45 FIRST Properties of Unit Tests

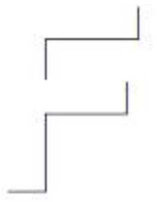
- **F**ast
- **I**solated
- **R**epeatable
- **S**elf-verifying
- **T**imely

A unit test is a small automated test, coded by a programmer, that verifies whether or not a small piece of production code—a unit—works as expected in isolation. The moniker unit test was popularized with the advent of tools such as SUnit (for Smalltalk) and JUnit. The XP crowd used the term to mean tests that were produced as a result of programmers practicing test-driven development (TDD). Some developers balked—the phrase already had a considerably different meaning in some circles.

Mike Hill of Industrial Logic proposed the unique term microtest, along with which he provided a detailed definition to avoid any confusion. Michael Feathers proposed a similar definition that listed five things a unit test is not. Both sets of criteria are great guidelines for building unit tests. But is there a more concise way to capture and remember them?

Tim Ottinger and Brett Schuchert, Object Mentors at the time, devised the FIRST mnemonic for a more concise set of criteria for effective unit tests. The acronym is overloaded: its very name answers the important question of when we write tests: FIRST.

F for Fast



The faster your tests run, the more often you'll run them. Tests take half a minute to run? You'll maybe run them every 5 minutes instead of every few seconds. Tests take over two minutes to run? You might run those tests once every half hour, and you can write a lot of questionable code in that time! Tests take ten, fifteen, or more minutes to run? Forget it. You'll run those a few times per day at best.

Developers cope with long-running test suites by running a subset of the tests. That's a guessing game that can work reasonably well as long as you run all of the tests before check-in. It is faster in the short term, but far less certain. All it takes is one extensive debugging session to wipe out the time savings. Unit tests save you time by finding errors immediately after you make them.

You can better tolerate slow tests if a Continual Testing (CT) tool runs them whenever code changes. Smart CT tools like Infinitest will calculate which tests to run based on dependencies in the code. Either way, CT makes running tests less painful. But it is better still if your microtests are blazing fast.

The primary goal of the following test is to verify that HasArticles answers true when articles are present in a feed. It takes almost half a second on a beefy Mac with a fast internet connection. How many reasons can you spot?

```
class testme(unittest.TestCase):
def test_recognizesEntries(self):
url = "http://agileinaflash.com/feeds/posts/default"
content = urllib2.urlopen(url).read() # 1
self.assertGreater(len(content),0) # 2
open("localfeed.rss", "w").write(content) # 3
sut = ArticleRssParser('localfeed.rss') # 4
self.assertTrue(sut.HasArticles())
```

Examine the statements at the comment markers.

1. The test goes to the web for data.
2. The test fails if the RSS feed is empty, without testing HasArticles at all.
3. Writing to the file system increases run time. (The untidy test doesn't delete this file; correcting this defect adds another file system round-trip to the test time.)
4. The RSS parser class reads from a file.

Instead of accessing a web page, you could write a sample RSS document with a couple “faked” entries to a file. A fast file system might bring the cost of that write to perhaps a tenth of a second. But do you need the file handling at all? If you extract the ArticleRssParser’s file handling into a separate class method, you can easily bypass the file system:

```
class ArticleRssParser(object):
@classmethod
def get_feed_content(cls, filename):
return open(filename).read()
# … the rest of the class elided
```

(We use a class method because we want to stub out a behavior that exists in the constructor, otherwise we’d extract to a normal instance method.)

To string this all together, we’ll use the mocking library pymox to create a record/playback test:

```
class basic_feed_parsing(unittest.TestCase):
def test_reports_it_has_articles_from_2_article_feed(self):
mocks = mox.Mox()
mocks.StubOutWithMock(ArticleRssParser, 'get_feed_content')
ArticleRssParser\
.get_feed_content('fakefilename')\
.AndReturn(sample_with_2_entries)
mocks.ReplayAll()
parser = ArticleRssParser("fakefilename")
self.assertTrue(parser.HasArticles())
mocks.VerifyAll()
```

Running this test requires neither web access nor file access.

The test’s new line count is higher, and its setup more complicated. As you write additional tests, you’ll be able to abstract away more details. Here’s a later, refactored version:

```
def test_reports_it_has_articles_from_2_article_feed(self):
self.mocks.StubOutWithMock(ArticleRssParser, 'get_feed_content')
ArticleRssParser\
.get_feed_content('fakefilename')\
.AndReturn(sample_rss_with_2_entries)
with replay(self.mocks):
parser = ArticleRssParser("fakefilename")
self.assertTrue(parser.HasArticles())
```

This test usually runs in under 0.02 seconds on Tim's machine. Much faster, but still unacceptable if your suite contains thousands of similarly slow tests.

It turns out that this class uses the python feedparser library, which is already well-tested and reliable. Since you don't need to re-test feedparser, you can speed up the test by stubbing it out.

```
def test_reports_it_has_articles_from_2_article_feed(self):
result_with_2_entries = {"entries":[1,2]}
self.mocks.StubOutWithMock(ArticleRssParser, 'parse_input')
ArticleRssParser\
.parse_input('fakefilename')\
.AndReturn(result_with_2_entries)
with replay(self.mocks):
parser = ArticleRssParser("fakefilename")
self.assertTrue(parser.HasArticles())
```

Result? Runtime disappears into the noise. A unit test in python takes 0.012 to 0.015s if it does nothing but assertTrue(True). This version of the test takes no measurable time at all. The test has assumptions about feedparser, but it still shows that you're handling the results correctly.

Now we can afford to add a large number of similar tests without long automated test cycles sucking the joy out of our day.

I for Isolated

Each unit test should have a single reason to fail. Our prior example fails this criterion: the test would fail if the file system was full or read-only, if the file existed and was not writeable, if the network was down, if there were no articles in the RSS feed, or if it did not recognize entries in the feed.

You must design your tests to be independent not only of external factors, but of each other as well. When tests are interdependent, a change to one test can cause several others to fail in puzzling ways. Ordering unit tests to optimize their execution is a sign that isolation is poor. Each unit test should instead stand on its own as a complete case that documents one discrete behavior.

Dave Astel's recommendation of one assert per test promotes tests that are I for isolated. Here's a test that verifies aspects of both checking out and checking in books:

```
@Test
public void testAvailability() {
theTrialHolding.checkOut(TODAY);
assertFalse(theTrialHolding.isAvailable());

theTrialHolding.checkIn(TOMORROW, BranchTestData.ROCKRIMMON_BRANCH);
assertTrue(theTrialHolding.isAvailable());
}
```

Many unit testers would leave this test alone, but it’s really testing aspects of two behaviors. Splitting it into two single-assertion tests makes the reason for any failure immediate. A split also allows you to improve the test names:

```
@Test
public void isNoLongerAvailableAfterCheckout() {
theTrialHolding.checkOut(TODAY);

assertFalse(theTrialHolding.isAvailable());
}

@Test
public void isAvailableAfterCheckin() {
theTrialHolding.checkOut(TODAY);

theTrialHolding.checkIn(TOMORROW, BranchTestData.ROCKRIMMON_BRANCH);


assertTrue(theTrialHolding.isAvailable());
}
```

Such single-purpose tests make AAA (Arrange-Act-Assert) notation effective.

The sheer size of most systems means that no one knows the entire code base by heart. You must design your test suite names, your test names, and your assert messages to reduce and simplify investigation.

R for Repeatable

You should obtain the same results every time you run a test. Tests can fail intermittently for a few reasons; here are the most common:

1. Static data or other in-memory constructs not cleaned up
2. Volatility of external services (e.g. file system, database, web services, API calls)
3. Non-deterministic behavior due to incorrect use of threads/processes
4. Dependence on an uninitialized (or slow-initializing) local class
5. Over-specification (for example, comparing entire screen images or HTML when only a small part of the result is interesting)

Intermittently-failing tests can be difficult to decipher. If you are uncertain about the current state of things, add precondition assertions. If any precondition is not met, the test will stop before it has a chance to run, and you'll know exactly why:

```
@Test
public void addCustomer() {
assertThat(customerStore.retrieve(ACME_ID), is(null));
customerStore.add(new Customer(ACME_ID));
// ...
}
```

Testing threads is a more difficult challenge. You might figure out how to test multi-threaded code, observe the test passing dozens of times in a row, and then watch it curiously fail. Some developers will say—perhaps unhelpfully—that testing with multiple threads is more of an integration test than a unit test. A discussion on threaded testing at Stack Overflow is a good place to start, and will give you an idea of the kinds of challenges you're up against.

Increasing isolation tends to enhance repeatability.

S for Self-Verifying

A good unit test fails or passes unambiguously. When all tests run green, you have high confidence that you can ship the code to the next level (likely the acceptance test automated suite). If any of the tests fail, you don’t proceed until they get fixed. The pass/fail nature of the automated testing system makes it viable even for large teams.

When a test suite leaves some results open to human interpretation, validating them becomes a drag on the productivity of all programmers. Ambiguous tests fail to be helpful and fall into disuse.

A classic way to game the code coverage metric is to write a test that exercises a broad swath of code, yet never assert a thing.

```
@Test
public void createReport() {
Catalog catalog = new Catalog();
catalog.add(new Holding(BookTestData.AGILE_JAVA));
catalog.add(new Holding(BookTestData.JAWS));
catalog.add(new Holding(BookTestData.THE_TRIAL));

InventoryReport report = new InventoryReport(catalog);
System.out.println(report.allBooks());
}
```

If the system misbehaves, there is a rather dubious hope that a programmer will be monitoring the console output and will know what output to expect.

Coded tests should instead actually verify something in an automated fashion:

```
@Test
public void createReport() {
Catalog catalog = new Catalog();
// ...
InventoryReport report = new InventoryReport(catalog);
assertThat(report.contains(BookTestData.AGILE_JAVA.getClassification());
// ...
}
```

Do programmers really write unit tests that test nothing? Sadly, yes. A recent SD Times article related this story: “One organization outsourced its test case implementation with the goal of achieving 80% test coverage. The test coverage went up, but because the tests were calling methods but not validating them, the outcome was useless.” (Morgan, Lisa. “Driving Higher Value from Build Management and Continuous Integration,” SD Times, October 2011.) We’ve both personally witnessed this kind of dysfunction.

Usually a “wide swath” test can be reworked into a series of fast, isolated, repeatable, self-validating tests. In pathological cases, the complicated setup and poor environmental control may make salvage untenable.

When pairing with developers new to TDD, we’ll sometimes gut a method completely, reducing it to merely returning a null value. Our pair partner is either enlightened by reading the failing tests, or else amused as all the automated tests pass. Try this technique for yourself, and see what happens!

Tests that prove nothing have no benefit to your team. Removing them will make your coverage metrics honest and useful.

T for Timely

Should you write tests before writing the production code or after it’s already built?

You should always know what you’re trying to build before you build it. Tests written first specify the behavior that you’re about to build into the code. These “specifications by example” can help everyone more rapidly understand what the code’s doing, particularly down the road when no one remembers what it was supposed to do.

You also want to know when you’re done—that is, when you’ve successfully coded things correctly to that specification. Running tests constantly lets you know when you can integrate your changes into the source repository.

Writing tests before the code encourages you to think of the use of the code before you think of its implementation. The tests are the very first client of the code you’re building. A clean unit test should make it very clear what the client code will look like. Hopefully you’ll do things like improve your function names and design simpler parameter lists as a result.

Can’t you achieve all of these qualities if you write tests after the code, something we refer to as Test-After Development (TAD)? Theoretically, there’s no reason you can’t get the same benefits. But in our experience, it simply doesn’t happen.

TAD programmers want to verify their code, too, but they typically have little interest in treating the tests as “specifications by example.” For them, the usually singular goal of unit testing is to verify whether or not some aspects of the code work. These unit tests typically cover up to two-thirds of the code base (see bullseye, binstock, and stack overflow). Other forms of integration tests can increase effective coverage, but unit tests are the closest reliable documentation for the code you’re modifying.

The choice to cover only a fraction of the code base with unit tests is a judgment call, hopefully one that weighs effort against risk. Our take: we know that defects, sometimes devastating ones, creep up in the strangest places. We’d prefer to spend time creating confidence that all the logic in the system is solid instead of spending it in long debugging sessions.

The real sticking point with TAD is productivity. When you don’t think about how to craft code so it’s testable, you’ll often end up with code that is near-impossible to test. The test-writer has to either rework it or use a lot of finesse to avoid rework. This makes testing feel like a waste of time. No wonder so few programmers have fully embraced unit-testing! We find TAD to be simply less fun, less beneficial, less productive, and more frustrating than TDD.

Studies have shown that TDD incurs an additional, initial development cost of anywhere from 15% to 35% more time. Our experiences doing and coaching TDD back that up, but we’ll be more daring and say that things speed up over time. Our experience with TAD is that it takes more than 35% of development time for far less benefits, and that figure tends to get worse over time.

Coming back to write a unit test for the ReportMailer class, you find that the constructor throws an exception when called. A static call to MailDestination.getEndpoint, apparently made to verify that each MailDestination object passed has a valid endpoint, is the problem.

```
public ReportMailer(MailDestination[] destinations) {
this.destinations = destinations;
if (destinations.length == 0)
throw new RuntimeException("dests required");
for (int i = 0; i < destinations.length; i++)
if (MailDestination.getEndpoint(destinations[i]) == null)
throw new RuntimeException("invalid endpoint");
}
```

A bit of digging reveals that getEndpoint requires a live external API call. You consider making the method instance-side, so that the client calls it like this:

```
 if (destinations[i].getEndpoint() == null)
```

but then you discover that other clients use it statically, and you don't want to have to touch other code. You could introduce the Feathers' WELC pattern Extract and Override Call to fix this problem. This testability challenge would have been addressed already had we been test-driving from the start.

Granted, fixing this isolated example doesn't represent a huge amount of additional effort. Compound it with gobs of other similar code with bad dependency upon bad dependency and long convoluted methods, and it's enough to make most developers toss up their hands in disgust.

Effective Unit Tests

Whatever you call them and however you define them, the most important thing about unit tests is that they be useful and effective for your programming team. The FIRST mnemonic is a simple mechanism to guide you there.

Tim Ottinger is the originator and co-author of Agile in a Flash, a contributor to Clean Code, and a 30-year (plus) software developer. Tim is a senior consultant with Industrial Logic where he helps transform teams and organizations through education, process consulting, and technical practices coaching. He is an incessant blogger and incorrigible punster. He still writes code, and he likes it.

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash with Tim, he's written over 100 articles on software development and a couple books, Agile Java and Essential Java Style, and contributed to Uncle Bob's Clean Code. Jeff runs the consulting and training company Langr Software Solutions from Colorado Springs.

Send the authors your feedback or discuss the article in the magazine forum.