Chapter 9

I Can't Get This Class into a Test Harness

This is the hard one. If it were always easy to instantiate a class in a test harness, this book would be a lot shorter. Unfortunately, it's often hard to do.

Here are the four most common problems we encounter:

1. Objects of the class can't be created easily.
2. The test harness won't easily build with the class in it.
3. The constructor we need to use has bad side effects.
4. Significant work happens in the constructor, and we need to sense it.

In this chapter, we go through a series of examples that highlight these problems in different languages. There is more than one way to tackle each of these problems. However, reading through these examples is a great way of becoming familiar with the arsenal of dependency breaking techniques and learning how to trade them off and apply them in particular situations.

The Case of the Irritating Parameter

When I need to make a change in a legacy system, I usually start out buoyantly optimistic. I don't know why I do. I try to be a realist as much as I can, but the optimism is always there. "Hey," I say to myself (or a partner), "this sounds like it will be easy. We just have to make the `Floogle flumoux` a bit, and then we'll be done." It all sounds so easy in words until we get to the `Floogle` class (whatever that is) and look at it a bit. "Okay, so we need to add a method here, and change this other method, and, of course we'll need to get it in a testing harness." At this point, I start to doubt a little. "Gee, it looks like the simplest constructor on this class accepts three parameters. But," I say optimistically, "maybe it won't be too hard to construct it."

Let's take a look at an example and see whether my optimism is appropriate or just a defense mechanism.

In the code for a billing system, we have an untested Java class named `CreditValidator`.

```
public class CreditValidator
{
    public CreditValidator(RGHConnection connection,
                           CreditMaster master,
                           String validatorID) {
        ...
    }

    Certificate validateCustomer(Customer customer)
            throws InvalidCredit {
        ...
    }

    ...
}
```

One of the many responsibilities of this class is to tell us whether customers have valid credit. If they do, we get back a certificate that tells us how much credit they have. If they don't, the class throws an exception.

Our mission, should we choose to accept it, it is to add a new method to this class. The method will be named `getValidationPercent`, and its job will be to tell us the percentage of successful `validateCustomer` calls we've made over the life of the `validator`.

How do we get started?

When we need to create an object in a test harness, often the best approach is to just try to do it. We could do a lot of analysis to find out why it would or would not be easy or hard, but it is just as easy to create a JUnit test class, type this into it, and compile the code:

```
public void testCreate() {
     CreditValidator validator = new CreditValidator();
}
```

The best way to see if you will have trouble instantiating a class in a test harness is to just try to do it. Write a test case and attempt to create an object in it. The compiler will tell you what you need to make it really work.

This test is a construction test. Construction tests do look a little weird. When I write one, I usually don't put an assertion in it. I just try to create the object. Later, when I'm finally able to construct an object in the test harness, I usually get rid of the test or rename it so that I can use it to test something more substantial.

Back to our example:

We haven't added any of the arguments to the constructor yet, so the compiler complains. It tells us that there is no default constructor for `CreditValidator`. Hunting through the code, we discover that we need an `RGHConnection`, a `CreditMaster`, and a password. Each of these classes has only one constructor. This is what they look like:

```
public class RGHConnection
{
    public RGHConnection(int port, string name, string passwd)
            throws IOException {
        ...
    }
}

public class CreditMaster
{
    public CreditMaster(string filename, boolean isLocal) {
        ...
    }
}
```

When an `RGHConnection` is constructed, it connects with a server. The connection uses that server to get all of the reports it needs to validate a customer's credit.

The other class, `CreditMaster`, gives us some policy information that we use in our credit decisions. On construction, a `CreditMaster` loads the information from a file and holds it in memory for us.

So, it does seem pretty easy to get this class in a testing harness, right? Not so fast. We can write the test, but can we live with it?

```
public void testCreate() throws Exception {
    RGHConnection connection = new RGHConnection(DEFAULT_PORT,
                                            "admin", "rii8ii9s");
    CreditMaster master = new CreditMaster("crm2.mas", true);
    CreditValidator validator = new CreditValidator(
                                      connection, master, "a");
}
```

It turns out that establishing RGHConnections to the server in a test is not a good idea. It takes a long time, and the server isn't always up. On the other hand, the CreditMaster is not really a problem. When we create a CreditMaster, it loads its file quickly. In addition, the file is read-only, so we don't have to worry about our tests corrupting it.

The thing that is really getting in our way when we want to create the validator is the RGHConnection. It is an *irritating parameter.* If we can create some sort of a fake RGHConnection object and make CreditValidator believe that it's talking to a real one, we can avoid all sorts of connection trouble. Let's take a look at the methods that RGHConnection provides (see Figure 9.1).

It looks like RGHConnection has a set of methods that deal with the mechanics of forming a connection: connect, disconnect, and retry, as well as more business-specific methods such as RFDIReportFor and ACTIOReportFor. When we write our new method on CreditValidator, we are going to have to call RFDIReportFor to get all of the information that we need. Normally, all of that information comes from the server, but because we want to avoid using a real connection, we'll have to find some way to supply it ourselves.

In this case, the best way to make a fake object is to use *Extract Interface (362)* on the RGHConnection class. If you have a tool with refactoring support, chances are good that it supports *Extract Interface*. If you don't have a tool that supports *Extract Interface*, remember that it is easy enough to do by hand.

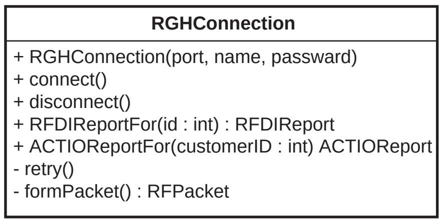
RGHConnection
+ RGHConnection(port, name, passward)
+ connect()
+ disconnect()
+ RFDIReportFor(id : int) : RFDIReport
+ ACTIOReportFor(customerID : int) ACTIOReport
- retry()
- formPacket() : RFPacket

Figure 9.1 RGHConnection.

After we do *Extract Interface (362)*, we end up with a structure like the one shown in Figure 9.2.

We can start to write tests by creating a little fake class that provides the reports that we need:

```
public class FakeConnection implements IRGHConnection
{
    public RFDIReport report;

    public void connect() {}
    public void disconnect() {}
    public RFDIReport RFDIReportFor(int id) { return report; }
    public ACTIOReport ACTIOReportFor(int customerID) { return null; }
}
```

With that class, we can start to write tests like this:

```
void testNoSuccess() throws Exception {
    CreditMaster master = new CreditMaster("crm2.mas", true);
    IRGHConnection connection = new FakeConnection();
    CreditValidator validator = new CreditValidator(
                                    connection, master, "a");
    connection.report  = new RFDIReport(...);

    Certificate result = validator.validateCustomer(new Customer(...));

    assertEquals(Certificate.VALID, result.getStatus());
}
```

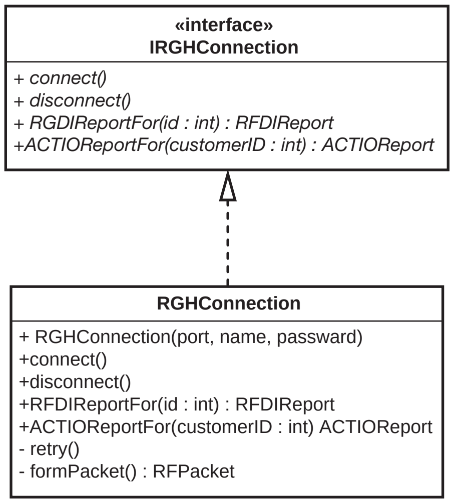
«interface»
IRGHConnection
+ connect()
+ disconnect()
+ RGDIReportFor(id : int) : RFDIReport
+ACTIOReportFor(customerID : int) : ACTIOReport
RGHConnection
+ RGHConnection(port, name, passward)
+connect()
+disconnect()
+RFDIReportFor(id : int) : RFDIReport
+ACTIOReportFor(customerID : int) ACTIOReport
- retry()
- formPacket() : RFPacket

Figure 9.2 `RGHConnection` *after extracting an interface*

The FakeConnection class is a little weird. How often do we ever write methods that don't have any bodies or that just return null to callers? Worse, it has a public variable that anyone can set whenever they want to. It seems like the class violates all of the rules. Well, it doesn't really. The rules are different for classes that we use to make testing possible. The code in FakeConnection isn't production code. It won't ever run in our full working application—just in the test harness.

Now that we can create a validator, we can write our getValidationPercent method. Here is a test for it.

```
void testAllPassed100Percent() throws Exception {
    CreditMaster master = new CreditMaster("crm2.mas", true);
    IRGHConnection connection = new FakeConnection("admin", "rii8ii9s");
    CreditValidator validator = new CreditValidator(
                                    connection, master, "a");

    connection.report  = new RFDIReport(...);
    Certificate result = validator.validateCustomer(new Customer(...));
    assertEquals(100.0, validator.getValidationPercent(), THRESHOLD);
}
```

Test Code vs. Production Code

Test code doesn't have to live up to the same standards as production code. In general, I don't mind breaking encapsulation by making variables public if it makes it easier to write tests. However, test code should be clean. It should be easy to understand and change.

Take a look at the testNoSuccess and testAllPassed100Percent tests in the example. Do they have any duplicate code? Yes. The first three lines are duplicated. They should be extracted and placed in a common place, the setUp() method for this test class.

The test checks to see if the validation percent is roughly 100.0 when we get a single valid credit certificate.

The test works fine, but as we write the code for getValidationPercent, we notice something interesting. It turns out that getValidationPercent isn't going to use the CreditMaster at all, so why are we making one and passing it into the CreditValidator? Maybe we don't need to. We could create the CreditValidator like this in our test:

```
CreditValidator validator = new CreditValidator(connection, null, "a");
```

Are you still there?

The way people react to lines of code like that often says a lot about the kind of system they work on. If you looked at it and said, "Oh, fine, so he's passing a null into the constructor—we do that all the time in our system," chances are, you've got a pretty nasty system on your hands. You probably have checks for null all over the place and a lot of conditional code that you use to figure out what you have and what you can do with it. On the other hand, if you looked at it and said, "What is wrong with this guy?! Passing null around in a system? Doesn't he know anything at all?", well, for those of you in the latter group (at least those who are still reading and haven't slammed the book shut in the bookstore), I just have this to say: Remember, we're only doing this in the tests. The worst that can happen is that some code will attempt to use the variable. In that case, the Java runtime will throw an exception. Because the harness catches all exceptions thrown in tests, we'll find out pretty quickly whether the parameter is being used at all.

Pass Null

When you are writing tests and an object requires a parameter that is hard to construct, consider just passing null instead. If the parameter is used in the course of your test execution, the code will throw an exception and the test harness will catch the exception. If you need behavior that really requires an object, you can construct it and pass it as a parameter at that point.

Pass Null is a very handy technique in some languages. It works well in Java and C# and in just about every language that throws an exception when null references are used at runtime. This implies that it really isn't a great idea to do this in C and C++ unless you know that the runtime will detect null pointer errors. If it doesn't, you'll just end up with tests that will crash mysteriously, if you are lucky. If you are unlucky, your tests will just be silently and hopelessly wrong. They will corrupt memory as they run, and you'll never know.

When I work in Java, I often start with a test like this in the beginning and fill in the parameters as I need them.

```
public void testCreate() {
    CreditValidator validator = new CreditValidator(null, null, "a");
}
```

The important thing to remember is this: Don't pass null in production code unless you have no other choice. I know that some libraries out there expect you to, but when you write fresh code there are better alternatives. If you are tempted to use null in production code, find the places where you are returning nulls and passing nulls, and consider a different protocol. Consider using the *Null Object Pattern* instead.

Null Object Pattern

The *Null Object Pattern* is a way of avoiding the use of null in programs. For example, if we have a method that is going to return an employee given an ID, what should we return if there is no employee with that ID?

```
for(Iterator it = idList.iterator(); it.hasNext(); ) {
       EmployeeID id = (EmployeeID)it.next();
       Employee e = finder.getEmployeeForID(id);
       e.pay();
}
```

We have a couple of choices. We could just decide to throw an exception so that we don't have to return anything, but that would force clients to deal with the error explicitly. We could also return null, but then clients would have to check for null explicitly.

There is a third alternative. Does the previous code really care whether there is an employee to pay? Does it have to? What if we had a class called `NullEmployee`? An instance of `NullEmployee` has no name and no address, and when you tell it to pay, it just does nothing.

Null objects can be useful in contexts like this; they can shield clients from explicit error checking. As nice as null objects are, you have to be cautious when you use them. For instance, here is a very bad way of counting the number of paid employees:

```
int employeesPaid = 0;
for(Iterator it = idList.iterator(); it.hasNext(); ) {
   EmployeeID id = (EmployeeID)it.next();
   Employee e = finder.getEmployeeForID(id);
   e.pay();
   mployeesPaid++;       // bug!
}
```

If any of the returned employees are null employees, the count will be wrong.

Null objects are useful specifically when a client doesn't have to care whether an operation is successful. In many cases, we can finesse our design so that this is the case.

Pass Null and *Extract Interface (362)* are two ways of approaching irritating parameters. But another alternative can be used at times. If the problematic dependency in a parameter isn't hard-coded into its constructor, we can use *Subclass and Override Method (401)* to get rid of the dependency. That could be possible in this case. If the constructor of `RGHConnection` uses its `connect` method to form a connection, we could break the dependency by overriding `connect()` in a testing subclass. *Subclass and Override Method (401)* can be a very useful way of breaking dependencies, but we have to be sure that we aren't altering the behavior we want to test when we use it.

The Case of the Hidden Dependency

Some classes are deceptive. We look at them, we find a constructor that we want to use, and we try to call it. Then, bang! We run into an obstacle. One of the most common obstacles is *hidden dependency*; the constructor uses some resource that we just can't access nicely in our test harness. We run into this situation in this next example, a poorly designed C++ class that manages a mailing list:

```
class mailing_list_dispatcher
{
public:
                    mailing_list_dispatcher ();
    virtual         ~mailing_list_dispatcher;

    void            send_message(const std::string& message);
    void            add_recipient(const mail_txm_id id,
                                const mail_address& address);
    ...

private:
    mail_service    *service;
    int             status;
};
```

Here is part of the constructor of the class. It allocates a `mail_service` object using `new` in the constructor initializer list. That is poor style, and it gets worse. The constructor does a lot of detailed work with the `mail_service`. It also uses a magic number, 12—what does 12 mean?

```
mailing_list_dispatcher::mailing_list_dispatcher()
: service(new mail_service), status(MAIL_OKAY)
{
    const int client_type = 12;
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```

We can create an instance of this class in a test, but it's probably not going to do us much good. First of all, we'll have to link to the mail libraries and configure the mail system to handle registrations. And if we use the send_message function in our tests, we'll really be sending mail to people. It will be hard to test that functionality in an automated way unless we set up a special mailbox and connect to it repeatedly, waiting for a mail message to arrive. That could be great as an overall system test, but if all we want to do now is add some new tested functionality to the class, that could be overkill. How can we create a simple object to add some new functionality?

The fundamental problem here is that the dependency on mail_service is hidden in the mailing_list_dispatcher constructor. If there was some way to replace the mail_service object with a fake, we could sense through the fake and get some feedback as we change the class.

One of the techniques we can use is *Parameterize Constructor (379)*. With this technique, we externalize a dependency that we have in a constructor by passing it into the constructor.

This is what the constructor code looks like after *Parameterize Constructor (379)*:

```
mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
: status(MAIL_OKAY)
{
    const int client_type = 12;
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```

The only difference, really, is that the mail_service object is created outside the class and passed in. That might not seem like much of an improvement, but it does give us incredible leverage. We can use *Extract Interface (362)* to make an interface for mail_service. One implementer of the interface can be the production class that really sends mail. Another can be a fake class that senses the things that we do to it under test and lets us make sure that they happened.

Parameterize Constructor (379) is a very convenient way to externalize constructor dependencies, but people don't think of it very often. One of the stumbling blocks is that people often assume that all clients of the class will have to be changed to pass the new parameter, but that isn't true. We can handle it like this. First we extract the body of the constructor into a new method that we can call `initialize`. Unlike most method extractions, this one is pretty safe to attempt without tests because we can *Preserve Signatures (312)* as we do it.

```
void mailing_list_dispatcher::initialize(mail_service *service)
{
    status = MAIL_OKAY;
    const int client_type = 12;
    service.connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}

mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
{
    initialize(service);
}
```

Now we can supply a constructor that has the original signature. Tests can call the constructor parameterized by `mail_service`, and clients can call this one. They don't need to know that anything has changed.

```
mailing_list_dispatcher::mailing_list_dispatcher()
{
    initialize(new mail_service);
}
```

This refactoring is even easier in languages such as C# and Java because we can call constructors from other constructors in those languages.

For instance, if we were doing something similar in C#, the resultant code would look like this:

```
public class MailingListDispatcher
{
    public MailingListDispatcher()
    : this(new MailService())
    {}

    public MailingListDispatcher(MailService service) {
        ...
    }
}
```

Dependencies hidden in constructors can be tackled with many techniques. Often we can use *Extract and Override Getter (xx)*, *Extract and Override*

Factory Method (350), and *Supersede Instance Variable (404)*, but I like to use *Parameterize Constructor (379)* as often as I can. When an object is created in a constructor and it doesn't have any construction dependencies itself, *Parameterize Constructor* is a very easy technique to apply.

The Case of the Construction Blob

Parameterize Constructor (379) is one of the easiest techniques that we can use to break hidden dependencies in a constructor, and it is the one that I often turn to first. Unfortunately, it isn't always the best choice. If a constructor constructs a large number of objects internally or accesses a large number of globals, we could end up with a very large parameter list. In worse situations, a constructor creates a few objects and then uses them to create other objects, like this:

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
    {
        ...
        anteriorPanel  = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }
    ...
}
```

If we want to sense through the cursor, we are in trouble. The cursor object is embedded in a blob of object creation. We can try to move all of the code used to create the cursor outside of the class. Then a client can create the cursor and pass it as an argument. But that isn't very safe if we don't have tests in place, and it could be a big burden on clients on this class.

If we have a refactoring tool that safely extracts methods, we can use *Extract and Override Factory Method (350)* on code in a constructor, but that doesn't work in all languages. In Java and C#, we can do it, but C++ doesn't allow calls to virtual functions in constructors to resolve to virtual functions defined in derived classes. And in general, it isn't a good idea. Functions in derived classes often assume that they can use variables from their base class. Until the constructor of the base class is completely finished, there is a chance that an overridden function that it calls can access an uninitialized variable.

Another option is *Supersede Instance Variable (404)*. We write a setter on the class that allows us to swap in another instance after we construct the object.

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
    {
        ...
        anteriorPanel  = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }

    void supersedeCursor(FocusWidget *newCursor)
    {
        delete cursor;
        cursor = newCursor;
    }
}
```

In C++, we have to be very careful with this refactoring. When we replace an object, we have to get rid of the old one. Often that means that we have to use the delete operator to call its destructor and destroy its memory. When we do that, we have to understand what the destructor does and whether it destroys anything that is passed to the object's constructor. If we are not careful about how we clean up memory, we can introduce some subtle bugs.

In most other languages, *Supersede Instance Variable (404)* is pretty straightforward. Here is the result recoded in Java. We don't have to do anything special to get rid of the object that `cursor` was referring to; the garbage collector will get rid of it eventually. But we should be very careful not to use the superseding method in production code. If the objects that we are superseding manage other resources, we can cause some serious resource problems.

```
void supersedeCursor(FocusWidget newCursor) {
    cursor = newCursor;
}
```

Now that we have a superseding method, we can attempt to create a `FocusWidget` outside the class and then pass it into the object after construction. Because we need to sense, we can use *Extract Interface (362)* or *Extract Implementer (356)* on the `FocusWidget` class and create a fake object to pass in. It will certainly be easier to create than the `FocusWidget` that is created in the constructor.

```
TEST(renderBorder, WatercolorPane)
{
    ...
    TestingFocusWidget *widget = new TestingFocusWidget;
    WatercolorPane pane(form, border, backdrop);

    pane.supersedeCursor(widget);

    LONGS_EQUAL(0, pane.getComponentCount());
}
```

I don't like to use *Supersede Instance Variable (404)* unless I can't avoid it. The potential for resource-management problems is too great. However, I do use it in C++ at times. Often I'd like to use *Extract and Override Factory Method (350)*, and we can't do that in C++ constructors. For that reason, I use *Supersede Instance Variable (404)* occasionally.

The Case of the Irritating Global Dependency

For years in the software industry, people have bemoaned the fact that there aren't more reusable components on the market. It's getting better over time; there are plenty of commercial and open-source frameworks, but in general, many of them are not really things that we use; they are things that use our code. Frameworks often manage the lifecycle of an application, and we write code to fill in the holes. We can see this in all sorts of frameworks, from ASP.NET to Java Struts. Even the xUnit frameworks behave this way. We write test classes; xUnit calls them and displays their results.

Frameworks solve many problems, and they do give us a boost when we start projects, but this isn't the kind of reuse that people really expected early on in software development. Old-style reuse happens when we find some class or set of classes that we want to use in our application and we just do it. We just add them to a project and use them. It would be nice to be able to do this routinely, but frankly, I think we are kidding ourselves even thinking about that sort of reuse if we can't pull a random class out of an average application and compile it independently in a test harness without doing a lot of work (grumble, grumble).

Many different kinds of dependency can make it hard to create and use classes in a testing framework, but one of the hardest to deal with is global variable usage. In simple cases, we can use *Parameterize Constructor (379)*, *Parameterize Method (383)*, and *Extract and Override Call (348)* to get past these dependencies, but sometimes dependencies on globals are so extensive that it is easier to deal with the problem at the source. We run into this situation in this next example, a class in a Java application that records building permits for a governmental agency. Here is one of the primary classes:

```
public class Facility
{
    private Permit basePermit;

    public Facility(int facilityCode, String owner, PermitNotice notice)
                throws PermitViolation {

        Permit associatedPermit =
            PermitRepository.getInstance().findAssociatedPermit(notice);

        if (associatedPermit.isValid() && !notice.isValid()) {
            basePermit = associatedPermit;
        }
        else if (!notice.isValid()) {
            Permit permit = new Permit(notice);
            permit.validate();
            basePermit = permit;
        }
        else
            throw new PermitViolation(permit);
    }
    ...
}
```

We want to create a `Facility` in a test harness, so we start by trying to create an object in the test harness:

```
public void testCreate() {
    PermitNotice notice = new PermitNotice(0, "a");
    Facility facility = new Facility(Facility.RESIDENCE, "b", notice);
}
```

The test compiles okay, but when we start to write additional tests, we notice a problem. The constructor uses a class named `PermitRepository`, and it needs to be initialized with a particular set of permits to set up our tests properly. Sneaky, sneaky. Here is the offending statement in the constructor:

```
Permit associatedPermit =
            PermitRepository.getInstance().findAssociatedPermit(notice);
```

We could get past this by parameterizing the constructor, but in this application, this isn't an isolated case. There are 10 other classes that have roughly the same line of code. It sits in constructors, regular methods, and static methods. We can imagine spending a lot of time confronting this problem in the code base.

If you've studied design patterns, you probably recognize this as an example of the *Singleton Design Pattern (372)*. The getInstance method of PermitRepository is a static method whose job is to return the only instance of PermitRepository that can exist in our application. The field that holds that instance is static also, and it lives in the PermitRepository class.

In Java, the singleton pattern is one of the mechanisms people use to make global variables. In general, global variables are a bad idea for a couple of reasons. One of them is opacity. When we look at a piece of code, it is nice to be able to know what it can affect. For instance, in Java, when we want to understand how this piece of code can affect things, we have to look only a couple places:

```
Account example = new Account();
example.deposit(1);
int balance = example.getBalance();
```

We know that an account object can affect things that we pass into the Account constructor, but we aren't passing anything in. Account objects can also affect objects that we pass as parameters to methods, but in this case, we aren't passing anything in that can be changed—it's just an int. Here we are assigning the return value of getBalance to a variable, and that is really the only value that should be affected by this set of statements.

When we use global variables, this situation is turned upside down. We can look at the use of a class such as Account and not have a clue whether it is accessing or modifying variables declared someplace else in the program. Needless to say, this can make it harder to understand programs.

The tough part in a testing situation is that we have to figure which globals are being used by a class and set them up with the proper state for a test. And we have to do that before each test if the setup is different. It's pretty tedious; I've done it on dozens of systems to get them under test, and it doesn't get any more enjoyable.

The Case of the Irritating Global Dependency

Back to our regularly scheduled example:

PermitRepository is a singleton. Because it is, it is particularly hard to fake. The whole idea of the singleton pattern is to make it impossible to create more than one instance of a singleton in an application. That might be fine in production code, but, when testing, each test in a suite of tests should be a mini-application, in a way: It should be totally isolated from the other tests. So, to run code containing singletons in a test harness, we have to relax the singleton property. Here's how we do it.

The first step is to add a new static method to the singleton class. The method allows us to replace the static instance in the singleton. We'll call it `setTestingInstance`.

```
public class PermitRepository
{
    private static PermitRepository instance = null;

    private PermitRepository() {}

    public static void setTestingInstance(PermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static PermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice) {
        ...
    }
    ...
}
```

Now that we have that setter, we can create a testing instance of a `PermitRepository` and set it. We'd like to write code like this in our test setup:

```
public void setUp() {
    PermitRepository repository = new PermitRepository();
    ...
    // add permits to the repository here
    ...
    PermitRepository.setTestingInstance(repository);
}
```

Introduce Static Setter (372) isn't the only way of handling this situation. Here is another approach. We can add a `resetForTesting()` method to the singleton that looks like this:

```
public class PermitRepository
{
    ...
    public void resetForTesting() {
        instance  = null;
    }
    ...
}
```

If we call this method in our test `setUp` (and it's a good idea to call it in `tearDown` also), we can create fresh singletons for every test. The singleton will reintialize itself for every test. This scheme works well when the public methods on the singleton allow you to set up the singleton's state every way you need to during testing. If the singleton doesn't have those public methods or uses some external resources that affect its state, *Introduce Static Setter (372)* is the better choice. You can subclass the singleton, override methods to break dependencies, and add public methods to the subclass to set up state properly.

Will that work? Not yet. When people use the *Singleton Design Pattern (372)*, they often make the constructor of the singleton class private, and with good reason. That is the clearest way to make sure that no one outside the class can make another instance of the singleton.

At this point, we have a conflict between two design goals. We want to make sure that we have only one instance of a `PermitRepository` in a system, and we want a system in which the classes are testable independently. Can we have both?

Let's backtrack for a minute. Why do we want only one instance of a class in a system? The answer varies depending on the system, but here are some of the most common answers:

1. **We are modeling the real world, and there is only one of these things in the real world.** Some hardware-control systems are like this. People make a class for each circuit board they need to control; they figure that if there is just one of each, it should be a singleton. The same holds true for databases. There is only one collection of permits in our agency, so the thing that provides access to it should be a singleton.

2. **If two of these things are created, we could have a serious problem.** This often happens, again, in the hardware control domain. Imagine accidentally creating two nuclear control rod controllers and having two different parts of a program operating the same control rods without knowing about each other.

3. **If someone creates two of these things, we'll be using too many resources.** This happens often. The resources can be physical things such as disk space or memory consumption, or they can be abstract things such as the number of software licenses.

Those are the primary reasons why people want to enforce a single instance, but they aren't the primary reasons why people use singletons. Often people create singletons because they want to have a global variable. They feel that it would be too painful to pass the variable around to the places where it is needed.

If we have a singleton for the latter reason, there really isn't any reason to keep the singleton property. We can make the constructor protected, public, or package scope and still have a decent, testable system. In the other cases, it is still worth exploring that alternative. We can introduce other protection if we need to. We could add a check to our build system in which we search through all the source files to make sure that `setTestingInstance` is not called by non-testing code. We can do the same thing with runtime checks. If `setTestingInstance` is called at runtime, we can issue an alarm or suspend the system and wait for operator intervention. The truth is, it wasn't possible to enforce singleton-ness in many pre-OO languages, and people did manage to make many safe systems. In the end, it comes down to responsible design and coding.

If breaking the singleton property isn't a serious problem, we can rely on a team rule. For instance, everyone on the team should understand that we have one instance of the database in the application and that we shouldn't have another.

To relax the singleton property on `PermitRepository`, we can make the constructor public. And that will work fine for us as long as the public methods on `PermitRepository` allow us to do everything that we need to set up a repository for our tests. For example, if `PermitRepository` has a method named `addPermit` that allows us to fill it up with whatever permits we need for our tests, it might be enough to just allow ourselves to make repositories and use them in our tests. At other times, we might not have the access we need, or, worse, the singleton might be doing things that we would not want to have happen in a test harness, such as talk to a database in the background. In these cases, we can *Subclass and Override Method (401)* and make derived classes that make testing easier.

Here is an example in our permit system. In addition to the method and variables that make `PermitRepository` a singleton, we have the following method:

```
public class PermitRepository
{
    ...
    public Permit findAssociatedPermit(PermitNotice notice) {
        // open permit database
        ...

        // select using values in notice
        ...

        // verify we have only one matching permit, if not report error
        ...

        // return the matching permit
        ...
}
}
```

If we want to avoid talking to the database, we can subclass PermitRepository like this:

```
public class TestingPermitRepository extends PermitRepository
{
    private Map permits = new HashMap();

    public void addAssociatedPermit(PermitNotice notice, permit) {
        permits.put(notice, permit);
    }

    public Permit findAssociatedPermit(PermitNotice notice) {
        return (Permit)permits.get(notice);
    }
}
```

When we do this, we can preserve part of the singleton property. Because we are using a subclass of PermitRepository, we can make the constructor of PermitRepository protected rather than public. That will prevent the creation of more than one PermitRepository, although it does allow us to create subclasses.

```
public class PermitRepository
{
    private static PermitRepository instance = null;

    protected PermitRepository() {}

    public static void setTestingInstance(PermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static PermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice)
    {
        ...
    }
    ...
}
```

In many cases, we can use *Subclass and Override Method (401)* like this to get a fake singleton in place. At other times, the dependencies are so extensive that it is easier to use *Extract Interface (362)* on the singleton and change all of the references in the application so that they use the interface name. This can be a lot of work, but we can *Lean on the Compiler (315)* to make the change. This is what the PermitRepository class will look like after the extraction:

```
public class PermitRepository implements IPermitRepository
{
    private static IPermitRepository instance = null;

    protected PermitRepository() {}

    public static void setTestingInstance(IPermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static IPermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice)
    {
        ...
    }
    ...
}
```

The IPermitRepository interface will have signatures for all of the public non-static methods on PermitRepository.

```
public interface IPermitRepository
{
    Permit findAssociatedPermit(PermitNotice notice);
    ...
}
```

If you are using a language that has a refactoring tool, you might be able to perform this interface extraction automatically. If you are using a language without one, it might be easier to use *Extract Implementer (356)* instead.

The name for this whole refactoring is *Introduce Static Setter (372)*. This is a technique that we can use to get tests in place despite extensive global dependencies. Unfortunately, it doesn't do much to get past the global dependencies. If you choose to tackle that problem, you can do so by using *Parameterize Method (383)* and *Parameterize Constructor (379)*. With those refactorings, you trade a global reference for either a temporary variable in a method or a field in an object. The downside to *Parameterize Method (383)* is that you can end up with many additional methods that distract people when they try to understand the classes. The downside to *Parameterize Constructor (379)* is that each object that currently uses the global ends up with an additional field. The field will have to be passed to its constructor, so the class that creates the object needs to have access to the instance also. If too many objects need this additional field, it can substantially impact the amount of memory used by the application, but often that indicates other design problems.

Let's look at the worst case. We have an application with several hundred classes that creates thousands of objects at runtime, and each of them needs access to the database. Without even looking at the application, the first question that comes to my mind is, why? If the system does anything more than access a database, it can be factored so that some classes do those other things and others store and retrieve data. When we make a concerted effort to separate responsibilities in an application, dependencies become localized; we might not need a reference to a database in every object. Some objects are populated using data retrieved from the database. Others perform calculation on data supplied through their constructors.

As an exercise, pick a global variable in a large application and search for it. In most cases, variables that are global are globally accessible, but they really aren't globally used. They are used in a relatively small number of places. Imagine how we could get that object to the objects that need it if it couldn't be a global variable. How would we refactor the application? Are there responsibilities that we can separate out of sets of classes to decrease the scope of the global?

If you find a global variable that really is being used every place, it means there isn't any layering in your code. Take a look at Chapter 15, *My Application Is All API Calls,* and Chapter 17, *My Application Has No Structure*.

The Case of the Horrible Include Dependencies

C++ was my first OO language, and I have to admit that I felt very proud of myself for learning many of its details and complexities. It became dominant in the industry because it was an utterly pragmatic solution to many vexing problems at the time. Machines are too slow? Okay, here is a language in which everything is optional. You can get all of the efficiency of raw C if you use only the C features. Can't get your team to use an OO language? Okay, here is a C++ compiler; you can write in the C subset of C++ and learn OO as you go.

Although C++ became very popular for a while, it eventually fell behind Java and some of the newer languages in popularity. There was leverage in maintaining backward compatibility with C, but there was much more leverage in making languages easier to work with. Repeatedly, C++ teams have learned that the language defaults are not ideal for maintenance, and they have to go beyond them a bit to keep a system nimble and easy to change.

One part of C++'s C legacy that is especially problematic is its way of letting one part of a program know about another part. In Java and C#, if a class in one file needs to use a class in another file, we use an import or using statement to make its definition available. The compiler looks for that class and checks to see if it has been compiled already. If it hasn't, it compiles it. If it has been compiled, the compiler reads a brief snippet of information from the compiled file, getting only as much information as it needs to make sure that all of the methods the original class needs are on that class.

C++ compilers generally don't have this optimization. In C++, if a class needs to know about another class, the declaration of the class (in another file) is textually included in the file that needs to use it. This can be a much slower process. The compiler has to reparse the declaration and build up an internal representation every time it sees that declaration. Worse, the include mechanism is prone to abuse. A file can include a file that includes a file, and so on. On projects in which people haven't avoided this, it's not uncommon to find small files that end up transitively including tens of thousands of lines of code. People wonder why their builds take so long, but because the includes are spread around the system, it is hard to point at any one particular file and understand why it is taking so long to compile.

It might seem like I'm getting down on C++, but I'm not. It is an important language, and there is an incredible amount of C++ code out there—but it does take extra care to work with it well.

In legacy code, it can be hard to instantiate a C++ class in a test harness. One of the most immediate issues we confront is header dependency. What header files do we need to create a class by itself in a test harness?

Here is part of the declaration of a huge C++ class named `Scheduler`. It has more than 200 methods, but I've shown only about 5 of them in the declaration. In addition to being large, the class has very severe and tangled dependencies on many other classes. How can we make a `Scheduler` in a test?

```
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include "Meeting.h"
#include "MailDaemon.h"
...
#include "SchedulerDisplay.h"
#include "DayTime.h"

class Scheduler
{
public:
        Scheduler(const string& owner);
        ~Scheduler();

    void addEvent(Event *event);
    bool hasEvents(Date date);
    bool performConsistencyCheck(string& message);
    ...
};

#endif
```

The Case of the Horrible Include Dependencies

Among other things, the `Scheduler` class uses `Meetings`, `MailDaemons`, `Events`, `SchedulerDisplays`, and `Dates`. If we want to create a test for Schedulers, the easiest thing that we can do is try to build one in the same directory in another file named `SchedulerTests`. Why do we want the tests in the same directory? In the presence of the preprocessor, it is often just easier. If the project doesn't use paths to include files in consistent ways, we could have a lot of work to do if we try to create the tests in other directories.

```
#include "TestHarness.h"
#include "Scheduler.h"

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

If we create a file and just type that object declaration into a test, we'll be confronted with the include problem. To compile a Scheduler, we have to make sure that the compiler and linker know about all of the things that Scheduler needs, and all of the things those things need, and so on. Luckily, the build system gives us a large number of error messages and tells us about these things in exhaustive detail.

In simple cases, the Scheduler.h file includes everything we need to be able to create a Scheduler, but in some cases, the header file doesn't include everything. We have to supply some additional includes to create and use an object.

We could just copy over all of the #include directives from the Scheduler class source file, but the fact is, we might not need them all. The best tack to take is to add them one at a time and decide whether we really need those particular dependencies.

In an ideal world, the easiest thing would be to include all of the files that we need until we don't have any build errors, but that can force us into a muddle. If there is a long line of transitive dependencies, we could end up including far more than we really need. Even if the line of dependencies isn't too long, we could end up depending on things that are very hard to work with in a test harness. In this example, the SchedulerDisplay class is one of those dependencies. I'm not showing it here, but it is actually accessed in the constructor of Scheduler. We can get rid of the dependency like this:

```
#include "TestHarness.h"
#include "Scheduler.h"

void SchedulerDisplay::displayEntry(const string& entyDescription)
{
}

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

Here we've introduced an alternative definition for SchedulerDisplay::displayEntry. Unfortunately, when we do this, we need to have a separate build for the test cases in this file. We can have only one definition for each method in

SchedulerDisplay in a program, so we need to have a separate program for our scheduler tests.

Luckily, we can get some reuse for the fakes that we create this way. Instead of putting the definitions of classes such as SchedulerDisplay inline in the test file, we can put them in a separate include file that can be used across a set of test files:

```
#include "TestHarness.h"
#include "Scheduler.h"
#include "Fakes.h"

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

After doing it a couple of times, getting a C++ class instantiable in a harness like this is pretty easy and pretty mechanical. There are a couple of very serious downsides. We have to create that separate program, and we really aren't breaking dependencies at the language level, so we aren't making the code cleaner as we break dependencies. Worse, those duplicate definitions that we put in the test file (SchedulerDisplay::displayEntry in this example) have to be maintained as long as we keep this set of tests in place.

I reserve this technique for cases in which I have a very huge class with very severe dependency problems. It is not a technique to use often or lightly. If that class is going to be broken up into a large number of smaller classes over time, creating a separate test program for a class can be useful. It can act as a testing point for a lot of refactoring. Over time, this separate testing program can go away as you extract more classes and get them under test.

The Case of the Onion Parameter

I like simple constructors. I really do. It is great to be able to decide to create a class and then just type in a constructor call and have a nice live, working object available to use. But in many cases, it can be hard to create objects. Every object needs to be set up in a good state, a state that makes it ready for additional work. In many cases, this means we have to supply it with objects that are set up properly themselves. Those objects might require other objects so that they can be set up also, so we end up having to create objects to create objects to create objects to create a parameter for a constructor of the class that we want to test. Objects inside of other objects—it seems like a big onion. Here is an example of this sort of problem.

We have a class that displays a `SchedulingTask`:

```
public class SchedulingTaskPane extends SchedulerPane
{
    public SchedulingTaskPane(SchedulingTask task) {
        ...
    }
}
```

To create it, we need to pass it a `SchedulingTask`, but to create a `SchedulingTask`, we have to use its one and only one constructor:

```
public class SchedulingTask extends SerialTask
{
    public SchedulingTask(Scheduler scheduler, MeetingResolver resolver)
    {
        ...
    }
}
```

If we discover that we need more objects to create `Schedulers` and `MeetingResolvers`, we're liable to pull our hair out. The only thing that keeps us from total despair is the fact that there has to be at least one class that doesn't require objects of another class as arguments. If there isn't, there is no way the system could ever have compiled.

The way to handle this situation is to take a close look at what we want to do. We need to write tests, but what do we really need from the parameters passed into the constructor? If we don't need anything from them in the tests, we can *Pass Null (111)*. If we just need some rudimentary behavior, we can use *Extract Interface (362)* or *Extract Implementer (356)* on the most immediate dependency and use the interface to create a fake object. In this case, the most immediate dependency of `SchedulingTaskPane` is `SchedulingTask`. If we can create a fake `SchedulingTask`, we can create a `SchedulingTaskPane`.

Unfortunately, `SchedulingTask` inherits from a class named `SerialTask`, and all it does is override some protected methods. All of the public methods are in `SerialTask`. Can we use *Extract Interface (362)* on `SchedulingTask`, or do we have to use it on `SerialTask`, too? In Java, we don't. We can create an interface for `SchedulingTask` that includes methods from `SerialTask` also.

Our resulting hierarchy looks like Figure 9.3.

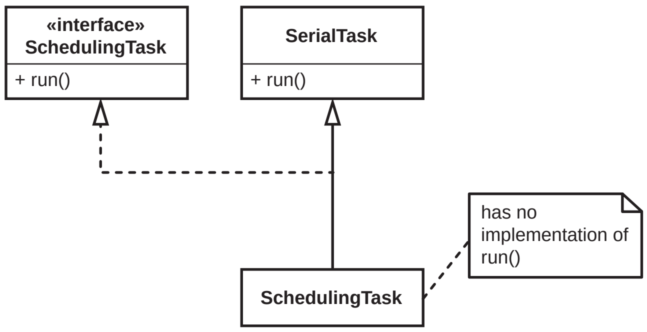

Figure 9.3 `SchedulingTask`.

In this case, we are lucky that we are using Java. In C++, unfortunately, we can't handle this case like this. There is no separate interface construct. Interfaces are typically implemented as classes containing only pure virtual functions. If this example was ported to C++, the `SchedulingTask` would become abstract because it inherits a pure virtual function from `SchedulingTask`. To instantiate a `SchedulingTask`, we'd need to provide a body for `run()` in `SchedulingTask`, which delegates to the `run()` from `SerialTask`. Fortunately, that would be easy enough to add. Here is what it looks like in code:

```
class SerialTask
{
public:
    virtual void run();
    ...
};

class ISchedulingTask
{
public:
    virtual void run() = 0;
    ...
};

class SchedulingTask : public SerialTask, public ISchedulingTask
{
public:
    virtual void run() { SerialTask::run(); }
};
```

In any language where we can create interfaces or classes that act like interfaces, we can systematically use them to break dependencies.

The Case of the Aliased Parameter

Often when we have parameters to constructors that get in the way, we can get past the problem by using *Extract Interface (362)* or *Extract Implementer (356)*. But sometimes this isn't practical. Let's take a look at another class in that building permit system that we saw in a previous section:

```
public class IndustrialFacility extends Facility
{
    Permit basePermit;

    public IndustrialFacility(int facilityCode, String owner,
                OriginationPermit permit) throws PermitViolation {

        Permit associatedPermit =
            PermitRepository.GetInstance()
                        .findAssociatedFromOrigination(permit);

        if (associatedPermit.isValid() && !permit.isValid()) {
            basePermit = associatedPermit;
        }
        else if (!permit.isValid()) {
            permit.validate();
            basePermit = permit;
        }
        else
            throw new PermitViolation(permit);
    }
    ...
}
```

We want to instantiate this class in a harness, but there are a couple of problems. One is that we are accessing a singleton again, `PermitRepository`. We can get past that problem by using the techniques we saw in the earlier section "The Case of the Irritating Global Dependency." But before we even get to that problem, we have another. It is hard to make the origination permit that we need to pass into the constructor. `OriginationPermits` have horrible dependencies. The immediate thought that I have is "Oh, I can use *Extract Interface* on the `OriginationPermit` class to get past this dependency," but it isn't that easy. Figure 9.4 shows the structure of the `Permit` hierarchy.

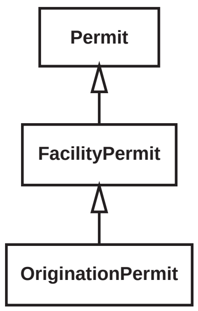

Figure 9.4 *The* `Permit` *hierarchy*

The `IndustrialFacility` constructor accepts an `OriginationPermit` and goes to the `PermitRepository` to get an associated permit; we use a method on `PermitRepository` that accepts an `OriginationPermit` and returns a `Permit`. If the repository finds the associated permit, it saves it to the `permit` field. If it doesn't, it saves the `OriginationPermit` to the `permit` field. We could create an interface for `OriginationPermit`, but that wouldn't do us any good. We would have to assign an `IOriginationPermit` to a `Permit` field, and that won't work. In Java, interfaces can't inherit from classes. The most obvious solution is to create interfaces all the way down and turn the `Permit` field into an `IPermit` field. Figure 9.5 shows what this would look like.

Yuck. That is a ridiculous amount of work, and I don t particularly like how the code ends up. Interfaces are great for breaking dependencies, but when we get to the point that we have nearly a one-to-one relationship between classes and interfaces, the design gets cluttered. Don t get me wrong: If our backs are against the wall, it would be fine to move toward this design, but if there are other possibilities, we should explore them. Fortunately, there are.

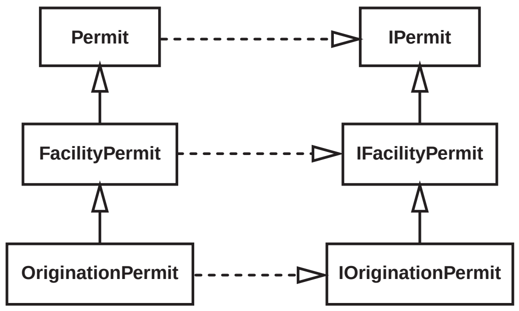

Figure 9.5 `Permit` *hierarchy with extract interfaces.*

Extract Interface (362) is just one way of breaking a dependency on a parameter. Sometimes it pays to ask why the dependency is bad. Sometimes creation is a pain. At other times, the parameter has a bad side effect. Maybe it talks to the file system or a database. At still other times, it just might take too long for its code to run. When we use *Extract Interface (362)*, we can get past all of these issues, but we do it by brutally severing the connection to a class. If only pieces of a class are problems, we can take another approach and sever only the connection to them.

Let's look closer at the `OriginationPermit` class. We don't want to use it in a test because it silently accesses a database when we tell it to validate itself:

```
public class OriginationPermit extends FacilityPermit
{
    ...
    public void validate() {
        // form connection to database
        ...
        // query for validation information
        ...
        // set the validation flag
        ...
        // close database
        ...
    }
}
```

We don't want to do this in a test: We'd have to make some fake entries in the database, and the DBA will get upset. We'd have to take him to lunch when he found out, and even then he'd still be upset. His job is hard enough as it is.

Another strategy that we can use is *Subclass and Override Method (401)*. We can make a class called `FakeOriginationPermit` that supplies methods that make it easy to change the validation flag. Then, in subclasses, we can override the `validate` method and set the validation flag any way that we need to while we are testing the `IndustrialFacility` class. Here is a good first test:

```
public void testHasPermits() {
    class AlwaysValidPermit extends FakeOriginationPermit
    {
        public void validate() {
            // set the validation flag
            becomeValid();
        }
    };

    Facility facility = new IndustrialFacility(Facility.HT_1, "b",
                                    new AlwaysValidPermit());
    assertTrue(facility.hasPermits());
}
```

In many languages, we can create classes "on the fly" like this in methods. Although I don't like to do it often in production code, it is very convenient when we are testing. We can make special cases very easily.

Subclass and Override Method (401) helps us break dependencies on parameters, but sometimes the factoring of methods in a class isn't ideal for it. We were lucky that the dependencies we didn't like were isolated in that `validate` method. In worse cases, they are intermingled with logic that we need, and we have to extract methods first. If we have a refactoring tool, that can be easy. If we don't, some of the techniques in Chapter 22, *I Need to Change a Monster Method and I Can't Write Tests for It*, might help.