

10

Working with legacy code

This chapter covers

- Examining common problems with legacy code
- Deciding where to begin writing tests
- Surveying helpful tools for working with legacy code

I once consulted for a large development shop that produced billing software. They had over 10,000 developers and mixed .NET, Java, and C++ in products, sub-products, and intertwined projects. The software had existed in one form or another for over five years, and most of the developers were tasked with maintaining and building on top of existing functionality.

My job was to help several divisions (using all languages) learn TDD techniques. For about 90% of the developers I worked with, this never became a reality for several reasons, some of which were a result of legacy code:

- It was difficult to write tests against existing code.
- It was next to impossible to refactor the existing code (or there wasn't enough time to do it).
- Some people didn't want to change their designs.

- Tooling (or lack of tooling) was getting in the way.
- It was difficult to determine where to begin.

Anyone who's ever tried to add tests to an existing system knows that most such systems are almost impossible to write tests for. They were usually written without proper places in the software (seams) to allow extensions or replacements to existing components.

There are several problems that need to be addressed when dealing with legacy code:

- There's so much work, where should you start to add tests? Where should you focus your efforts?
- How can you safely refactor your code if it has no tests to begin with?
- What tools can you use with legacy code?

This chapter will tackle these tough questions associated with approaching legacy code bases by listing techniques, references, and tools that can help.

10.1 Where do you start adding tests?

Assuming you have existing code inside components, you'll need to create a priority list of components for which testing makes the most sense. There are several factors to consider that can affect each component's priority:

- *Logical complexity*—This refers to the amount of logic in the component, such as nested `ifs`, `switch` cases, or recursion. Tools for checking cyclomatic complexity can also be used to determine this.
- *Dependency level*—This refers to the number of dependencies in the component. How many dependencies do you have to break in order to bring this class under test? Does it communicate with an outside email component, perhaps, or does it call a static log method somewhere?
- *Priority*—This is the component's general priority in the project.

You can give each component a rating for these factors, from 1 (low priority) to 10 (high priority).

Table 10.1 shows classes with ratings for these factors. I call this a *test-feasibility table*.

Table 10.1 A simple test-feasibility table

Component	Logical complexity	Dependency level	Priority	Notes
Utils	6	1	5	This utility class has few dependencies but contains a lot of logic. It will be easy to test, and it provides lots of value.
Person	2	1	1	This is a data-holder class with little logic and no dependencies. There's some (small) real value in testing this.

Table 10.1 A simple test-feasibility table (continued)

Component	Logical complexity	Dependency level	Priority	Notes
TextParser	8	4	6	This class has lots of logic and lots of dependencies. To top it off, it's part of a high-priority task in the project. Testing this will provide lots of value but will also be hard and time consuming.
ConfigManager	1	6	1	This class holds configuration data and reads files from disk. It has little logic but many dependencies. Testing it will provide little value to the project and will also be hard and time consuming.

From the data in table 10.1, you can create the diagram shown in figure 10.1, which graphs your components by the amount of value to the project and number of dependencies.

You can safely ignore items that are below your designated threshold of logic (which I usually set at 2 or 3), so *Person* and *ConfigManager* can be ignored. You're left with only the top two components from figure 10.1.

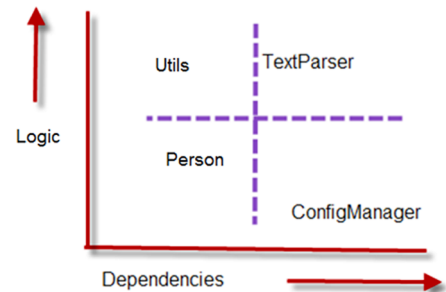
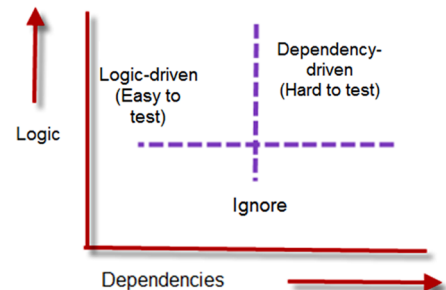
There are two basic ways to look at the graph and decide what you'd like to test first (see figure 10.2):

- Choose the one that's more complex and easier to test (top left).
- Choose the one that's more complex and harder to test (top right).

The question now is what path you should take. Should you start with the easy stuff or the hard stuff?

10.2 Choosing a selection strategy

As the previous section explained, you can start with the components that are easy to test or the ones that are hard to test (because they have many dependencies). Each strategy presents different challenges.

**Figure 10.1 Mapping components for test feasibility****Figure 10.2 Easy, hard, and irrelevant component mapping based on logic and dependencies**

10.2.1 Pros and cons of the easy-first strategy

Starting out with the components that have fewer dependencies will make writing the tests initially much quicker and easier. But there's a catch, as figure 10.3 demonstrates.

Figure 10.3 shows how long it takes to bring components under test during the lifetime of the project. Initially it's easy to write tests, but as time goes by, you're left with components that are increasingly harder and harder to test, with the particularly tough ones waiting for you at the end of the project cycle, just when everyone is stressed about pushing a product out the door.

If your team is relatively new to unit testing techniques, it's worth starting with the easy components. As time goes by, the team will learn the techniques needed to deal with the more complex components and dependencies.

For such a team, it may be wise to initially avoid all components over a specific number of dependencies (with four being a reasonable limit).

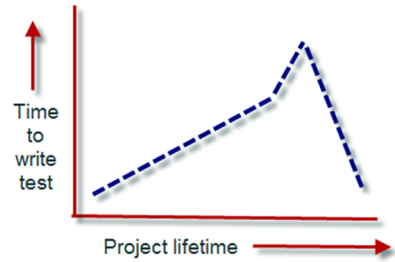


Figure 10.3 When starting with the easy components, the time required to test components increases more and more until the hardest components are done.

10.2.2 Pros and cons of the hard-first strategy

Starting with the more difficult components may seem like a losing proposition to begin with, but it has an upside, as long as your team has experience with unit testing techniques.

Figure 10.4 shows the average time to write a test for a single component over the lifetime of the project, if you start testing the components with the most dependencies first.

With this strategy, you could be spending a day or more to get even the simplest tests going on the more complex components. But notice the quick decline in the time required to write the test relative to the slow incline in figure 10.3. Every time you bring a component under test and refactor it to make it more testable, you may also be solving testability issues for the dependencies it uses or for other components. Specifically because that component has lots of dependencies, refactoring it can improve things for other parts of the system. That's the reason for the quick decline.

The hard-first strategy is only possible if your team has experience in unit testing techniques, because it's harder to implement. If your team does have experience, use the priority aspect of components to choose whether to start with the hard or

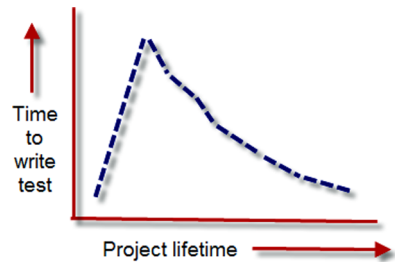


Figure 10.4 When you use a hard-first strategy, the time required to test components is initially high, but then decreases as more dependencies are refactored away.

easy components first. You might want to choose a mix, but it's important that you know in advance how much effort will be involved and what the possible consequences are.

10.3 Writing integration tests before refactoring

If you do plan to refactor your code for testability (so you can write unit tests), a practical way to make sure you don't break anything during the refactoring phase is to write integration-style tests against your production system.

I consulted on a large legacy project, working with a developer who needed to work on an XML configuration manager. The project had no tests and was hardly testable. It was also a C++ project, so we couldn't use a tool like Typemock Isolator to isolate components without refactoring the code.

The developer needed to add another value attribute into the XML file and be able to read and change it through the existing configuration component. We ended up writing a couple of integration tests that used the real system to save and load configuration data and that asserted on the values the configuration component was retrieving and writing to the file. Those tests set the "original" working behavior of the configuration manager as our base of work.

Next, we wrote an integration test that showed that once the component was reading the file, it contained no attribute in memory with the name we were trying to add. We proved that the feature was missing, and we now had a test that would pass once we added the new attribute to the XML file and correctly wrote to it from the component.

Once we wrote the code that saved and loaded the extra attribute, we ran the three integration tests (two tests for the original base implementation and a new one that tried to read the new attribute). All three passed, so we knew that we hadn't broken existing functionality while adding the new functionality.

As you can see, the process is relatively simple:

- Add one or more integration tests (no mocks or stubs) to the system to prove the original system works as needed.
- Refactor or add a failing test for the feature you're trying to add to the system.
- Refactor and change the system in small chunks, and run the integration tests as often as you can, to see if you break something.

Sometimes, integration tests may seem easier to write than unit tests, because you don't need to mess with DI. But making those tests run on your local system may prove annoying or time consuming because you have to make sure every little thing the system needs is in place.

The trick is to work on the parts of the system that you need to fix or add features to. Don't focus on the other parts. That way, the system grows in the right places, leaving other bridges to be crossed when you get to them.

As you continue adding more and more tests, you can refactor the system and add more unit tests to it, growing it into a more maintainable and testable system. This takes time (sometimes months and months), but it's worth it.

Did I mention that you need to have good tools? Let's look at some of my favorites.

10.4 Important tools for legacy code unit testing

Here are a few tips on tools that can give you a head start if you're doing any testing on existing code in .NET:

- Isolate dependencies easily with JustMock or Typemock Isolator.
- Use JMockit for Java legacy code.
- Use Vise while refactoring your Java code.
- Use FitNesse for acceptance tests before you refactor.
- Read Michael Feathers's book on legacy code.
- Use NDepend to investigate your production code.
- Use ReSharper to navigate and refactor your production code more easily.
- Detect duplicate code (and bugs) with Simian and TeamCity.

Let's look at each of these in more detail.

10.4.1 Isolate dependencies easily with unconstrained isolation frameworks

Unconstrained frameworks such as Typemock Isolator were introduced in chapter 6. What makes such frameworks uniquely suited for this challenge is their ability to fake dependencies in production code without needing to refactor it at all, saving valuable time in bringing a component under test, initially.

NOTE Full disclosure: while writing the first edition of this book, I also worked as a developer at Typemock on a different product. I also helped design the API in Isolator 5.0. I stopped working at Typemock in December 2010.

Why Typemock and not Microsoft Fakes?

Although Microsoft Fakes is free, and Isolator and JustMock are not, I believe using Microsoft Fakes will create a very big batch of unmaintainable test code in your project, because its design and usage (code generation, and delegates all over the place) lead to a very fragile API that's hard to maintain. This problem is even mentioned in an ALM Rangers document about Microsoft Fakes, which can be found at <http://vsartesttoolingguide.codeplex.com/releases/view/102290>. There, it states that "if you refactor your code under test, the unit tests you have written using Shims and Stubs from previously generated Fakes assemblies will no longer compile. At this time, there is no easy solution to this problem other than perhaps using a set of bespoke regular expressions to update your unit tests. Keep this in mind when estimating any refactoring to code that has been extensively unit tested. It may prove a significant cost."

I'm going to use Typemock Isolator for the next examples, because it's the framework I feel most comfortable with. Isolator (7.0 at the time of writing this book) uses the term *fake* and removes the words *mock* and *stub* from the API. Using this framework, you can "fake" interfaces, sealed and static types, nonvirtual methods, and static methods. This means you don't need to worry about changing the design (which you may not have time for, or perhaps can't for security reasons). You can start testing almost immediately. There's also a free, constrained version of Typemock, so you can download this product and try it on your own. Just know that by default it's constrained, so it will work only on standard testable code.

The listing that follows shows a couple of examples of using the Isolator API to fake instances of classes.

Listing 10.1 Faking static methods and creating fake classes with Isolator

```
[Test]
public void FakeAStaticMethod()
{
    Isolate
        .WhenCalled(() => MyClass.SomeStaticMethod())
        .WillThrowException(new Exception());
}

[Test]
public void FakeAPrivateMethodOnAClassWithAPrivateConstructor()
{
    ClassWithPrivateConstructor c =
        Isolate.Fake.Instance<ClassWithPrivateConstructor>();
    Isolate.NonPublic
        .WhenCalled(c, "SomePrivateMethod").WillReturn(3);
}
```

As you can see, the API is simple and clear, and it uses generics and delegates to return fake values. There's also an API specifically dedicated for VB.NET that has a more VB-centric syntax. In both APIs, you don't need to change anything in the design of your classes under test to make these tests work.

10.4.2 Use JMockit for Java legacy code

JMockit or PowerMock is an open source project that uses the Java instrumentation APIs to do some of the same things that Typemock Isolator does in .NET. You don't need to change the design of your existing project to isolate your components from their dependencies.

JMockit uses a *swap* approach. First, you create a manually coded class that will replace the class that acts as a dependency to your component under test (say you code a FakeDatabase class to replace a Database class). Then you use JMockit to swap calls from the original class to your own fake class. You can also redefine a class's methods by defining them again as anonymous methods inside the test.

The next listing shows a sample of a test that uses JMockit.

Listing 10.2 Using JMockit to swap class implementations

```

public class ServiceATest extends TestCase    {
    private boolean serviceMethodCalled;

    public static class MockDatabase          {
        static int findMethodCallCount;
        static int saveMethodCallCount;

        public static void save(Object o)    {
            assertNotNull(o);
            saveMethodCallCount++;
        }

        public static List find(String ql, Object arg1) {
            assertNotNull(ql);
            assertNotNull(arg1);
            findMethodCallCount++;
            return Collections.EMPTY_LIST;
        }
    }

    protected void setUp() throws Exception {
        super.setUp();
        MockDatabase.findMethodCallCount = 0;
        MockDatabase.saveMethodCallCount = 0;
        Mockit.redefineMethods(Database.class,
                                MockDatabase.class);
    }

    public void testDoBusinessOperationXyz() throws Exception {
        final BigDecimal total = new BigDecimal("125.40");

        Mockit.redefineMethods(ServiceB.class,
                                new Object())

    {
        public BigDecimal computeTotal(List items)
        {
            assertNotNull(items);
            serviceMethodCalled = true;
            return total;
        }
    });

    EntityX data = new EntityX(5, "abc", "5453-1");
    new ServiceA().doBusinessOperationXyz(data);

    assertEquals(total, data.getTotal());
    assertTrue(serviceMethodCalled);
    assertEquals(1, MockDatabase.findMethodCallCount);
    assertEquals(1, MockDatabase.saveMethodCallCount);
}
}

```

**The magic
happens
here**

JMockit is a good place to start when testing Java legacy code.

10.4.3 Use Vise while refactoring your Java code

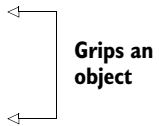
Michael Feathers wrote an interesting tool for Java that allows you to verify that you aren't messing up the values that may change in your method while refactoring it. For example, if your method changes an array of values, you want to make sure that as you refactor you don't screw up a value in the array.

The following listing shows an example of using the `Vise.grip()` method for such a purpose.

Listing 10.3 Using Vise in Java code to verify values aren't changed while refactoring

```
import vise.tool.*;

public class RPRequest {
    ...
    public int process(int level, RPPacket packet) {
        if (...) {
            if (...) {
                ...
            } else {
                ...
                bar_args[1] += list.size();
                Vise.grip(bar_args[1]);
                packet.add(new Subpacket(list, arrivalTime));
                if (packet.calcSize() > 2)
                    bar_args[1] += 2;
                Vise.grip(bar_args[1]);
            }
        } else {
            int reqLine = -1;
            bar_args[0] = packet.calcSize(reqLine);
            Vise.grip(bar_args[0]);
            ...
        }
    }
}
```



NOTE The code in listing 10.3 is copied with permission from www.artima.com/weblogs/viewpost.jsp?thread=171323.

Vise forces you to add lines to your production code, and it's there to support refactoring of the code. There's no such tool for .NET, but it should be pretty easy to write one. Every time you call the `Vise.grip()` method, it checks whether the value of the passed-in variable is still what it's supposed to be. It's like adding an internal assert to your code, with a simple syntax. Vise can also report on all "gripped" items and their current values.

You can read about and download Vise free from Michael Feathers's blog: www.artima.com/weblogs/viewpost.jsp?thread=171323.

10.4.4 Use acceptance tests before you refactor

It's a good idea to add integration tests to your code before you start refactoring it. FitNesse is one tool that helps create a suite of integration- and acceptance-style tests. Another one you might want to look into is Cucumber or SpecFlow. (You might need to know some Ruby to work with Cucumber. SpecFlow is native to .NET and is built to parse Cucumber scenarios.) FitNesse allows you to write integration-style tests (in Java or .NET) against your application, and then change or add to them easily without needing to write code.

Using the FitNesse framework involves three steps:

- 1 Create code adapter classes (called *fixtures*) that can wrap your production code and represent actions that a user might take against it. For example, if it were a banking application, you might have a `BankingAdapter` class that has `withdraw` and `deposit` methods.
- 2 Create HTML tables using a special syntax that the FitNesse engine recognizes and parses. These tables will hold the values that will be run during the tests. You write these tables in pages in a specialized wiki website that runs the FitNesse engine underneath, so that your test suite is represented to the outside world by a specialized website. Each page with a table (which you can see in any web browser) is editable like a regular wiki page, and each has a special `Execute Tests` button. These tables are then parsed by the testing runtime and translated into test runs.
- 3 Click the `Execute Tests` button on one of the wiki pages. That button invokes the FitNesse engine with the parameters in the table. Eventually, the engine calls your specialized wrapper classes that invoke the target application and asserts on return values from your wrapper classes.

Figure 10.5 shows an example FitNesse table in a browser. You can learn more about FitNesse at <http://fitnesse.org/>. For .NET integration with FitNesse, go to <http://fitnesse.org/FitNesse.DotNet>.

Personally, I've almost always found FitNesse a big bother to work with—the usability suffers a lot and it doesn't work half the time, especially with .NET stuff. Cucumber might be worth looking into instead. It's found at <http://cukes.info/>.

10.4.5 Read Michael Feathers's book on legacy code

Working Effectively with Legacy Code, by Michael Feathers, is the only source I know that deals with the issues you'll encounter with legacy code (other than this chapter). It shows many refactoring techniques and gotchas in depth that this book doesn't attempt to cover. It's worth its weight in gold. Get it.

10.4.6 Use NDepend to investigate your production code

NDepend is a relatively new commercial analyzer tool for .NET that can create visual representations of many aspects of your compiled assemblies, such as dependency

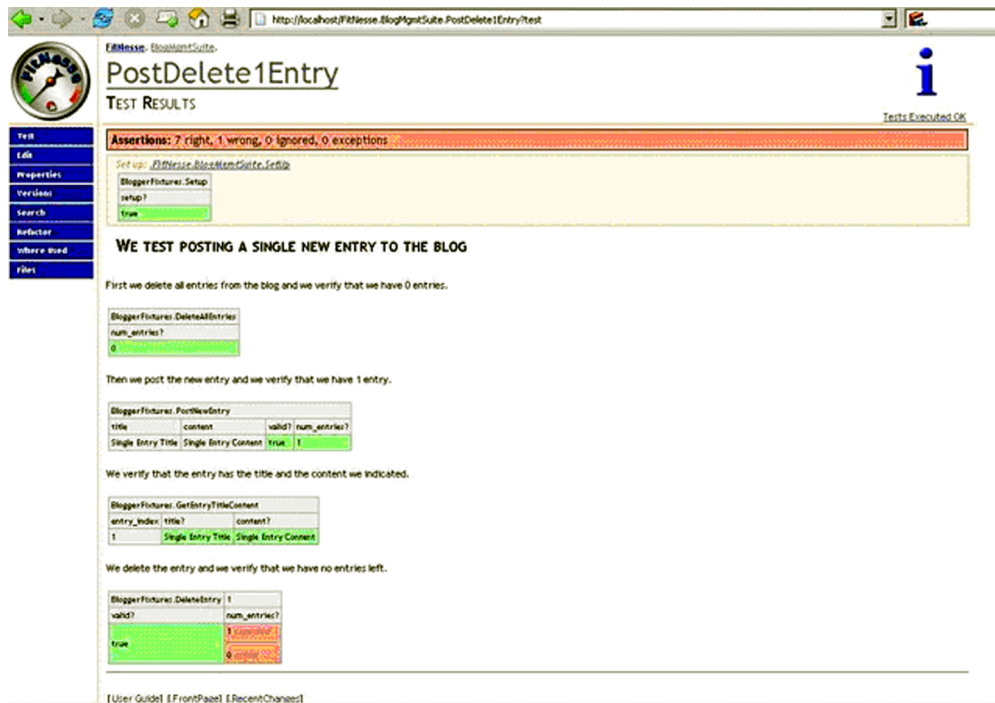


Figure 10.5 Using FitNesse for integration testing

trees, code complexity, changes between the versions of the same assembly, and more. The potential of this tool is huge, and I recommend you learn how to use it.

NDepend's most powerful feature is a special query language (called CQL) you can use against the structure of your code to find out various component metrics. For example, you could easily create a query that reports on all components that have a private constructor.

You can get NDepend from www.ndepend.com.

10.4.7 Use ReSharper to navigate and refactor production code

ReSharper is one of the best productivity-related plug-ins for VS.NET. In addition to powerful automated refactoring abilities (much more powerful than the ones built into Visual Studio 2008), it's known for its navigation features. When jumping into an existing project, ReSharper can easily navigate the code base with shortcuts that allow you to jump from any point in the solution to any other point that might be related to it.

Here are examples of possible navigations:

- When in a class or method declaration, you can jump to any inheritors of that class or method or jump up to the base implementation of the current member or class, if one exists.

- You can find all uses of a given variable (highlighted in the current editor).
- You can find all uses of a common interface or a class that implements it.

These and many other shortcuts make it much less painful to navigate and understand the structure of existing code.

ReSharper works on both VB.NET and C# code. You can download a trial version at www.jetbrains.com.

10.4.8 Detect duplicate code (and bugs) with Simian and TeamCity

Let's say you found a bug in your code, and you want to make sure that bug was not duplicated somewhere else.

TeamCity contains a built-in duplicates finder for .NET. Find more information on the TeamCity Duplicates finder at [http://confluence.jetbrains.com/display/TCD6/Duplicates+Finder+\(.NET\)](http://confluence.jetbrains.com/display/TCD6/Duplicates+Finder+(.NET)).

With Simian, it's easy to track down code duplication and figure out how much work you have ahead of you, as well as refactor to remove duplication. Simian is a commercial product that works on .NET, Java, C++, and other languages. You can get Simian here: www.harukizaemon.com/simian/.

10.5 Summary

In this chapter, I talked about how to approach legacy code for the first time. It's important to map out the various components according to their number of dependencies, their amount of logic, and the project priority. Once you have that information, you can choose the components to work on based on how easy or how hard it will be to get them under test.

If your team has little or no experience in unit testing, it's a good idea to start with the easy components and let the team's confidence grow as they add more and more tests to the system. If your team is experienced, getting the hard components under test first can help you get through the rest of the system more quickly.

If your team doesn't want to start refactoring code for testability, but only to start with unit testing out of the box, using unconstrained isolation frameworks will prove helpful because they allow you to isolate dependencies without changing the existing code's design. Consider them when dealing with legacy .NET code. If you work with Java, consider JMockit or PowerMock for the same reasons.

I also covered a number of tools that can prove helpful in your journey to better code quality for existing code. Each of these tools can be used in different stages of the project, but it's up to your team to choose when to use which tool (if any at all).

Finally, as a friend once said, a good bottle of vodka never hurts when dealing with legacy code.