used by the tests. They are a testing language that programmers use to help them write their tests and to help those who must read those tests later on.

### Composed Assertions

The functions `assertResponseIsXML` and `assertResponseContains` are part of that domain-specific testing language. These kinds of assertion functions are called *composed assertions*.

Testing APIs like these are not designed up front; rather, they evolve from the continued refactoring of test code that has gotten too tainted by obfuscating detail. Just as you saw me refactor the original tests, so too will disciplined developers refactor their test code into more succinct and expressive forms.

### Composed Test Results

Another technique for creating a domain-specific testing API is to create composed test results. These are used when the test results are needlessly complex and can be compressed down into a smaller form that is easy to understand.

Consider the tests I wrote as part of an environment control system I was prototyping. Without going into the details, you can tell that this test checks that the low-temperature alarm, the heater, and the blower are all turned on when the temperature is “way too cold.”

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exc
```

```
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
  }
```

There are, of course, lots of details here. For example, what is that `tic` function all about? In fact, I'd rather you not worry about that while reading this test. I'd rather you just worry about whether you agree that the end state of the system is consistent with the temperature being "way too cold."

Notice, as you read the test, that your eye needs to bounce back and forth between the name of the state being checked and the *sense* of the state being checked. You see `heaterState`, and then your eyes glissade left to `assertTrue`. You see `coolerState,` and your eyes must track left to `assertFalse`. This is tedious and unreliable. It makes the test hard to read.

To make matters worse, this is one of 14 very similar tests. Imagine how your eyes will cross as you try to read all that.

I improved the reading of these tests greatly by transforming them as follows.

```
  @Test
  public void turnOnLoTempAlarmAtThreshold() throws Exc
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

Of course, I hid the detail of the `tic` function by creating a `wayTooCold` function. But the thing to note is the strange string in the `assertEquals`. Uppercase means "on," lowercase means "off," and the letters are always in the following order: {heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}.

Once you know the meaning, your eyes glide across that string and you can quickly interpret the results. Reading the test becomes almost a pleasure. Just take a look at a few of these tests and see how easy they are to comprehend.

```
@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Ex
  tooHot();
  assertEquals("hBChl", hw.getState());
}
@Test
public void turnOnHeaterAndBlowerIfTooCold() throws E
  tooCold();
  assertEquals("HBchl", hw.getState());
}
@Test
public void turnOnHiTempAlarmAtThreshold() throws Exc
  wayTooHot();
  assertEquals("hBCHl", hw.getState());
}
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exc
  wayTooCold();
  assertEquals("HBchL", hw.getState());
}
```

The `getState` function that creates the composed test result is shown below.

```
public String getState() {
  String state = "";
  state += heater ? "H" : "h";
  state += blower ? "B" : "b";
  state += cooler ? "C" : "c";
  state += hiTempAlarm ? "H" : "h";
  state += loTempAlarm ? "L" : "l";
  return state;
}
```

## Dual Standard

You might complain that composing the test result in this fashion is wasteful of computing resources. After all, I didn’t even use a `StringBuffer` or `StringBuilder` to do the concatenation.

This application is clearly an embedded real-time system, and it is likely that computer and memory resources are very constrained. However, the test environment runs in my laptop and is not constrained at all.

That is the nature of the dual standard between test and production. There are things that you might never do in a production environment that are perfectly fine in a test environment. Usually they involve issues of memory or CPU efficiency. But they never involve issues of cleanliness.

## The Single Assert Rule

This rule states that a good test asserts one and only one thing. This is often misunderstood to mean that a test should have only a single assertion statement. In reality, a test can have many assertion statements and still assert one logical fact.

For example, consider the first set of environment controller tests above. They may be hard to read with all those assertions, but they are asserting a single logical fact. Indeed, that is why we could refactor them with composed test results. Or, consider the refactored assertions in the `pageHierarchy` example above. They are asserting the single logical fact that the response is an XML document that has certain contents.

You might find this confusing, or at least subjective. After all, the contents of the document and the type of the document are different facts, aren’t they?

The confusion arises because the Single Assert Rule is poorly named.

## The Single Act Rule

The rule should really be called the Single Act Rule. The Arrange-Act-Assert pattern says that every test should be composed of three elements: the arrangement, the action, and the assertion. The first arranges the data to be tested. The second is the action upon that data that we wish to test. The third is the assertion that the data has been properly transformed by the action.

The goal of the Single Assert Rule is to make sure that each test is testing *one and only one* action. We want to avoid tests that arrange, act, assert, act, assert. We also want to avoid tests that arrange, act, act, assert, assert. We want every action tested individually because we don't want downstream assertions to be corrupted by upstream actions. We want every test to stand alone.

## F.I.R.S.T.

Here is a simple memory aid for the desired characteristics of tests.

### Fast

Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently. If you don't run them frequently, you won't find problems early enough to fix them easily. You won't feel as free to clean up the code. Eventually the code will begin to rot. So consider the speed of the tests to be a design imperative. Design your tests to be fast. Very fast.

### Isolated

This is a restatement of the Single Act Rule. Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

### Repeatable

Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren't repeatable in any environment, then you'll always have an excuse for why they fail. You’ll also find yourself unable to run the tests when the environment isn’t available.

### Self-Validating

The tests should have a boolean output. Either they pass or they fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren’t self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

### Timely

The tests need to be written in a timely fashion. Unit tests should be written along with the production code that makes them pass. If you write tests long after the production code, then you may find the production code to be too hard to test. Timely tests cause the design of the production code to be testable.

## Test Design

As we saw in Chapter 9, “The Clean Method,” tests need to be designed to reduce their coupling to the production code. This makes sure that changes to the requirements affect as few tests as possible.

Suppose you are working on a large system composed of hundreds of modules. Suppose a single change to that system forces you to make changes to a majority of those modules. That system is poorly designed—by definition.

The principles and rules of software design, when carefully followed, prevent—or at least mitigate—that nightmare scenario.

Now imagine you have a well-designed system with hundreds of modules and a test suite with thousands of unit tests. Suppose a single change to that system breaks hundreds of those tests. That is a very poorly designed suite of tests.

The principles and rules of software design apply to the tests as much as they apply to the production code. Coupling must be minimized. One change to the signature of a function in the production code must not cause hundreds of tests to be altered, just as it must not cause hundreds of other modules to be altered.

## Conclusion

We have barely scratched the surface of this topic. My book *Clean Craftsmanship* goes very deep into the topic of clean and well-designed tests. Tests are as important to the health of a project as the production code is. Perhaps they are even more important, because tests preserve and enhance the flexibility, maintainability, and reusability of the production code. So keep your tests constantly clean and well designed. Work to make them expressive and succinct. Invent testing APIs that act as domain-specific languages that help you write the tests.

If you let the tests rot, then your code will rot too. Maintain your disciplines and keep your tests clean.