CHAPTER 5

FIRST Properties of Good Tests

Unit tests provide many significant benefits when crafted with care. But your tests also represent code you must write and maintain. You and your team can lose lots of time and sleep due to the following problems with your tests:

- Tests that make little sense to someone following them
- Tests that fail sporadically
- "Tests" that don't prove anything worthwhile
- Tests that require a long time to execute
- Tests that don't sufficiently cover the code
- Tests that couple too tightly to implementation, meaning that small changes break lots of tests all at once
- Convoluted tests that jump through numerous setup hoops

In this chapter you'll learn some key concepts and a few simple tactics that can help make your tests shine and ensure that they pay off more than they cost.

FIRST It Helps to Remember That Good Tests Are FIRST

You can avoid many of the pitfalls that unit testers often drop into by following the FIRST principles of unit testing:

- [F]ast
- [I]solated
- [R]epeatable
- [S]elf-validating
- [T]imely

The word *first* itself has significant meaning in the context of unit testing. Right now, you're probably writing your code first, then writing unit tests after the fact. But, perhaps surprisingly, you can get different and better

results if you write the unit tests before you write the corresponding code. A host of folks practice the discipline known as test-driven development (TDD). The sole delineation between plain ol' unit testing (POUT) and TDD is that in TDD, the tests come *first*. Check out Chapter 12, *Test-Driven Development*, on page 153 if you're intrigued.

Whether you write tests first or last, you'll go farther with them if you adhere to the FIRST principles.

[F]IRST: [F]ast!

The dividing line between fast and slow unit tests is somewhat arbitrary—as Justice Potter Stewart said, "I know it when I see it." Fast tests deal solely in code and take a few milliseconds at most to execute. Slow tests interact with code that must handle external evil necessities such as databases, files, and network calls. They take dozens, hundreds, or thousands of milliseconds.

On a typical Java system, you'll probably want a few thousand unit tests. If an average test takes 200 ms, you'll wait over eight minutes each time to run 2,500 unit tests. Eight minutes might not seem terrible, but you're not going to run an eight-minute set of tests too many times throughout your development day.

"So what?" Pat says. "I can run just the tests around the code I'm changing."

Dale laughs. "I remember the last time you merged in one of your changes. It took us hours to uncover your nasty little defect. What did you say? Oh yes: 'My code changes can't possibly break that code way over there.'"

Pat says, "Well, I'll just wait until I've built up a good pile of changes, then run all the tests. One or two times a day should be enough."

"The last time you piled up a bunch of changes, you spent an extra couple hours merging. It really pays off to merge more frequently and know that your changes work well with the rest of the system," says Dale.

As your system grows, your unit tests will take longer and longer to run. Eight minutes easily turns into fifteen or even thirty. Don't feel alone if this happens to you—it's a common quandary—but don't feel at all proud.

When your unit tests reach the point where it's painful to run them more than a couple times per day, you've tipped the scale in the wrong direction. The value of your suite of unit tests diminishes as their ability to provide continual, comprehensive, and fast feedback about the health of your system

also diminishes. When you allow your tests to fall out of favor, you and your team will question the investment you made to create them.

Keep your tests fast! You can do so by keeping your design clean: minimize the dependencies on code that executes slowly, first and foremost. If all your tests interact with code that ultimately always makes a database call, all your tests will be slow.

We're faced with writing tests around the responsesByQuestion() method, which returns a histogram that breaks down the number of true and false responses for each question:

iloveyouboss/16-branch-persistence/src/iloveyouboss/domain/StatCompiler.java

```
public class StatCompiler {
   private QuestionController controller = new QuestionController();

   public Map<String, Map<Boolean, AtomicInteger>> responsesByQuestion(
         List<BooleanAnswer> answers) {
      Map<Integer, Map<Boolean, AtomicInteger>> responses = new HashMap<>();
      answers.stream().forEach(answer -> incrementHistogram(responses, answer));
      return convertHistogramIdsToText(responses);
   }

   private Map<String, Map<Boolean, AtomicInteger>> convertHistogramIdsToText(
         Map<Integer, Map<Boolean, AtomicInteger>> responses) {
      Map<String, Map<Boolean, AtomicInteger>> textResponses = new HashMap<>();
      responses.keySet().stream().forEach(id ->
➤        textResponses.put(controller.find(id).getText(), responses.get(id)));
      return textResponses;
   }

   private void incrementHistogram(
         Map<Integer, Map<Boolean, AtomicInteger>> responses,
         BooleanAnswer answer) {
      Map<Boolean, AtomicInteger> histogram =
            getHistogram(responses, answer.getQuestionId());
      histogram.get(Boolean.valueOf(answer.getValue())).getAndIncrement();
   }

   private Map<Boolean, AtomicInteger> getHistogram(
         Map<Integer, Map<Boolean, AtomicInteger>> responses, int id) {
      Map<Boolean, AtomicInteger> histogram = null;
      if (responses.containsKey(id))
         histogram = responses.get(id);
      else {
         histogram = createNewHistogram();
         responses.put(id, histogram);
      }
      return histogram;
   }
```

```
   private Map<Boolean, AtomicInteger> createNewHistogram() {
      Map<Boolean, AtomicInteger> histogram;
      histogram = new HashMap<>();
      histogram.put(Boolean.FALSE, new AtomicInteger(0));
      histogram.put(Boolean.TRUE, new AtomicInteger(0));
      return histogram;
   }
}
```

The histogram is a map of Booleans to a count for each. The responses hash map pairs question IDs with a histogram for each. The incrementHistogram() method updates the histogram for a given answer. Finally, the convertHistogramIdsToText() method transforms the responses map to a map of question-text-to-histogram.

Unfortunately, convertHistogramIdsToText() presents a testing challenge. Its call to the QuestionController find() method represents an interaction with a slow persistent store. Not only will the test be slow, but it will also require that the underlying database be populated with appropriate question entities. Because of the distance between the database data and the expected data values in the test, the test will be hard to follow and brittle.

Rather than have the code query the controller for the questions, let's first retrieve the questions, then pass their text as an argument to responsesByQuestion().

First, create a questionText() method whose sole job is to create a map of question-ID-to-question-text for questions referenced by the answers:

iloveyouboss/16-branch-persistence-redesign/src/iloveyouboss/domain/StatCompiler.java

```
public Map<Integer,String> questionText(List<BooleanAnswer> answers) {
   Map<Integer,String> questions = new HashMap<>();
   answers.stream().forEach(answer -> {
      if (!questions.containsKey(answer.getQuestionId()))
         questions.put(answer.getQuestionId(),
            controller.find(answer.getQuestionId()).getText()); });
   return questions;
}
```

Change responsesByQuestion() to take on the question-ID-to-question-text map:

iloveyouboss/16-branch-persistence-redesign/src/iloveyouboss/domain/StatCompiler.java

```
public Map<String, Map<Boolean, AtomicInteger>> responsesByQuestion(
      List<BooleanAnswer> answers, Map<Integer,String> questions) {
   Map<Integer, Map<Boolean, AtomicInteger>> responses = new HashMap<>();
   answers.stream().forEach(answer -> incrementHistogram(responses, answer));
   return convertHistogramIdsToText(responses, questions);
}
```

responsesByQuestion() then passes the map over to convertHistogramIdsToText():

iloveyouboss/16-branch-persistence-redesign/src/iloveyouboss/domain/StatCompiler.java

```
private Map<String, Map<Boolean, AtomicInteger>> convertHistogramIdsToText(
      Map<Integer, Map<Boolean, AtomicInteger>> responses,
➤     Map<Integer,String> questions) {
   Map<String, Map<Boolean, AtomicInteger>> textResponses = new HashMap<>();
   responses.keySet().stream().forEach(id ->
➤     textResponses.put(questions.get(id), responses.get(id)));
   return textResponses;
}
```

The code in questionText() still depends on the slow persistent store, but it's a small fraction of the code we were trying to test. We'll figure out how to test that later. The code in convertHistogramIdsToText() now depends only on an in-memory hash map, not a lookup to a slow persistent store. We can now easily write a test around the good amount of code involved with responsesByQuestion():

iloveyouboss/16-branch-persistence-redesign/test/iloveyouboss/domain/StatCompilerTest.java

```
@Test
public void responsesByQuestionAnswersCountsByQuestionText() {
   StatCompiler stats = new StatCompiler();
   List<BooleanAnswer> answers = new ArrayList<>();
   answers.add(new BooleanAnswer(1, true));
   answers.add(new BooleanAnswer(1, true));
   answers.add(new BooleanAnswer(1, true));
   answers.add(new BooleanAnswer(1, false));
   answers.add(new BooleanAnswer(2, true));
   answers.add(new BooleanAnswer(2, true));
   Map<Integer,String> questions = new HashMap<>();
   questions.put(1, "Tuition reimbursement?");
   questions.put(2, "Relocation package?");

   Map<String, Map<Boolean,AtomicInteger>> responses =
         stats.responsesByQuestion(answers, questions);

   assertThat(responses.get("Tuition reimbursement?").
         get(Boolean.TRUE).get(), equalTo(3));
   assertThat(responses.get("Tuition reimbursement?").
         get(Boolean.FALSE).get(), equalTo(1));
   assertThat(responses.get("Relocation package?").
         get(Boolean.TRUE).get(), equalTo(2));
   assertThat(responses.get("Relocation package?").
         get(Boolean.FALSE).get(), equalTo(0));
}
```

The responsesByQuestionAnswersCountsByQuestionText test is indeed a fast test, just the way we like them. It covers a good amount of interesting logic in responsesByQuestion(), convertHistogramIdsToText(), and incrementHistogram(). We could write a

number of interesting tests against the combinations of logic in all three of these methods. By the time we're done, we could easily write a handful of tests. A handful of fast tests covering more logic will easily outperform a single test dependent on a database call.

Your unit tests will run faster, and you'll find them easier to write, if you seek to minimize the amount of code that ultimately depends on slow things. Minimizing such dependencies is also a goal of good design—again and again, as here, you'll find that unit testing gets easier the more you're willing to align your code with clean object-oriented (OO) design concepts.

We still want to test the logic in questionText(), which still depends on the controller. We'll learn a technique for how to accomplish that in Chapter 10, *Using Mock Objects*, on page 123, and we'll write the test for questionText() in *Testing Databases*, on page 180.

F[I]RST: [I]solate Your Tests

Good unit tests focus on a small chunk of code to verify. That's in line with our definition of *unit*. The more code that your test interacts with, directly or indirectly, the more things are likely to go awry.

The code you're testing might interact with other code that reads from a database. Data dependencies create a whole host of problems. Tests that must ultimately depend on a database require you to ensure that the database has the right data. If your data source is shared, you have to worry about external changes (maybe out of your control) breaking your tests. Don't forget that other developers are often running their tests at the same time! Simply interacting with an external store increases the likelihood that your test will fail for availability or accessibility reasons.

Good unit tests also don't depend on other unit tests (or test cases within the same test method). You might think you're speeding up your tests by carefully crafting their order so that several tests can reuse some of the same expensively constructed data. But you're simultaneously creating an evil chain of dependencies. When things go wrong—and they will—you'll spend piles of time figuring out which one thing buried in a long chain of prior events caused your test to fail.

You should be able to run any one test at any time, in any order.

It's easy to keep your tests focused and independent if each test concentrates only on a small amount of behavior. When you start to add a second assert

to a test, ask yourself, "Does this assertion help to verify a single behavior, or does it represent a behavior that I could describe with a new test name?"

The Single Responsibility Principle (SRP) of OO class design (see *SOLID Class-Design Principles*, on page 110) says that classes should be small and single-purpose. More specifically, the SRP says your classes should have only one reason to change.

The SRP provides a great guideline for your test methods, also. If one of your test methods can break for more than one reason, consider splitting it into separate tests. When a focused unit test breaks, it's usually obvious why.

Your tests want to each be like Switzerland! Keep 'em isolated and running like clockwork!

FI[R]ST: Good Tests Should Be [R]epeatable

Tests don't appear out of thin air—*you're* the one who gets to design them, which means they are entirely under your control. You have the power to devise a test's conditions, which also means that you don't need a crystal ball to know what the test outcome should be. Part of your job in test design, then, is to provide an assertion that specifies what the outcome should be *each and every time* the test is run.

A repeatable test is one that produces the same results each time you run it. To accomplish repeatable tests, you must *isolate* them from anything in the external environment not under your direct control.

Your system will inevitably need to interact with elements not under your control, however. Any need to deal with the current time, for example, means your test must somehow deal with a rogue element that will make it harder to write repeatable tests. You can use a *mock object* Chapter 10, *Using Mock Objects*, on page 123 as one way to isolate the rest of the code under test and keep it independent from the volatility of time.

In the iloveyouboss application, we want to verify that when new questions are added to a profile, they are saved with a creation timestamp. Timestamps are moving targets, making it a bit of a challenge to assert what the creation timestamp should be.

After we add the question to the profile in the test, we could immediately request the system time. Maybe we're not worried about milliseconds, so we could compare the persisted time to the test's time. Most of the time, this might even work...but it will likely fail the first time the persistence time is something like 17:34:05.999.

Tests that fail sporadically are nuisances. Sometimes, particularly when tests drive concurrently executing code, they expose a flaw in the system. But more often, tests that intermittently fail cry wolf. Someone has to spend the time and take a look: "Is that a real problem? Hmm. Oh, I see, some chucklehead has added a comment, /* this test may cry wolf from time to time. */" Don't be that chucklehead.

Back to our time challenge. If only we could stop time from moving! Well, we can't stop time, but we *can* fake it out. Or rather, we can fake out our code to think it's getting the real time, when it instead obtains the current time from a different source. In Java 8, we can create a java.time.Clock object that always returns a fixed time. From a test, pass this fake clock object to the code that needs to obtain the current time:

iloveyouboss/16-branch-persistence/test/iloveyouboss/controller/QuestionControllerTest.java

```
@Test
public void questionAnswersDateAdded() {
   Instant now = new Date().toInstant();
   controller.setClock(Clock.fixed(now, ZoneId.of("America/Denver")));
   int id = controller.addBooleanQuestion("text");

   Question question = controller.find(id);

   assertThat(question.getCreateTimestamp(), equalTo(now));
}
```

The first line of the preceding test creates an Instant instance and stores it in the now local variable. The second line creates a Clock object fixed to the now Instant—when asked for the time, it will always return the now instant—and *injects* it into the controller through a setter method. The test's assertion verifies that the question's creation timestamp is the same as now:

iloveyouboss/16-branch-persistence/src/iloveyouboss/controller/QuestionController.java

```
public class QuestionController {
   private Clock clock = Clock.systemUTC();
   // ...

   public int addBooleanQuestion(String text) {
      return persist(new BooleanQuestion(text));
   }

   void setClock(Clock clock) {
      this.clock = clock;
   }
   // ...

   private int persist(Persistable object) {
➤     object.setCreateTimestamp(clock.instant());
```

```
        executeInTransaction((em) -> em.persist(object));
        return object.getId();
    }
}
```

The persist() method obtains an instant from the injected clock instance and passes it along to the setCreateTimestamp() method on the Persistable. If no client code injects a Clock instance using setClock(), the clock defaults to the systemUTC clock as initialized at the field level.

Voila! The QuestionController doesn't know squat about the nature of the Clock, only that it answers the current Instant. The clock used by the test acts as a *test double*—a stand-in for the real thing. You'll read more about test doubles and the myriad ways to implement and take advantage of them in Chapter 10, *Using Mock Objects*, on page 123.

On occasion, you'll need to interact directly with an external environmental influence such as a database. You'll want to set up a private sandbox to avoid conflicts with other developers whose tests concurrently alter the database. That might mean a separate Oracle instance or perhaps a separate web server on a nonstandard port.

Without repeatability, you might be in for some surprises at the worst possible moments. What's worse, these sort of surprises are usually bogus—it's not really a bug, it's just a problem with the test. You can't afford to waste time chasing down phantom problems.

Each test should produce the same results every time.

FIR[S]T: [S]elf-Validating

"I've been writing tests for years," says Pat. "Every once in a while, I write a main() method that drives some of my code. It spews a bunch of output onto the console using System.out.println(). I take a look at each result and compare it with what I expect the right answer to be."

"That's great," says Dale, "Most of us have done that sometime in our career, but it doesn't hold up so well to larger numbers of tests. I remember having to add lots of comments in main() to explain what I was testing next."

"I split them into smaller methods when they get out of hand," says Pat.

"Hmm…," says Dale, sporting a bemused look. "I also recall having a problem remembering what the expected output should look like. Sometimes I'd have to peer intently at the screen to spot the problem in a sea of output."

Pat replies, "I just add more comments to tell me what the expected output should be. Once in a while, I print a Boolean that says whether or not results are as I expected."

"It seems like you're beginning to reinvent JUnit, but there's one important difference: your main() test driver requires you to always visually verify the output. JUnit does that work for you—tests either pass or fail."

"You know," says Pat, "Maybe I don't agree with all of the things you're pushing with unit testing, but I'm starting to think it'd be a great idea to rework some of my main() methods into proper JUnit tests."

It sounds like Pat could be on the way to being test-infected!

Tests aren't tests unless they assert that things went as expected. You write unit tests to save you time, not take more of your time. Manually verifying the results of tests is a time-consuming process that can also introduce more risk—it's easy to get dozy and gloss over important signs when you pore over the voluminous output that pseudotests can produce.

Not only must your tests be self-validating, but they must also be self-arranging. Make sure you don't do anything silly, such as designing a test to require manual arrange steps before you can run it. You must automate any setup your test requires. Remember, regardless: requiring external setup in order to a run a test violates the [I]solated part of FIRST.

Grow the theme of self-validating as much as you can. Your tests will run as part of a larger suite of unit tests for your system. You might run these tests manually on occasion—but you could take things one step further and automate the process of when and how the tests are run.

If you use Eclipse or IntelliJ IDEA, consider incorporating a tool like Infinitest. As you make changes to your system, Infinitest identifies and runs (in the background) any tests that are potentially impacted. With Infinitest, testing moves from being a proactive task to being a gating criterion, much like compilation, that prevents you from doing anything further until you've fixed a reported problem.

On an even larger scale, you can use a continuous integration (CI) tool such as Jenkins[1] or TeamCity.[2] A CI tool watches your source repository and kicks off a build/test process when it recognizes changes.

1. http://jenkins-ci.org/
2. https://www.jetbrains.com/teamcity/

The sky's the limit. As an ideal, imagine a system where you write tests for all changes you make. Whenever you integrate the code into your source repository, a build automatically kicks off and runs all the tests (unit and otherwise), indicating that your system is acceptably healthy. The build server takes that vote of confidence and goes one step further, deploying your change to production.

Pat snorts from the corner, "Yeah, sure."

Don't laugh, Pat. Many teams today have the confidence to embrace continuous delivery (CD) and have significantly reduced the overhead of taking a need from inception to deployed product.

FIRS[T]: [T]imely

You can write unit tests at virtually any time. You could dredge up code in any old portion of your system and start tacking on unit tests to it. But you're better off focusing on writing unit tests in a timely fashion.

Unit testing is a good habit. With most good habits that you've not yet completely ingrained, such as brushing your teeth, it's easy to procrastinate and make excuses why you can skip the practice "just this once." Your dentist might love your funding his or her practice, but you're going to hate the time it takes to scrape away the tartar that's built up.

Likewise, the more you defer probing at your code with unit tests, the more plaque buildup (cruft) and cavities (defects) you'll need to deal with. Also, once you check code into your source repository, chances are low that you'll find the time to come back and write tests.

Many test-infected dev teams have guidelines or strict rules around unit testing. Some use review processes or even automated tools to reject code without sufficient tests.

"We use pair programming and a bit of peer pressure in our team to ensure that programmers don't check in untested code," says Dale. "Frequent check-ins into our CI environment has helped our programmers ingrain the habit of writing timely unit tests. Our team loves how our tests help demonstrate the health of our system."

You'll want to establish similar rules that make sense for your team. Keeping atop good practices like unit testing requires continual vigilance.

The more you unit-test, the more you'll find that it pays to write smaller chunks of code before tackling a corresponding unit test. First, it'll be easier

to write the test, and second, the test will pay off immediately as you flesh out the rest of the behaviors in the surrounding code.

If you've evolved to short code-then-test cycles, consider mutating to the next step of test-then-code. Take a forward look to Chapter 12, *Test-Driven Development*, on page 153 to see how writing tests first can help.

Finally, tackling any old code to test can be a waste of time. If the code doesn't exhibit any defects or need to change in the near future (such as "now"—as in, you're about to change the code and want to ensure that you don't break anything), your effort will return little value on the investment. Direct your efforts to more troubled or dynamic spots in your system.

After

Writing unit tests requires a considerable investment in time. Although your tests can repay that investment, every test you write adds more code that you must maintain. Guard that investment by ensuring your that tests retain high quality. Use the FIRST acronym to remind you of the characteristics of quality tests.

The Right-BICEP, next, provides you with a mnemonic to decide what kinds of JUnit tests you should write.