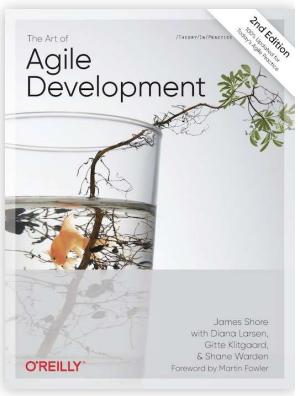
James Shore

The Art of Agile℠

The Art of Agile Development: Test-Driven Development

March 25, 2010

The second edition is now available! *The Art of Agile Development* has been completely revised and updated with all new material. Visit the Second Edition page for more information, or buy it on Amazon.



/v2/books/aoad2
https://www.amazon.com/Art-Agile-Development-James-Shore/dp/1492080691

Next: Refactoring /Agile-Book/refactoring.html
Previous: Customer Tests /Agile-Book/customer_tests.html
Up: Chapter 9: Developing /Agile-Book/developing_intro.html

Bonus Material

- Let's Play: Test-Driven Development is a comprehensive screencast showing a project developed in real-time using TDD. /Blog/Lets-Play/
- What Does a Good Test Suite Look Like? considers the characteristics of test suites. /Blog/What-Does-a-Good-Test-Suite-Look-Like.html

Full Text

Test-Driven Development

Audience
Programmers

We produce well-designed, well-tested, and well-factored code in small, verifiable steps.

"What programming languages really need is a 'DWIM' instruction," the joke goes. "Do what I mean, not what I say."

Programming is demanding. It requires perfection, consistently, for months and years of effort. At best, mistakes lead to code that won't compile. At worst, they lead to bugs that lie in wait and pounce at the moment that does the most damage.

People aren't so good at perfection. No wonder, then, that software is buggy.

Wouldn't it be cool if there was a tool that alerted you to programming mistakes moments after you made them—a tool so powerful that it virtually eliminated the need for debugging?

There is such a tool—or rather, a technique. It's test-driven development, and it actually delivers these results.

Test-driven development, or *TDD*, is a rapid cycle of testing, coding, and refactoring. When adding a feature, a pair may perform dozens of these cycles, implementing and refining the software in baby steps until there is nothing left to add and nothing left to take away. Research shows that TDD substantially reduces the incidence of defects [Janzen & Saiedian]. When used properly, it also helps improve your design, documents your public interfaces, and guards against future mistakes.

TDD isn't perfect, of course. (Is anything?) TDD is difficult to use on legacy codebases. Even with green-field systems, it takes a few months of steady use to overcome the learning curve. Try it anyway—although TDD benefits from other XP practices, it doesn't require them. You can use it on almost any project.

Why TDD Works

Back in the days of punchcards, programmers laboriously hand-checked their code to make sure it would compile. A compile error could lead to failed batch jobs and intense debugging sessions to look for the misplaced character.

Getting code to compile isn't such a big deal anymore. Most IDEs check your syntax as you type, and some even compile every time you save. The feedback loop is so fast that errors are easy to find and fix. If something doesn't compile, there isn't much code to check.

Test-driven development applies the same principle to programmer intent. Just as modern compilers provide more feedback on the syntax of your code, TDD cranks up the feedback on the execution of your code. Every few minutes—as often as every 20 or

30 *seconds*—TDD verifies that the code does what you think it should do. If something goes wrong, there are only a few lines of code to check. Mistakes are easy to find and fix.

TDD uses an approach similar to double-entry bookkeeping. You communicate your intentions twice, stating the same idea in different ways: first with a test, then with production code. When they match, it's likely they were both coded correctly. If they don't, there's a mistake somewhere.

> *Note*
> It's theoretically possible for the test and code to both be wrong in exactly the same way, thereby making it seem like everything's okay when it's not. In practice, unless you cut-and-paste between the test and production code, this is so rare it's not worth worrying about.

In TDD, the tests are written from the perspective of a class's public interface. They focus on the class' *behavior*, not its *implementation*. Programmers write each test before the corresponding production code. This focuses their attention on creating interfaces that are easy to use rather than easy to implement, which improves the design of the interface.

After TDD is finished, the tests remain. They're checked in with the rest of the code, and they act as living documentation of the code. More importantly, programmers run all of the tests with (nearly) every build, ensuring that code continues to work as originally intended. If someone accidentally changes the code's behavior—for example, with a misguided refactoring—the tests fail, signalling the mistake.

How to Use TDD

You can start using TDD today. It's one of those things that takes moments to learn and a lifetime to master.

> *Note*
> The basic steps of TDD are easy to learn, but the mindset takes a while to sink in. Until it does, TDD will likely seem clumsy, slow, and awkward. Give yourself two or three months of full-time TDD use to adjust.

Imagine TDD as a small, fast-spinning motor. It operates in a very short cycle that repeats over and over again. Every few minutes, this cycle ratchets your code forward a notch, providing code that—although it may not be finished—has been tested, designed, coded, and is ready to check in.

Every few minutes, TDD provides proven code that has been tested, designed, and coded.

To use TDD, follow the "red, green, refactor" cycle illustrated in Figure. With experience, unless you're doing a lot of refactoring, each cycle will take fewer than five minutes. Repeat the cycle until your work is finished. You can stop and integrate whenever all your tests pass, which should be every few minutes.

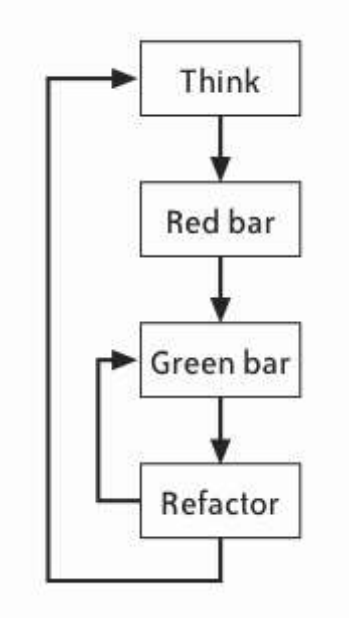


Figure. The TDD Cycle

Step 1: Think

TDD uses small tests to force you to write your code—you only write enough code to make the tests pass. The XP saying is, "Don't write any production code unless you have a failing test."

Your first step, therefore, is to engage in an a rather odd thought process. Imagine what behavior you want your code to have, then think of a small increment that will require fewer than five lines of code. Next, think of a test—also a few lines of code—that will fail unless that behavior is present.

In other words, think of a test that will force you to add the next few lines of production code. This is the hardest part of TDD because the concept of tests driving your code seems backwards, and because it can be difficult to think in small increments.

Pair programming helps. While the driver tries to make the current test pass, the navigator should stay a few steps ahead, thinking of tests that will drive the code to the next increment.

Ally
Pair Programming

Step 2: Red Bar

Now write the test. Write only enough code for the current increment of behavior—typically fewer than five lines of code. If it takes more, that's okay, just try for a smaller increment next time.

Code in terms of the class behavior and its public interface, not how you think you will implement the internals of the class. Respect encapsulation. In the first few tests, this often means that you write your test to use method and class names that don't exist yet. This is intentional—it forces you to design your class's interface from the perspective of a user of the class, not as its implementer.

After the test is coded, run your entire suite of tests and watch the new test fail. In most TDD testing tools, this will result in a red progress bar.

This is your first opportunity to compare your intent with what's actually happening. If the test *doesn't* fail, or if it fails in a different way than you expected, something is wrong. Perhaps your test is broken, or it doesn't test what you thought it did. Troubleshoot the problem; you should always be able to predict what's happening with the code.

> *Note*
> It's just as important to troubleshoot unexpected *successes* as it is to troubleshoot unexpected *failures*. Your goal isn't merely to have tests that work; it's to remain in control of your code—to always know what the code is doing and why.

Step 3: Green Bar

Next, write just enough production code to get the test to pass. Again, you should usually need less than five lines of code. Don't worry about design purity or conceptual elegance—just do what you need to do to make the test pass. Sometimes you can just hardcode the answer. This is okay because you'll be refactoring in a moment.

Run your tests again, and watch all the tests pass. This will result in a green progress bar.

When your tests fail and you can't figure out why, revert to known-good code.

This is your second opportunity to compare your intent with reality. If the test fails, get back to known-good code as quickly as you can. Often, you or your pairing partner can see the problem by taking a second look at the code you just wrote. If you can't see the problem, consider erasing the new code and trying again. Sometimes it's best to delete the new test (it's only a few lines of code, after all) and start the cycle over with a smaller increment.

> *Note*
> Remaining in control is key. It's okay to back up a few steps if that puts you back in control of your code. If you can't bring yourself to revert right away, set a five or ten-minute timer. Make a deal with your pairing partner that you'll revert to known-good code if you haven't solved the problem when the timer goes off.

Step 4: Refactor

With all your tests passing again, you can now refactor without worrying about breaking anything. Review the code and look for possible improvements. Ask your navigator if he's made any notes.

Ally
Refactoring

For each problem you see, refactor the code to fix it. Work in a series of very small refactorings—a minute or two each, certainly not longer than five minutes—and run the tests after each one. They should always pass. As before, if the test doesn't pass and the answer isn't immediately obvious, undo the refactoring and get back to known-good code.

Refactor as many times as you like. Make your design as good as you can, but limit it to the code's existing behavior. Don't anticipate future needs, and certainly don't add any

behavior. Remember, refactorings aren't supposed to change behavior. New behavior requires a failing test.

Step 5: Repeat

When you're ready to add new behavior, start the cycle over again.

Each time you finish the TDD cycle, you add a tiny bit of well-tested, well-designed code. The key to success with TDD is *small increments*. Typically, you'll run through several cycles very quickly, then spend more time on refactoring for a cycle or two, then speed up again.

> The key to TDD is small increments.

With practice, you can finish more than twenty cycles in an hour. Don't focus too much on how fast you go, though. That might tempt you to skip refactoring and design, which is too important to skip. Instead, take very small steps, run the tests frequently, and minimize the time you spend with a red bar.

A TDD Example

I recently recorded how I used TDD to solve a sample problem. The increments are very small—they may even seem ridiculously small—but this makes finding mistakes trivially easy, and that helps me go faster.

> *Note*
> Programmers new to TDD are often surprised at how small each increment can be. Although you might think that only beginners need to work in small steps, my experience is the reverse: the more TDD experience you have, the smaller steps you take and the faster you go.

As you read, keep in mind that it takes far longer to explain an example like this than to program it. I completed each step in a matter of seconds.

The Task

Imagine that you need to program a Java class to parse an HTTP query string.[1] You've decided to use TDD to do so.

[1] *Pretend you're in an alternate reality without a gazillion libraries that already do this.*

One name/value pair

Step 1: Think. The first step is to imagine the features you want the code to have. My first thought was, "I need my class to separate name/value pairs into a `HashMap`." Unfortunately, this would take more than five lines to code, so I needed to think of a smaller increment.

Often, the best way to make the increments smaller is to start with seemingly trivial cases. "I need my class to put *one* name/value pair into a `HashMap`," I thought, which

sounded like it would do the trick.

Step 2: Red Bar. The next step is to write the test. Remember, this is partially an exercise in interface design. In this example, my first temptation was to call the class `QueryStringParser`, but that's not very object-oriented. I settled on `QueryString`.

As I wrote the test, I realized that a class named `QueryString` wouldn't return a `HashMap`; it would *encapsulate* the `HashMap`. It would provide a method such as `valueFor(name)` to access the name-value pairs.

> *Note*
> Note that thinking about the test forced me to figure out how I wanted to design my code.

Building that seemed like it would require too much code for one increment, so I decided to have this test to drive the creation of a `count()` method instead. I decided that the `count()` method would return the total number of name/value pairs. My test checked that it would work when there was just one pair.

```
public void testOneNameValuePair() {
    QueryString qs = new QueryString("name=value");
    assertEquals(1, qs.count());
}
```

The code didn't compile, so I wrote a do-nothing `QueryString` class and `count()` method.

```
public class QueryString {
    public QueryString(String queryString) {}
    public int count() { return 0; }
}
```

That gave me the red bar I expected.

Green Bar. To make this test pass, I hardcoded the right answer. I could have programmed a better solution, but I wasn't sure how I wanted the code to work. Would it count the number of equal signs? Not all query strings have equal signs. I decided to punt.

```
public int count() { return 1; }
```

Green bar.

> *Note*
> Although I had ideas about how I would finish writing this class, I didn't commit myself to any particular course of action in advance. I remained open to discovering new approaches as I worked.

Step 4: Refactor. I didn't like the `QueryString` name, but I had another test in mind and I was eager to get to it. I made a note to fix the name on an index card—perhaps `HttpQuery` would be better. I'd see how I felt next time through the cycle.

Step 5: Repeat. Yup.

An empty string

Think. I wanted to force myself to get rid of that hard-coded `return 1`, but I didn't want to have to deal with multiple query strings yet. My next increment would probably be about the `valueFor()` method, and I wanted to avoid the complexity of multiple queries. I decided that testing an empty string would require me to code `count()` properly without making future increments too difficult.

Red Bar. New test.

```
public void testNoNameValuePairs() {
    QueryString qs = new QueryString("");
    assertEquals(0, qs.count());
}
```

Red bar. `Expected: <0> but was: <1>`. No surprise there.

This inspired two thoughts. First, I should test the case of a `null` argument to the `QueryString` constructor. Second, I was starting to see duplication in my tests that needed refactoring. I added both notes to my index card.

Green Bar. Now I had to stop and think. What was the fastest way for me to get back to a green bar? I decided to check if the query string was blank.

```
public class QueryString {
    private String _query

    public QueryString(string queryString) {
```

```
        _query = queryString;
    }

    public int count() {
        if ("".equals(_query)) return 0;
        else return 1;
    }
}
```

Refactor. I doublechecked my to-do list. I needed to refactor the tests, but I decided to wait for another test to demonstrate the need. "Three strikes and you refactor," as the saying goes.

It was time to do the cycle again.

testNull()

Think. My list included `testNull()`, which meant I needed to test the case when the query string is null. I decided to put that in.

Red Bar. This test forced me to think about what behavior I wanted when the value was null. I've always been a fan of code that fails fast, so I decided that a null value was illegal. This meant the code should throw an exception telling callers not to use null values. (Simple Design, later in this chapter, discusses failing fast in detail.)

/Agile-Book/simple_design.html

```
public void testNull() {
    try {
        QueryString qs = new QueryString(null);
        fail("Should throw exception");
    }
    catch (NullPointerException e) {
        // expected
    }
}
```

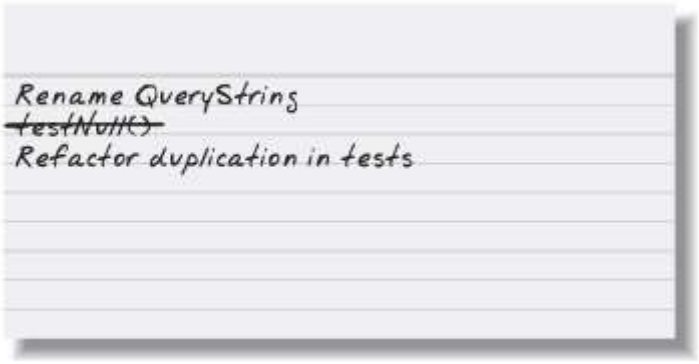
Green Bar. Piece of cake.

```
public QueryString(String queryString) {
    if (queryString == null) throw new NullPointerException();

    _query = queryString;
}
```

Refactor. I still needed to refactor my tests, but the new test didn't have enough in common with the old tests to make me feel it was necessary. The production code looked okay, too, and there wasn't anything significant on my index card. No refactorings this time.

> *Note*
> Although I don't refactor on every cycle, I always stop and seriously consider whether my design needs refactoring.

valueFor()



Think. Okay, now what? The easy tests were done. I decided to put some meat on the class and implement the `valueFor()` method. Given a name of a name/value pair in the query string, this method would return the associated value.

As I thought about this test, I realized I also needed a test to show what happens when the name doesn't exist. I wrote that on my index card for later.

Red Bar. To make the tests fail, I added a new assertion at the end of my existing `testOneNameValuePair()` test.

```
public void testOneNameValuePair() {
    QueryString qs = new QueryString("name=value");
    assertEquals(1, qs.count());
    assertEquals("value", qs.valueFor("name"));
}
```

Green Bar. This test made me think for a moment. Could the `split()` method work? I thought it would.

```
public String valueFor(String name) {
    String[] nameAndValue = _query.split("=");
    return nameAndValue[1];
}
```

This code passed the tests, but it was incomplete. What if there were more than one equal sign, or no equal signs? It needed proper error handling. I made a note to add tests for those scenarios.

Refactor. The names were bugging me. I decided that `QueryString` was okay, if not perfect. The `qs` in the tests was sloppy, so I renamed it `query`.

Multiple name/value pairs

Think. I had a note reminding me to take care of error handling in `valueFor()`, but I wanted to tackle something more meaty. I decided to add a test for multiple name/value pairs.

Red Bar. When dealing with a variable number of items, I usually test the case of zero items, one item, and three items. I had already tested zero and one, so now I tested three.

```
public void testMultipleNameValuePairs() {
    QueryString query = new QueryString("name1=value1&name2=value2&name3=value3");
    assertEquals("value1", query.valueFor("name1"));
    assertEquals("value2", query.valueFor("name2"));
    assertEquals("value3", query.valueFor("name3"));
}
```

I could have written an assertion for `count()` rather than `valueFor()`, but the real meat was in the `valueFor()` method. I made a note to test `count()` next.

Green Bar. My initial thought was that the `split()` technique would work again.

```
public String valueFor(String name) {
    String[] pairs = _query.split("&");
    for (String pair : pairs) {
        String[] nameAndValue = pair.split("=");
        if (nameAndValue[0].equals(name)) return nameAndValue[1];
    }
    throw new RuntimeException(name + " not found");
}
```

Ewww... that felt like a hack—but the test passed!

> *Note*
> It's better to get to a green bar quickly than to try for perfect code. A green bar keeps you in control of your code and allows you to experiment with refactorings that clean up your hack.

Refactor: The additions to `valueFor()` felt hackish. I took a second look. The two issues that bothered me the most were the `nameAndValue` array and the `RuntimeException`. An attempt to refactor `nameAndValue` led to worse code, so I backed out the refactoring and decided to leave it alone for another cycle.

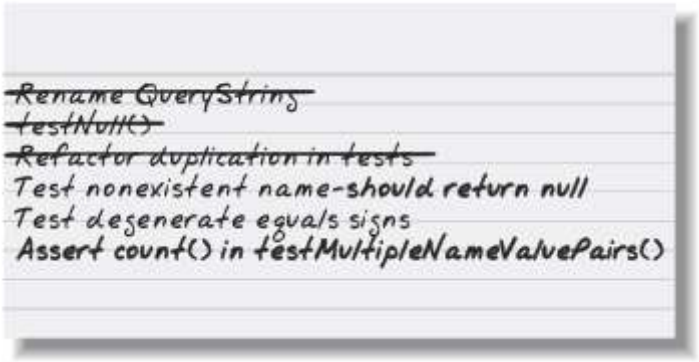
The `RuntimeException` was worse; it's better to throw a custom exception. In this case, though, the Java convention is to return `null` rather than throw an exception. I already

had a note that I should test the case where `name` isn't found; I revised it to say that the correct behavior was to return `null`.

Reviewing further, I saw that my duplicated test logic had reached three duplications. Time to refactor... or so I thought. After a second look, I realized that the only duplication between the tests was that I was constructing a `QueryString` object each time. Everything else was different, including `QueryString`'s constructor parameters. The tests didn't need refactoring after all. I scratched that note off of my list.

In fact, the code was looking pretty good... better than I initially thought, at least. I'm too hard on myself.

Multiple count()



Think. After reviewing my notes, I realized that I should probably test degenerate uses of the ampersand, such as two ampersands in a row. I made a note to add tests for that. At the time, though, I wanted to get the `count()` method working properly with multiple name/value pairs.

Red Bar. I added the `count()` assertion to my test. It failed as expected.

```
public void testMultipleNameValuePairs() {
    QueryString query = new QueryString("name1=value1&name2=value2&name3=value3");
    assertEquals(3, query.count());
    assertEquals("value1", query.valueFor("name1"));
    assertEquals("value2", query.valueFor("name2"));
    assertEquals("value3", query.valueFor("name3"));
}
```

Green Bar. To get this test to pass, I stole my existing code from `valueFor()` and modified it. This was blatant duplication, but I planned to refactor as soon as I saw the green bar.

```
public int count() {
    String[] pairs = _query.split("&");
    return pairs.length;
}
```

I was able to delete more of the copied code than I expected. To my surprise, however, it didn't pass! The test failed in the case of an empty query string: `expected: <0> but was: <1>`. I had forgotten that `split()` returned the original string when the split character isn't found. My code expected it to return an empty array when no split occurred.

I added a guard clause that took care of the problem. It felt like a hack, so I planned to take a closer look after the tests passed.

```
public int count() {
    if ("".equals(_query)) return 0;
    String[] pairs = _query.split("&");
    return pairs.length;
}
```

Refactor. This time I definitely needed to refactor. The duplication between `count()` and `valueFor()` wasn't too strong—it was just one line—but they both parsed the query string, which was a duplication of function if not code. I decided to fix it.

At first, I wasn't sure how to fix the problem. I decided to try to parse the query string into a `HashMap`, as I had originally considered. To keep the refactoring small, I left `count()` alone at first and just modified `valueFor()`. It was a small change.

```
public String valueFor(String name) {
    HashMap<String, String> map = new HashMap<String, String>();

    String[] pairs = _query.split("&");
    for (String pair : pairs) {
        String[] nameAndValue = pair.split("=");
        map.put(nameAndValue[0], nameAndValue[1]);
    }
    return map.get(name);
}
```

> *Note*
> This refactoring eliminated the exception that I threw when `name` wasn't found. Technically, it changed the behavior of the program. However, because I hadn't yet written a test for that behavior, I didn't care. I made sure I had a note to test that case later (I did) and kept going.

This code parsed the query string during every call to `valueFor()`, which wasn't a great idea. I had kept the code in `valueFor()` to keep the refactoring simple. Now I wanted to move it out of `valueFor()` into the constructor. This required a sequence of refactorings, described in Refactoring later in this chapter. /Agile-Book/refactoring.html

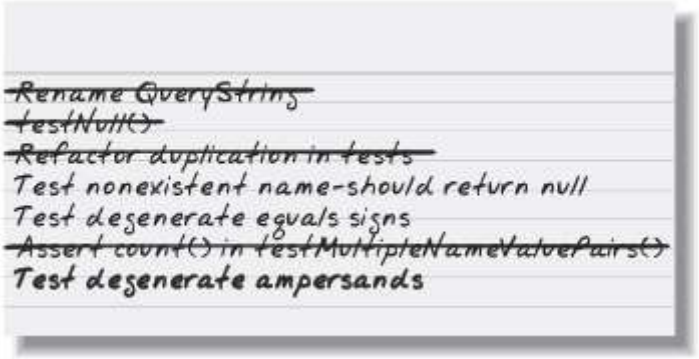
I reran the tests after each of these refactorings to make sure that I hadn't broken anything... and in fact, one refactoring did break the tests. When I called the parser from the constructor, `testNoNameValuePairs()`—the empty query test—bit me again, causing an exception in the parser. I added a guard clause as before, which solved the problem.

After all that refactoring, the tests and production code were nice and clean.

```
public class QueryStringTest extends TestCase {
    public void testOneNameValuePair() {
        QueryString query = new QueryString("name=value");
        assertEquals(1, query.count());
        assertEquals("value", query.valueFor("name"));
    }

    public void testMultipleNameValuePairs() {
        QueryString query = new QueryString("name1=value1&name2=value2&name3=value3");
        assertEquals(3, query.count());
        assertEquals("value1", query.valueFor("name1"));
        assertEquals("value2", query.valueFor("name2"));
        assertEquals("value3", query.valueFor("name3"));
    }

    public void testNoNameValuePairs() {
        QueryString query = new QueryString("");
        assertEquals(0, query.count());
```

```
    }

    public void testNull() {
        try {
            QueryString query = new QueryString(null);
            fail("Should throw exception");
        }
        catch (NullPointerException e) {
            // expected
        }
    }
}

public class QueryString {
    private HashMap<String, String> _map = new HashMap<String, String>();

    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
        parseQueryString(queryString);
    }

    public int count() {
        return _map.size();
    }

    public String valueFor(String name) {
        return _map.get(name);
    }

    private void parseQueryString(String query) {
        if ("".equals(query)) return;

        String[] pairs = query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
        }
    }
}
```

Your turn



The class wasn't done—it still needed to handle degenerate uses of the equals and ampersand signs, and it didn't fully implement the query string specification yet, either[2]. In the interest of space, though, I leave the remaining behavior as an exercise for you to complete yourself. As you try it, remember to take very small steps and to check for refactorings every time through the cycle.

[2]*For example, the semicolon works like the ampersand in query strings.*

Testing Tools

In order to use TDD, you need a testing framework. The most popular are the open-source *xUnit* tools, such as *JUnit* (for Java) and *NUnit* (for .NET). Although these tools have different authors, they typically share the basic philosophy of Kent Beck's pioneering SUnit.

> *Note*
> Instructions for specific tools are out of the scope of this book. Introductory guides for each tool are easily found online.

If your platform doesn't have an xUnit tool, you can build your own. Although the existing tools often provide GUIs and other fancy features, none of that is necessary. All you need is a way to run all of your test methods as a single suite, a few assert methods, and an unambiguous pass or fail result when the test suite is done.

Speed Matters

As with programming itself, TDD has myriad nuances to master. The good news is that the basic steps alone—red, green, refactor—will lead to very good results. Over time, you'll fine-tune your approach.

One of the nuances of TDD is test speed—not the frequency of each increment, which is also important, but how long it takes to run all of the tests. In TDD, you run the tests as often as one or two times every *minute*. They must be fast. If they aren't, they'll be a distraction rather than a help. You won't run them as frequently, which reduces the primary benefit of TDD: micro-increments of proof.

[Nielsen] reports that users lose interest and switch tasks when the computer makes them wait more than ten seconds. Computers only seem "fast" when they make users wait less than a second.

Make sure your tests take under ten seconds to run.

Although this research explored the area of user interface design, I've found it to be true when running tests as well. If they take more than ten seconds, I'm tempted to check my email, surf the web, or otherwise distract myself. Then it takes several minutes for me to get back to work. To avoid this, make sure your tests take under ten seconds to run. Less than a second is even better.

An easy way to keep your test times down is to run a subset of tests during the TDD cycle. Periodically run the whole test suite to make sure you haven't accidentally broken something, particularly before integrating and during refactorings that affect multiple classes.

Running a subset does incur the risk that you'll make a mistake without realizing it, leading to annoying debugging problems later. Advanced practitioners design their tests to run quickly. This requires that they make trade-offs between three basic types of automated tests:

- Unit tests, which run at a rate of hundreds per second
- Focused integration tests, which run at a rate of a handful per second
- End-to-end tests, which often require seconds per test

The vast majority of your tests should be unit tests. A small fraction should be integration tests, and only a handful should be end-to-end tests.

Unit Tests

Unit tests focus just on the class or method at hand. They run entirely in memory, which makes them very fast. Depending on your platform, your testing tool should be able to run at least 100 unit tests *per second*.

[Feathers] provides an excellent definition of a unit test:

> Unit tests run fast. If they don't run fast, they aren't unit tests.
>
> Other kinds of tests often masquerade as unit tests. A test is not a unit test if:
>
> 1. It talks to a database
> 2. It communicates across a network
> 3. It touches the file system
> 4. You have to do special things to your environment (such as editing configuration files) to run it

Tests that do these things are integration tests, not unit tests.

Creating unit tests requires good design. A highly-coupled system—a *big ball of mud*, or *spaghetti software*—makes it difficult to write unit tests. If you have trouble doing this, or if Feathers' definition seems impossibly idealistic, it's a sign of problems in your design. Look for ways to decouple your code so that each class, or set of related classes, may be tested in isolation. See Simple Design later in this chapter for ideas, and consider asking your mentor for help.

/Agile-Book/simple_design.html

Mock Objects

Mock objects are a popular tool for isolating classes for unit testing. When using mock objects, your test substitutes its own object (the "mock object") for an object that talks to the outside world. The mock object checks that it is called correctly and provides a pre-scripted response. In doing so, it avoids time-consuming communication to a database, network socket, or other outside entity.

Beware of mock objects. They add complexity and tie your test to the implementation of your code. When you're tempted to use a mock object, ask yourself if there's a way you could improve the design of your code so that a mock object isn't necessary. Can you decouple your code from the external dependency more cleanly? Can you provide the data it needs—in the constructor, perhaps—rather than having it go get the data itself?

Mock objects are a useful technique. Sometimes they're the best way to test your code. Before you assume that a mock object is appropriate for your situation, however, take a second look at your design. You might have an opportunity for improvement.

Focused Integration Tests

Unit tests aren't enough. At some point, your code has to talk to the outside world. You can use TDD for that code, too.

A test that causes your code to talk to a database, communicate across the network, touch the file system, or otherwise leave the bounds of its own process is an *integration test*. The best integration tests are *focused integration tests* that test just one interaction with the outside world.

Note
You may think of an integration test as a test that checks that the whole system fits together properly. I call that kind of test an *end-to-end test*.

One of the challenges of integration tests is the need to prepare the external dependency to be tested. Tests should run exactly the same way every time, regardless of which order you run them in or the state of the machine prior to running them. This is easy with unit tests but harder with integration tests. If you're testing your ability to select data from a database table, that data needs to be in the database.

Make sure each integration test can run entirely on its own. It should set up the environment it needs and then restore the previous environment afterwards. Be sure to do so even if the test fails or an exception is thrown. Nothing is more frustrating than a test that intermittently fails. Integration tests that don't set up and tear down their test environment properly are common culprits.

Make sure each test is isolated from the others.

Note
If you have a test that fails intermittently, don't ignore it, even if you can "fix" the failure by running the tests twice in a row. Intermittent failures are an example of technical debt. They make your tests more frustrating to run and disguise real failures.

You shouldn't need many integration tests. The best integration tests have a tight focus; each checks just one aspect of your program's ability to talk to the outside world. The number of focused integration tests in your test suite should be proportional to the types

of external interactions your program has, not the overall size of the program. (In contrast, the number of unit tests you have *is* proportional to the overall size of the program.)

If you need a lot of integration tests, it's a sign of design problems. It may mean that the code that talks to the outside world isn't cohesive. For example, if all your business objects talk directly to a database, you'll need integration tests for each one. A better design would be to have just one class that talks to the database. The business objects would talk to that class.[3] In this scenario, only the database class would need integration tests. The business objects could use ordinary unit tests.

[3] *A still better design might involve a persistence layer.*

End-to-End Tests

In a perfect world, the unit tests and focused integration tests mesh perfectly to give you total confidence in your tests and code. You should be able to make any changes you want without fear, comfortable in the knowledge that if you make a mistake, your tests will catch them.

How can you be sure that your unit tests and integration tests mesh perfectly? One way is to write end-to-end tests. End-to-end tests exercise large swaths of the system, starting at (or just behind) the user interface, passing through the business layer, touching the database, and returning. Acceptance tests and functional tests are common examples of end-to-end tests. Some people also call them integration tests, although I reserve that term for focused integration tests.

End-to-end tests can give you more confidence in your code, but they suffer from many problems. They're difficult to create because they require error-prone and labor-intensive setup and teardown procedures. They're brittle and tend to break whenever any part of the system or its setup data changes. They're very slow—they run in seconds or even minutes per test, rather than multiple tests per second. They provide a false sense of security, by exercising so many branches in the code that it's difficult to say which parts of the code are actually covered.

Instead of end-to-end tests, use exploratory testing to check the effectiveness of your unit and integration tests. When your exploratory tests find a problem, use that information to improve your approach to unit and integration testing, rather than introducing end-to-end tests.

Ally
Exploratory Testing

> *Note*
> Don't use exploratory testing to find bugs; use it to determine if your unit tests and integration tests mesh properly. When you find an issue, improve your TDD strategy.

In some cases, limitations in your design may prevent unit and integration tests from testing your code sufficiently. This is particularly true when you have legacy code. In that case, end-to-end tests are a necessary evil. Think of them as technical debt: strive to

make them unnecessary, and replace them with unit and integration tests whenever you have the opportunity.

TDD and Legacy Code

[Feathers] says *legacy code* is "code without tests." I think of it as "code you're afraid to change." This is usually the same thing.

The challenge of legacy code is that, because it was created without tests, it usually isn't designed for testability. In order to introduce tests, you need to change the code. In order to change the code with confidence, you need to introduce tests. It's this kind of chicken-and-egg problem that makes legacy code so difficult to work with.

To make matters worse, legacy code has often accumulated a lot of technical debt. (It's hard to remove technical debt when you're afraid to change existing code.) You may have trouble understanding how everything fits together. Methods and functions may have side effects that aren't apparent.

One way to approach this problem is to introduce end-to-end *smoke tests*. These tests exercise common usage scenarios involving the component you want to change. They aren't sufficient to give you total confidence in your changes, but they at least alert you when you make a big mistake.

With the smoke tests in place, you can start introducing unit tests. The challenge here is finding isolated components to test, as legacy code is often tightly coupled code. Instead, look for ways for your test to strategically interrupt program execution. [Feathers] calls these opportunities *seams*. For example, in an object-oriented language, if a method has a dependency you want to avoid, your test can call a test-specific subclass that overrides and stubs out the offending method.

Finding and exploiting seams often leads to ugly code. It's a case of temporarily making the code worse so you can then make it better. Once you've introduced tests, refactor the code to make it test-friendly, then improve your tests so they aren't so ugly. Then you can proceed with normal TDD.

Adding tests to legacy code is a complex subject that deserves its own book. Fortunately, [Feathers]' *Working Effectively with Legacy Code* is exactly that book.

Questions

What do I need to test when using TDD?

The saying is, "Test everything that can possibly break." To determine if something could possibly break, I think, "Do I have absolute confidence that I'm doing this correctly, and that nobody in the future will inadvertently break this code?"

I've learned through painful experience that I can break nearly anything, so I test nearly everything. The only exception is code without any logic, such as simple accessors and mutators (getters and setters), or a method that only calls another method.

You *don't* need to test third-party code unless you have some reason to distrust it.

How do I test private methods?

As in my extended `QueryString` example, start by testing public methods. As you refactor, some of that code will move into private methods, but the existing tests will still thoroughly test its behavior.

If your code is so complex that you need to test a private method directly, this may be a sign to refactor. You may benefit from moving the private methods into their own class and providing a public interface. The "Replace Method with Method Object" refactoring [Fowler 1999] (p. 135) may help.

How can I use TDD when developing a user interface?

TDD is particularly difficult with user interfaces because most UI frameworks weren't designed with testability in mind. Many people compromise by writing a very thin, untested translation layer that only forwards UI calls to a presentation layer. They keep all of their UI logic in the presentation layer and use TDD on that layer as normal.

There are some tools that allow you to test a UI directly, perhaps by making HTTP calls (for web-based software), or by pressing buttons or simulating window events (for client-based software). These are essentially integration tests and they suffer similar speed and maintainability challenges as other integration tests. Despite the challenges, these tools can be helpful.

You talked about refactoring your test code. Does anyone really do this?

Yes. I do, and everybody should. Tests are just code. The normal rules of good development apply: avoid duplication, choose good names, factor and design well.

I've seen otherwise-fine projects go off of the rails because of brittle and fragile test suites. By making TDD a central facet of development, you've committed to maintaining your test code just as much as you've committed to maintaining the rest of the code. Take it just as seriously.

Results

When you use TDD properly, you find that you spend little time debugging. Although you continue to make mistakes, you find those mistakes very quickly and have little difficulty fixing them. You have total confidence that the whole codebase is well-tested, which allows you to aggressively refactor at every opportunity, confident in the knowledge that the tests will catch any mistakes.

Contraindications

Although TDD is a very valuable tool, it does have a two-or-three month learning curve. It's easy to apply to toy problems such as the `QueryString` example, but translating that experience to larger systems takes time. Legacy code, proper unit test isolation, and integration tests are particularly difficult to master. On the other hand, the sooner you start using TDD, the sooner you'll figure it out, so don't let these challenges stop you.

Be careful when applying TDD without permission. Learning TDD could slow you down temporarily. This could backfire and cause your organization to reject TDD without proper consideration. I've found that combining testing time with development time when providing estimates helps alleviate pushback for dedicated developer testing.

Also be cautious about being the only one to use TDD on your team. You may find that your teammates break your tests and don't fix them. It's better to get the whole team to agree to try it together.

Alternatives

TDD is the heart of XP's programming practices. Without it, all of XP's other technical practices will be much harder to use.

A common misinterpretation of TDD is to design your entire class first, then write all of its test methods, then write the code. This approach is frustrating and slow, and it doesn't allow you to learn as you go.

Another misguided approach is to write your tests after you write your production code. This is very difficult to do well—production code must be designed for testability, and it's hard to do so unless you write the tests first. It doesn't help that writing tests after the fact is boring. In practice, the temptation to move on to the next task usually overwhelms the desire for well-tested code.

Although you can use these alternatives to introduce tests to your code, TDD isn't just about testing. It's really about using very small increments to produce high-quality, known-good code. I'm not aware of any alternatives that provide TDD's ability to catch and fix mistakes quickly.

Further Reading

Test-driven development is one of the most heavily-explored aspects of Extreme Programming. There are several excellent books on various aspects of TDD. Most are focused on Java and JUnit, but their ideas are applicable to other languages as well.

Test-Driven Development: By Example [Beck 2002] is a good introduction to TDD. If you liked the `QueryString` example, you'll like the extended examples in this book. The TDD

patterns in Part III are particularly good.

Test-Driven Development: A Practical Guide [Astels] provides a larger example that covers a complete project. Reviewers praise its inclusion of UI testing.

JUnit Recipes [Rainsberger] is a comprehensive book discussing a wide variety of testing problems, including a thorough discussion of testing J2EE.

Working Effectively with Legacy Code [Feathers] is a must-have for anybody working with legacy code.

Next: Refactoring /Agile-Book/refactoring.html
Previous: Customer Tests /Agile-Book/customer_tests.html
Up: Chapter 9: Developing /Agile-Book/developing_intro.html