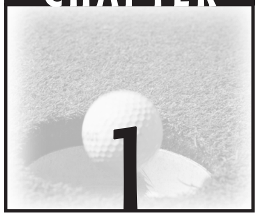
CHAPTER



1

Testing: Cactus and JUnit

With the advent of Extreme Programming (XP) and its emphasis on refactoring, unit testing has gained in popularity and exposure. In order to refactor anything, a good set of unit tests must be in place to make sure that current clients of the implementation will not be affected by the changes that are made. Many developers, as they embrace the XP approach, are suddenly "test infected" and writing all kinds of JUnit tests. Many developers who were doing unit testing with code in the main method of their Java classes are finding JUnit and Cactus to be a more thorough means to test their classes. This chapter is about what goes wrong when building a real-world test set for real-world applications with these tools.

Many pitfalls in unit tests come from the complexity of the components being tested or the complexity of the tests themselves. Also, the lack of assertions in the test code can cause problems. Without an assertion, a test just confirms that no exceptions were thrown. Although it is useful to know when exceptions are thrown, it is rarely enough. For example, not all unexpected state changes in a test subject will throw an exception. Developers shouldn't simply rely on printouts so that they can visually inspect the result of calling the tested code. While visual inspection is better than nothing, it's not nearly as useful as unit testing can be. This chapter shows several ways in which a lack of assertions shows up and provides strategies

MOCK OBJECT VERSUS "IN CONTAINER" TESTING

There are two ways to approach testing your server-side objects. They can be isolated from the containers in which they are intended to run and tested separately to ensure that the objects do what is expected of them. The other way is to build a framework that works with the container to allow your objects to be tested inside the container.

The first approach, called Mock Object testing (or the Mock Objects Pattern), is very effective at isolating the test subject. There is significant burden, though, in building and maintaining the Mock Objects that simulate the configuration and container objects for the test subject. They have to be built and maintained in order for the testing to be effective. Even though there is virtually no complexity to the actual Mock Objects, there is a lot of complexity in maintaining the large number of Mock Objects required to simulate the container.

Cactus takes the other approach and facilitates testing inside the container. Cactus gets between the test cases and the container and builds an environment for the test subject to be run in that uses the container-provided objects instead of Mock Objects. Both approaches are helpful in stamping out bugs.

to migrate existing tests (visual or not) to solid unit tests that assert the contract implied in the API being tested.

A quick word about the differences between JUnit and Cactus: JUnit tests run in the same JVM as the test subject whereas Cactus tests start in one JVM and are sent to the app server's JVM to be run. Cactus has a very clever means to do the sending to the remote machine. Just enough information is packaged so that the server side can find and execute the test. The package is sent via HTTP to one of the redirectors (ServletTestRedirector, FilterTestRedirector, or the JSPTestRedirector). The redirector then unpacks the info, finds the test class and method, and performs the test. Figure 1.1 represents this process.

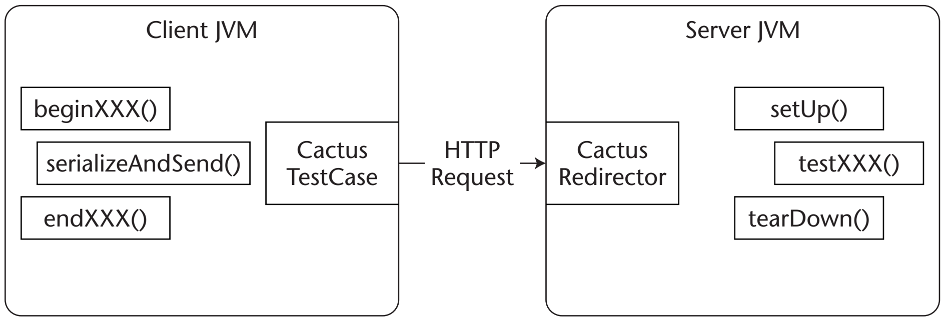


Figure 1.1 Cactus and test methods.

Misunderstanding the distributed nature of Cactus can lead to frustration if you are an old hand at building JUnit tests. Keep this in mind as you start to build Cactus tests.

Another thing to keep in mind as you build Cactus tests is that the redirector provides access to the container objects only on the server. Because the container objects are not available until the test is running on the server side, you cannot use container objects in the methods that are executed on the client side. For example, the config object in a ServletTest is not available in the beginXXX and endXXX methods, but you can use it in your setUp, textXXX, and tearDown methods.

JUnit has become the de facto standard unit testing framework. It is very simple to get started with it, and it has amazing flexibility. There are probably dozens, if not a couple of hundred, of extensions to JUnit available on the Web. This chapter focuses on JUnit (www.junit.org) and Cactus (www.jakarta.apache.org/cactus). Cactus allows "in container" testing of J2EE components. As stated earlier, this book assumes some experience with these tools.

Pitfall 1.1: No assert is the result of developers that do not realize that calling a method is not the same as testing it.

Pitfall 1.2: Unreasonable assert examines the tendency of developers new to unit testing to start asserting everything, even things that won't happen unless the JVM is not working properly.

Pitfall 1.3: Console-Based Testing addresses the problem of developers who get into the habit of using "System.out" to validate their applications. This method of testing is very haphazard and error-prone.

Pitfall 1.4: Unfocused Test Method is common to more experienced developers who get a little lazy about writing good tests and let the test become overly complex and hard to maintain.

Pitfall 1.5: Failure to Isolate Each Test is fixed using the setUp and tearDown methods defined in the JUnit framework. These methods allow each test to be run in an isolation that contains only the test subject and the required structure to support it.

Pitfall 1.6: Failure to Isolate Subject is related to the discussion of Mock Objects in the previous sidebar.

Pitfall 1.1: No Assert

This pitfall describes the tendency of developers new to unit testing to forget about asserts completely. New developers often assume that invoking the method is a sufficient test. They assume that if no exceptions are thrown when the method is called, then everything must be OK. Many bugs escape this kind of testing.

Here is some code for a simple addStrings method that returns the result of concatenating the value returned from the toString method of its two arguments. The current implementation should not be putting a space into the returned value, but it is. The initial test will not expose this bug because it is stuck in this pitfall; we will apply the solution to the test, though, and it will expose the bug.

```
public String appendTwoStrings(Object one, Object two) {
    StringBuffer buf = new StringBuffer(one.toString());
    buf.append(" ");
    buf.append(two.toString());
    return buf.toString();
}
```

Here is a sample test that simply invokes the method without really testing anything.

```
public void testAppendTwoStrings() throws Exception {
    myAppender.appendTwoStrings("one", "two");
}
```

This situation is typical of tests stuck in this pitfall. Even though it looks as if the appendTwoStrings method is tested, it is not. Users of the appendTwoStrings method have expectations of what the return value will be as a result of calling the method. And, in this case, the expectations will not be met. The API for a class is an implied contract for the users of the code. Whenever that contract is not met, the users of the code will see that failure as a bug. Unit tests should make sure that every unit of code performs as expected, that it fulfills the implied contract in the API. Unit tests that are stuck in this pitfall do not make sure that code is performing as expected, and they need to be fixed.

INTENT OF THE API

The "intent of the API" is what is documented or expected that the API will do with the inputs provided. It is also what the API will do to the internal state of the object on which the method is being called. For example, the *append* method on the StringBuffer class is documented to append the argument to its internal buffer such that the StringBuffer is longer by the length of the argument that is passed into the method. The internal state of the StringBuffer has changed, and nothing has happened to the argument. The test suite for StringBuffer should assert both "intents" of the StringBuffer API.

A test should make sure that the stated intentions of the API are met by asserting that what is expected to be true actually is. Another way to think of the intent of the API is that the API is like a contract between the clients that use the API and the provider of the API. The provider of the API is guaranteeing that the class will perform certain tasks, and the consumer is expecting those tasks to be performed. Formal ideas surrounding Design by Contract (DBC) are helpful in building tests for classes.

No assert is usually exposed at the point at which the application is passed over to the testing team, which tests the application by looking at the state changes that are occurring. As the testing team looks into the database to confirm that what was supposed to change did change, they will notice that the data is not changing as expected. As a result, many bug reports will be filed, and the bug fixes will more often than not be made in code that was tested with few asserts. Bug reports are no fun, especially when some effort was made to do unit testing. This situation will make it appear that unit testing added little value.

One of two things, laziness or lack of knowledge and experience, usually causes this pitfall. Everyone gets lazy from time to time. Developers are no exception, but it is important that we build good unit tests so that we can afford to be a little lazy. A good set of unit tests will expose bugs right when they are introduced, which is when they are easiest to fix. And because the bugs are easier to fix, we have less work.

Lack of knowledge and experience is fixed only through mentorship and experience. Over time, developers will begin to see how valuable unit tests are, especially if they have found, fixed, and prevented anyone else from seeing bugs in their code. You can encourage and teach good unit testing by doing periodic peer reviews with junior members. Peer reviews provide a great mechanism to mentor people, and if the senior people allow junior people to review their code, junior people will be able to see good examples on a regular basis.

TESTING FIRST

Many people in the JUnit community suggest that tests be written before the code that they are intended to test. The tests become almost a coded set of requirements for the test subject. This is a great habit to get into. The next time you are transitioning from design to development, try writing a few tests for the new code before implementing. When it comes time to use the code, you will thank yourself. The great benefit of testing first is that it forces the developer to focus on providing good APIs to future clients. The tests will expose nuances of what was expected to be true at design time versus what is really true on the ground in the code. And besides, if you write the tests first you will have a concrete gauge of when the class is done (that is, when it passes all the tests, it is done).

To stay out of this pitfall, you have to assert the intent stated in the API being tested. The intent of the API is what is expected to happen when the API is called. The intent is usually captured in the form of JavaDoc and the exception list that a method throws (sometimes referred to as the contract for the class). Unit tests should make sure that each piece of the API is doing what it should. As an example, if a method claims to throw an IllegalArgumentException when a null is passed, at least one test should assert that that exception is thrown when a null is passed.

Example

This example of No assert relies on a test for the contrived class called StringPair. Instances of StringPair will be used as keys in a map. An object that will be used as a key in a map must implement two methods: "equals" and hashCode. The two methods must be consistent, which means that if true is returned from the two objects involved in an equals comparison (that is, the receiver of the method call and the argument to the method), then hashCode must return the same value for both objects.

The StringPair class has two string properties, *right* and *left*. These two values are used in the equals and hashCode methods. To further complicate the subject, let's say that the StringPair class is used in a performance-sensitive environment and that it caches the hashCode so that it does not have to be recomputed each time. The hashCode should be reset to –1 when either the right or left value changes. This resetting behavior is crucial to the functioning of the StringPair class as a signal that the hashCode should be recomputed. The unit test here makes sure that the important methods on the StringPair class are called.

A good test for the StringPair class would assert that every intent described earlier is true (that hashCode and equals are consistent). The

JUnit test, however, is not good, as the test in Listing 1.1 does not assert anything in particular; in other words, this test case is trapped in this pitfall.

```
public class StringPairTest extends TestCase {
    private StringPair one = new StringPair("One", "Two");
    private StringPair oneA = new StringPair("One", "Two");
    private StringPair two = new StringPair("Three", "Four");
    private StringPair twoA = new StringPair("Three", "Four");

    public static Test suite() {
        return new TestSuite(StringPairTest.class);
    }

    public StringPairTest(String name) {
        super(name);
    }

    /**
     *  Test equals.
     */
    public void testEquals() throws Exception {
        one.equals(oneA);
        oneA.equals(one);
    }

    /**
     *  Test not equals.
     */
    public void testNotEquals() throws Exception {
        one.equals(two);
        two.equals(one);
    }

    /**
     *  Test hashCode.
     */
    public void testHashCode() throws Exception {
        one.hashCode();
        two.hashCode();
    }

    /**
     *  Test setting the values.
     */
    public void testSetValues() throws Exception {
        one.setRight("ROne");
        one.setLeft("LOne");
```

Listing 1.1 StringPairTest. *(continues)*

```
        two.setRight("RTwo");
        two.setLeft("LTwo");
    }

    /**
     *  Should throw an exception.
     */
    public void testNullPointerProtection() throws Exception {
        // since this will throw an exception the test will fail
        StringPair busted = new StringPair(null, "Four");
    }
}
```

Listing 1.1 *(continued)*

There are no asserts in the code for StringPairTest, so it is actually not testing very much. For example, take a look at the testSetValues method. All that happens is that the right and left property set methods are called. No check is made to make sure that the expected state changes happened on the StringPair instances. All this test is making sure of is that if valid strings are passed into the set methods (that is, not null) no exceptions are thrown. A lot of code is written in this way and then called test code. The StringPairTest test case is a classic example of this pitfall.

In this example, because the StringPair class is so simple, it might seem like overkill to put tests in place to make sure that equals and hashCode are performing as they should. Others, however, will be using this class and will expect it to function as advertised in its API. Which kind of class would you rather depend on in your code, one that is well tested (even when the code seems simple) or code that is not tested? A well-tested StringPair class can be used confidently. A poorly tested StringPair class that is tested only in integrated tests with the larger process will likely lead to much harder-to-find bugs. If StringPair is tested only through the Big Process test cases, then bugs in StringPair will be much harder to find because it cannot be stated with certainty that the bug is not in StringPair. The test needs to assert that the intent of the class as laid out in its API is actually being met, meaning that the hashCode is being reset when a value changes. If tests are in place that assert the intent of the StringPair API, then when bugs arise in the Big Process, they can be attributed confidently to something in the Big Process code.

INCREMENTAL TEST IMPROVEMENT

If a bug exposed in a higher-level test turns out to be in a lower-level component (as in the earlier StringPair, Big Process example), then use that occasion to improve the test of the lower-level component. Instead of just fixing the bug and moving on, write a test first that fails because of the bug. Then fix the bug so that the test passes. In the future, if someone attempts maintenance on the lower-level component, the test will help him or her avoid reintroducing the bug.

Solving Pitfall 1.1: Assert the Intent

The way to get your tests out of Pitfall 1.1: No Assert is to introduce asserts. The intent of the API needs to be asserted to make sure that the documented behavior is actually happening. Building tests that assert the intent is making sure that the contract that is implied by the API and its documentation is being met.

This solution is the primary solution to Pitfalls 1.1 and 1.2: Unreasonable assert. Note that sometimes there are no asserts because the test is stuck in Pitfall 1.3: Console-Based Testing. Review that pitfall for a more in-depth discussion of the issues surrounding using System.out as a test tool and a description of the solution to Pitfall 1.3, "System.out becomes assert."

Step-by-Step

1. Start the process with the simplest-to-test subject method.
   a. If there is an existing test method that calls the method, the solution will start there; otherwise, a new test method should be created.
2. Review the documentation for the method to identify the intent (or contract) of the method.
   a. Some methods have no documentation so you will have to review the implementation of the method looking for intent.
   b. Sometimes bugs in the documentation will be exposed in implementing this process.
   c. Statements about what state changes will happen or what actions will be taken are good candidates for tests.
3. For each intent stated in the documentation, assert that what is documented to happen really has happened.

4. Repeat this process for the more complex methods.
   a. Be careful not to do a bunch of pointless tests. Too many tests that are testing simple accessory methods can be another pitfall. Test the important API of the class first.
5. Run the tests, and debug any failures.

Example

To illustrate, the JUnit test for StringPair (discussed earlier) will be put through this solution so that the test is cleaned up and accurately tests the StringPair class so we can be confident in using the StringPair as it is documented. The code for StringPair with its JavaDoc comments that will be used to determine the intent of the API is found in Listing 1.2.

```
/**
 *  A class to provide a key into a map.
 */
public class StringPair {
    /**
     * <code>hashCode</code> is used to cache the hash code for
     * this class
     */
    private int hashCode = -1;
    /**
     * The right-hand side string.
     */
    private String right;
    /**
     * The left-hand side string.
     */
    private String left;

    /**
     * A simple constructor taking the right and left side strings.
     * @param right
     * @param left
     * @throws IllegalArgumentException if either <code>right</code> or
     * <code>left</code> are null.
     */
    public StringPair(String right, String left) {
        if (null == right || null == left) {
            throw new IllegalArgumentException(
                "Should not pass null "
                    + "right = "
                    + right
```

Listing 1.2 StringPair.

b 449156 Ch01.qxd 6/16/03 8:48 AM Page 11

```
                      + " left = "
                      + left);
            }
            this.right = right;
            this.left = left;
        }
        /**
         * Get the right side string.
         * @return String
         */
        public String getRight() {
            return right;
        }

        /**
         * Sets the right-hand side string, reinitializes the
         *  hashCode so that it is recalculated the next time
         *  hashCode  is called.
         * @param right
         */
        public void setRight(String right) {
            hashCode = -1;
            this.right = right;
        }

        /**
         * Returns the left-hand side string.
         * @return String
         */
        public String getLeft() {
            return left;
        }

        /**
         * Sets the left-hand side string, reinitializes the hashCode
         * so that it is recalculated the next time hashCode is called.
         * @param left
         */
        public void setLeft(String left) {
            hashCode = -1;
            this.left = left;
        }

        /**
         * Calculates the hashCode in a way consistent with equals.
         * @see java.lang.Object#hashCode()
         * @see java.util.Map
         */
        public int hashCode() {
```

Listing 1.2 *(continues)*

```
        if (-1 == hashCode) {
            hashCode = right.hashCode() ^ left.hashCode();
        }
        return hashCode;
    }

    /**
     * Implements the equals method consistent with the hashCode method.
     * @see java.lang.Object#equals(Object)
     * @see java.util.Map
     */
    public boolean equals(Object o) {
        boolean flag = false;
        if (o instanceof StringPair) {
            StringPair other = (StringPair) o;
            if (other.hashCode() == hashCode()) {
                if (right.equals(other.getRight())
                    && left.equals(other.getLeft())) {
                    flag = true;
                }
            }
        }
        return flag;
    }
}
```

Listing 1.2 *(continued)*

None of the tests in the original JUnit test class (Listing 1.2) specifically asserts anything. This kind of test code is deceiving because it looks as if there are tests in place, but there is no actual testing happening. With a test case like this one in place, StringPair looks as if it's fully tested when, in fact, the equals method (as tested in the testNotEquals method in the code listing) could be returning true when it should return false and vice versa. So how does the code get out of this pitfall? Let's apply the steps outlined previously to StringPairTest and solve this pitfall.

The first step is to start with the simplest-to-test method. The absolute simplest methods are the property accessors (the *get* methods). Although they are the easiest, they are not very important to test because there is no actual code, so let's move to the next method in complexity, the equals method. Because methods to test the equals method already exist, we can start there. Here is the original test code intended to test the equals method.

```
    /**
      *  Test equals.
      */
```

```
public void testEquals() throws Exception {
    one.equals(oneA);
    oneA.equals(one);
}

/**
 *  Test not equals.
 */
public void testNotEquals() throws Exception {
    one.equals(two);
    two.equals(one);
}
```

The next step to solving this pitfall is to review the intent of the API. The equals method is supposed to provide the typical equals behavior—that is, return true if the receiver and the argument should be considered the same object. The equals method also claims to be consistent with the hashCode method, meaning that if two objects are equal to each other then their hashCode will be the same. Both statements need to be tested. In order to test the two different aspects of the equals method we will introduce two methods: testEqualsReturn and testEqualsHashCodeConsistency (one is simply a rename from testEquals to testEqualsReturn). The third method, testNotEquals, simply uses asserts where there were none before. The code for the methods is shown here.

```
/**
  *  Test equals.
  */
 public void testEqualsReturn() throws Exception {
     assertTrue(one.equals(oneA));
     assertTrue(oneA.equals(one));
 }
```

Notice that the difference between testEquals and testEqualsReturn is very small; all we did was add two method calls. The two new method calls made all the difference. Now there is a test in place so that the computer can make sure that what is expected to happen is actually happening. In other words, the responsibility for making sure that the program is doing what it should has shifted from the programmer to the computer.

```
/**
  * Test equals & hashCode consistency.
  * @throws Exception
  */
 public void testEqualsHashCodeConsistency() throws Exception {
     assertTrue(one.equals(oneA));
```

```
        assertEquals(one.hashCode(), oneA.hashCode());
    }
   /**
     *  Test not equals.
     */
    public void testNotEquals() throws Exception {
        assertTrue(!one.equals(two));
        assertTrue(!two.equals(one));
    }
```

These two methods have assertions to make sure that the hashCode and equals methods are indeed doing what they are documented to do. The first method checks that the two methods are consistent with each other, and the next method checks that the equals is returning false when it should.

The final step in the process is to apply what we did before (that is, assert the intent of the API in the tests) to each of the methods in StringPair that need to be tested. The next method that needs to be tested would be the hashCode method, and there are many statements in the documentation of the class that are important to look at. Specifically, each time a set method is called (for example, setRight), the hashCode must be recalculated. A test should be written that makes sure the hashCode is different after a set method is called with a different value. Another interesting test that should probably be done is to test that the hashCode is the same after a set method is called with the same value that was there before. The last statement about hashCode that needs to be verified is in its consistency with equals. That aspect of the two methods is already tested in the code we saw earlier. Here is a list of methods that test the intent stated for hashCode.

```
   /**
     *  Test hashCode.
     */
    public void testHashCode() throws Exception {
        assertTrue(one.hashCode() == oneA.hashCode());
    }

    public void testHashCodeChanges() throws Exception {
        int initialHashCode = one.hashCode();
        one.setRight("Wowzer");
        assertTrue(initialHashCode != one.hashCode());
    }

    public void testHashCodeNoChanges() throws Exception {
        int initialHashCode = one.hashCode();
        one.setRight(one.getRight());
        assertTrue(initialHashCode == one.hashCode());
    }
```

The final aspect of the StringPair API and its old JUnit test that needs to be addressed is the passing in of *null* to the constructor. The constructor is supposed to throw an exception if null is passed in. The old test code just called the method with null and documented that the runner of the test should expect a failure there. Although that approach kind of works on very small projects, it is troublesome on bigger projects because no one knows which tests should succeed and which are expected to fail without looking at the comments in the code. It is far better to catch the exception and fail the test if the exception is not thrown. Code that addresses this issue is shown here.

```
/**
  *  Test the constructor's protection against null
  */
 public void testNullPointerProtection() throws Exception {
     try {
         StringPair busted = new StringPair(null, "Four");
         fail("The constructor should have thrown an exception");
     } catch(IllegalStateException e) {
         // just ignore the exception because we expect it
     }
 }
```

We have completed our reworking of the StringPairTest JUnit test to cover all the stated intents of the StringPair class. Now we should run the test to make sure that everything passes. If you go back and review the differences between the two sets of code, you will notice that they are almost the same except for the addition of the asserts. The important thing to remember is that a test without an assert is not a very good test. Applying the steps outlined in this solution will help you fix any code you have that is stuck in this pitfall.

Example 2: Cactus

In this next example, we apply this solution to Cactus tests that are trapped in Pitfall 1.1: No assert. The steps to solve the pitfall do not change, but some of the details of Cactus tests need to be kept in mind while applying the steps. In particular, there is more API to review and assert, and each EJB has at least a home interface and a bean interface (local or remote and sometimes both). Also, because the code is running inside the application server, the test invocation will go through the container objects to get to your code. Because the container code is generated from your deployment descriptor, there are aspects of the descriptor that contribute to the contract of the API. For example, if it is important for one of your methods to participate in an existing transaction but not start one, then you might want to write a test that makes sure of that (that is, invoke the method without an existing transaction). Because there is so much detail (parts of the intent or contract) for each EJB method, the typical Cactus test will have a few asserts instead of one or two like the typical JUnit tests.

The Shopping Cart Session Bean (part of the Petstore demo application from the Sun Blueprints group) will be tested. The initial test is trapped in this pitfall and will be cleaned up so that it is not trapped anymore. The Shopping Cart Session Bean does all the things that a typical shopping cart does, and it uses methods to add and remove items. The quantity of a particular item can be updated, and a subtotal of the items currently in the cart can be calculated. The code in Listing 1.3 shows the interface for the cart that will be tested (all the comments have been removed for brevity; we will review the intent as we review the tests).

Listing 1.4 is the initial test class that is stuck in this pitfall. The only thing this test is really testing is that the methods can be called. While that might be a good test for the application server, the cart is deployed, and it is not what this test should be testing.

```
/**
 * This interface provides methods to add an item to the
 * shopping cart, delete an item from the shopping cart,
 * and update item quantities in the shopping cart.
 */
public interface ShoppingCartLocal extends EJBLocalObject {
    /*
     * Methods to update the state of shopping cart.
     */
    public void addItem(String itemID);
    public void setLocale(Locale locale);
    public Collection getItems();
    public void deleteItem(String itemID);
    public void updateItemQuantity(String itemID, int newQty);
    public Double getSubTotal();
    public Integer getCount();
    public void empty();
}
/**
 * The home interface for the shopping cart.
 */
public interface ShoppingCartLocalHome extends EJBLocalHome {
    public ShoppingCartLocal create() throws CreateException;
}
```

Listing 1.3 ShoppingCart interface.

```
public class ShoppingCartCactusTest extends ServletTestCase {
... constructor and main
    public void testDeployed() throws Exception {
        InitialContext ic = new InitialContext();
        // if it's not deployed this will throw an exception
        // this is a pointless test because the rest of the tests
        // will expose any problem in deploying the bean.
        ShoppingCartLocalHome sHome = (ShoppingCartLocalHome)ic.
            lookup(JNDI_NAME);
    }

   public void testCreate() throws Exception {
        InitialContext ic = new InitialContext();
        ShoppingCartLocalHome sHome = (ShoppingCartLocalHome)ic.
            lookup(JNDI_NAME);
        // assume it is deployed and try a create
        // if something goes wrong this will throw an
        // exception
        ShoppingCartLocal cart = sHome.create();
    }

    public void testGetItems() throws Exception {
        InitialContext ic = new InitialContext();
        ShoppingCartLocalHome sHome = (ShoppingCartLocalHome)ic.
            lookup(JNDI_NAME);
        ShoppingCartLocal cart = sHome.create();
        // assume we can create and call the getItems method
        cart.getItems();
    }

    public void testUpdateItemQuantity() throws Exception {
        String itemId = "EST-3";
        InitialContext ic = new InitialContext();
        ShoppingCartLocalHome sHome = (ShoppingCartLocalHome)ic.
            lookup(JNDI_NAME);
        ShoppingCartLocal cart = sHome.create();
        cart.addItem(itemId);
        cart.updateItemQuantity(itemId, 2);
        Collection items = cart.getItems();
        CartItem item = (CartItem)items.iterator().next();
    }
... other tests here
}
```

Listing 1.4 ShoppingCartCactusTest.

The first indication that this test is stuck in this pitfall is that there are no asserts in any of the tests. The tests are counting on exceptions being thrown if something is not correct. But, as discussed earlier, a test that does not assert the intent of the API is not actually testing anything of value.

As with a JUnit test, start the solution by tackling the simplest method to test first. The simplest method on this bean is the getCount method. The existing test does not have a test for this method so we will add one. The code for the new method is listed here.

```
public void testGetCount() throws Exception {
    // get the shopping cart home
    InitialContext ic = new InitialContext();
    ShoppingCartLocalHome sHome =
        (ShoppingCartLocalHome) ic.lookup(JNDI_NAME);
    // create the cart
    ShoppingCartLocal cart = sHome.create();
    // add an item
    cart.addItem("EST-3");
    // update the quantity of that item to 4
    cart.updateItemQuantity("EST-3", 4);
    // assert that what we put in is there
    assertEquals(cart.getCount().intValue(), 1);
}
```

Notice that the test asserts what is expected. The quantity of items that should be in the cart is one, and that is asserted through the assertEquals method call. This test covers the stated intent of the getCount method so the next step in the process is to improve the tests for the rest of the intent of the API.

The next method to test is the updateQuantity method. The testUpdateItemQuantity test method will be updated by applying the steps to the solution as before. The original test method code is listed here.

```
public void testUpdateItemQuantity() throws Exception {
    String itemId = "EST-1";
    InitialContext ic = new InitialContext();
    ShoppingCartLocalHome sHome = (ShoppingCartLocalHome)ic.
        lookup(JNDI_NAME);
    ShoppingCartLocal cart = sHome.create();
    cart.addItem(itemId);
    cart.updateItemQuantity(itemId, 2);
}
```

Notice that there are no asserts here; the test is not doing what it should in making sure the quantity has been updated as expected. The next step is to identify the intent of the API. The updateQuantity method is supposed to find the item, identified by the itemID argument, and update the quantity of that item in the cart to the amount specified by the second argument. In order to assert that the update has happened, the test must get the list of

CartItems from the cart and look at the quantities stored there. The new test code is listed here.

```
public void testUpdateItemQuantity () throws Exception {
    String itemId = "EST-";
    InitialContext ic = new InitialContext();
    ShoppingCartLocalHome sHome = (ShoppingCartLocalHome)ic.
        lookup(JNDI_NAME);
    ShoppingCartLocal cart = sHome.create();
    cart.addItem(itemId);
    cart.updateItemQuantity(itemId, 2);
    Collection items = cart.getItems();
    assertNotNull("Items should not be null", items);
    assertEquals("There should be exactly one item", 1,
                 items.size());
    // the direct call of next here is ok because we just checked
    // that the count is 1 in the previous line
    CartItem item = (CartItem)items.iterator().next();
    assertEquals("The item id is wrong", itemId, item.getItemId());
    assertEquals("The quantity of " + itemId + " is wrong",
                 2, item.getQuantity());
}
```

This test method is much more complete in asserting the state changes that should have occurred because of the change to the quantity of the item.

If you are new to Cactus testing, note that the test is interacting with the local home interface for the shopping cart. Remember that the test begins in a client JVM but is routed into the server's JVM by the specialized test classes and the redirectors in the Cactus framework. As discussed previously, Cactus runs inside the container in the application server so that you do not have to have Mock Objects for JNDI and the other server-side services provided by the application server.

TESTING EXISTING CODE

This last test on updating the quantity of a particular item in the cart is illustrative of the difficulty of working with existing code and trying to write tests for that code. Because the cart has no way of getting the quantity for a particular item other than through the CartItem class, there is no way to get at the value from outside. A better unit test would be interacting only with the bean and not with any of the helper classes like CartItem. The only way to accomplish a strict unit test, though, would be to modify the ShoppingCart bean. So this test makes due with what is available and manages to test the intent of the API.

Pitfall 1.2: Unreasonable Assert

Unreasonable assert is the tendency of inexperienced developers to assert everything that can be imagined. Often developers new to unit testing will take one of two tracks: They will have read about asserts before and thus recognize the concept when they see it in JUnit, or they will not have heard of asserts before and skip over them. The first kind of developer typically gets trapped in this pitfall. Tests end up bloated with a lot of asserts that basically just make sure that the JVM is working. Most JUnit tests will have only one assert; some tests will have a couple of asserts. If a JUnit test has several asserts, then it is very probable that either the test needs to be pulled into many tests or the underlying class needs to be simplified so that a single method is not doing so much. In Cactus, the number of asserts is usually a few instead of one due to the nature of testing in a distributed environment.

CACTUS AND UNIT TESTING

The strictest definition of unit testing requires that each test subject and each test run in complete isolation from all the supporting classes on which the test subject relies. This definition is a guide but is hard to accomplish in a J2EE environment. For example, in order to strictly unit test an entity EJB, the test harness must have Mock Objects implemented for each of the container-generated classes that support that bean. The harness must also provide mock implementations of all the container services (transactions, security, and so on).

Cactus takes a pragmatic approach to this debate and makes tests pseudo-units (according to the strict definition outlined earlier) that use the actual container-generated classes and the services provided by the container. In a typical test of an EJB there is usually more than one thing that can and should be asserted. In an earlier example, the Shopping Cart Session Bean that returns the list of items in the cart was tested. Instead of strictly applying the unit test definition, the test asserted several things that should be true about that collection of objects in one test. This approach, while not purely a unit test, provides a practical approach to testing EJBs.

Each invocation of the getItems method takes considerable time to set up and execute; if only one small part of the intent of that method were executed with each test, the whole test suite would take too long to run. If the tests take too long to run, developers will often stop running them. The best tests in the world are useless if they are never run. The approach outlined here takes a pragmatic approach to testing.

Just like a lack of asserts, many pointless asserts do not provide enough unit testing. When the test team tests the application, many bugs will be exposed that should have or could have been caught by good unit testing. Other symptoms include a reluctance to maintain the tests because they do not appear to add much value. Ultimately, spending lots of time and energy building tests that make sure the JVM is working wastes effort.

Usually this pitfall takes the form of tests that assert things that are almost impossible. For example, here is some code that ensures that the JVM is functioning properly.

```
     public void testRidiculous() throws Exception {
         StringPair subject = new StringPair("One", "Two");
         assertNotNull(subject);
    }
```

This test is making sure that the constructor does not return null. If the JVM is working, it's not possible to get null back from a constructor. If the JVM is unable to allocate memory for the new object, it will throw an OutOfMemoryException. If something else goes wrong with constructing the object, an exception will be thrown. None of the things that should be tested in this method is being tested. For example, the two strings that are passed into the constructor are supposed to initialize the right and left properties on the StringPair object. The test should assert this intent of the StringPair API.

Example

This JUnit test case is intended to test the StringPair class, but it is not asserting anything about the API of the class. The actual testing needs of the StringPair class are being missed in all the noise of the useless asserts, as you can see in Listing 1.5. Notice that there are lots of asserts, but none of them is making sure that what the class should be doing is actually being done. The test subject is the same StringPair class that was tested in Pitfall 1.1: No assert.

```
public class StringPairTest extends TestCase {
    StringPair subject = new StringPair("One", "Two");

    public static Test suite() {
        return new TestSuite(StringPairTest.class);
```

Listing 1.5 StringPairTest. *(continues)*

```
    }

    public StringPairTest(String name) {
        super(name);
    }

    public void testConstructor() throws Exception {
        StringPair localSubject = new StringPair("One", "Two");
        assertNotNull(localSubject);
        assertTrue(null != localSubject.getRight());
        assertTrue(null != localSubject.getLeft());
    }

    public void testRightValue() throws Exception {
        subject.setRight("Three");
        assertTrue(null != subject.getRight());
    }

    public void testLeftValue() throws Exception {
        subject.setLeft("aValue");
        assertTrue(null != subject.getLeft());
    }

    public void testHashCode() throws Exception {
        int hash = subject.hashCode();
        assertTrue(subject.hashCode() == hash);
    }

    public void testEquals() throws Exception {
        assertEquals(subject, subject);
    }

    public void setUp() throws Exception {
    }
    public void tearDown() throws Exception {
    }
}
```

Listing 1.5 *(continued)*

The most important thing for the test writer to know is the intent of the API; the writer then should make sure the tests are asserting that. In this test case, nothing much is being asserted. For unit tests to be useful, they need to ensure that the expected behavior is what is actually happening.

Solving Pitfall 1.2: Assert the Intent

Just as with Pitfall 1.1: No assert, the solution to this pitfall is asserting only the intent of the test subject. Developers of the tests should know the intent of the API being tested and write tests that make sure that intent is met. In addition to adding asserts that make sure the API is acting as it should, many asserts (as described earlier in the solution to Pitfall 1.1) will have to be removed to fix the code. While applying the steps ask the question, “What aspect of the API does this assert test?” for each assert that is in the test. If there is no clear answer, then remove the assert. The new asserts will be added just as they were to solve Pitfall 1.1.

Pitfall 1.3: Console-Based Testing

Pitfall 1.3: Console-Based Testing is the practice of using System.out. println in the test code and then visually inspecting the output to validate that the test subject is doing what it should. Developers new to unit testing typically don't know how to write good test code and will not have enough experience to write good assertions. Often in these cases, the developer knows something more needs to be done but does not know quite what to do, so he or she decides that the results should be printed to the console. The developer is then forced to inspect the output visually to determine if the test succeeded.

Most of the time, this common practice leads to a numb stare at the console as untold numbers of lines stream by. After some experience with this blank-stare syndrome, developers often resort to putting strange leading and trailing characters into the logs so that the output in question catches the eye. As we all know, there are just not enough special characters on a keyboard to make every interesting thing eye-catching. Besides, with enough strange characters streaming by, nothing will catch the eye. Usually when the code is transitioned to a test team, they will find many bugs that should and could have been discovered via the unit tests.

The typical way to get trapped in this pitfall is to put lots of logging messages (for example, System.out.println) into the code, manually stimulate the application either through clicking around on the UI or writing simple client programs that invoke the remote API, and then review the output for correct values. For some reason, this practice seems to give most of us a warm fuzzy feeling that we have done what we can to make sure everything is working as it should. It's strange that developers would think this way; we are writing software to automate some part of the user's task, but we are unwilling to use a computer to automate some of our task. This is like a bad habit that is very hard to break.

The form this pitfall usually takes is big test methods with little else but println in them. In these test methods, there are many lines of test code, but the value of that code is small. A typical test method trapped in this pitfall usually looks something like this.

```
public void testFindCustomerByName() throws Exception {
    CustomerPO customer = new CustomerPO();
    customer.setFirstName("Al");
    customer.setLastName("Capone");
    // print out the presentation object
    System.out.println("Al Capone = " + customer);
    InitialContext ic = new InitialContext();
```

```
        CustomerSearchHome csHome =

(CustomerSearchHome)ic.lookup(JNDINames.CUSTOMER_SEARCH_NAME);
        CustomerSearch search = csHome.create();
        // pass null as the address because the address is not
        // important for this search
        CustomerPO alCapone = search.findCustomer(customer,  null);
        // print out the first and last name
        System.out.println("Found Al Capone = " + alCapone);
    }
```

This test would let the developer know that the values are what they should be, but imagine 200 tests running that look something like this. Now imagine that this is the simplest of the 200. Clearly, with all that output, any bugs or unexpected behavior will be hard to spot.

The best way to avoid being stuck in this pitfall is to develop the discipline of removing println code in test code as soon as possible. A basic rule of thumb is that wherever a call to System.out.println is tempting, put an assert statement instead.

Example

The test subject for this example is a cache. The cache keeps two lists of customer objects, one ordered for the presentation layer and another map that is keyed on the customer's social security number for quick lookup during processing. The CustomerCache is shown in Listing 1.6.

```
public class CachedCustomerList {
    private List customers = new ArrayList();
    private Map customerMap = new HashMap();

    /**
     * Add newCustomer to both collections.
     * @param newCustomer
     */
    public void addCustomer(CustomerPO newCustomer) {
        customers.add(newCustomer);
        customerMap.put(newCustomer.getSSN(), newCustomer);
    }

    /**
     * Return the list of customers. This list should be in the same
     * order that the customer objects were placed into the cache.
     * @return List
     */
```

Listing 1.6 CachedCustomerList. *(continues)*

```
    public List getCustomers() {
        return customers;
    }

    /**
     * Return the customer identified by <code>ssn</code>.
     * This method uses the hash map for quick lookup.
     * @param ssn
     * @return CustomerPO
     */
    public CustomerPO getCustomer(String ssn) {
        return (CustomerPO)customerMap.get(ssn);
    }

     /**
     * Returns true if the customer is in this cache, false
     * otherwise.

     * @param testCustomer
     * @return boolean
     */
    public boolean containsCustomer(CustomerPO testCustomer) {
        return customerMap.get(testCustomer.getSSN()) != null;
    }
}
```

Listing 1.6 *(continued)*

The intent of this API is captured in its comments. The JUnit test in Listing 1.7 is stuck in Console-Based Testing so there is very little assurance that the intent of the API is being met.

```
public class CachedCustomerListTest extends TestCase {
    private CachedCustomerList cache = new CachedCustomerList();

    public CachedCustomerListTest(String name) {
        super(name);
    }

    public static Test suite() {
        return new TestSuite(CachedCustomerListTest.class);
    }

     /**
     * Test getting the customers from the cache.
     */
    public void testGetCustomers() throws Exception {
```

Listing 1.7 CachedCustomerListTest.

b 449156 Ch01.qxd 6/16/03 8:48 AM Page 27

```
        List customers = cache.getCustomers();
        // there is no check here (except visually by the developer)
        // of what the count is, and it's not altogether apparent
        System.out.println("Count = " + customers.size());
        System.out.println("Customers = " + customers);
    }

    /**
     * Test getting the customer map from the cache.
     */
    public void testCustomerMap() {
        CustomerPO one = cache.getCustomer(ONE_SSN);
        System.out.println("Customer = " + one);
        System.out.println("ssn should be " + ONE_SSN);
    }

    private final String ONE_SSN = "999-00-8888";
    private final String TWO_SSN = "888-00-9999";
    private final String THREE_SSN = "333-00-3456";

    public void setUp() throws Exception {
        // customer one
        CustomerPO cust = new CustomerPO();
        cust.setFirstName("Willy");
        cust.setLastName("Wonka");
        cust.setSSN(ONE_SSN);
        cache.addCustomer(cust);
        // customer two
        cust = new CustomerPO();
        cust.setFirstName("Mogilla");
        cust.setLastName("Gorrila");
        cust.setSSN(TWO_SSN);
        cache.addCustomer(cust);
        // customer three
        cust = new CustomerPO();
        cust.setFirstName("Speed");
        cust.setLastName("Racer");
        cust.setSSN(THREE_SSN);
        cache.addCustomer(cust);
    }
}
```

Listing 1.7 *(continued)*

Take a look at the testGetCustomers method, which prints out the customer collection. It is up to the test runner to parse the output and determine if the subject is performing as expected. Also notice that it is hard to tell exactly what is being tested in this method. Is the intent to test that the size of the returned collection is proper, or is the test intended to test the objects in the list?

Keep in mind that building tests with lots of output is OK as long as the intent of the API is being asserted. If you prefer reviewing logged output to stepping through the code in a debugger, then feel free to put in println, but make sure that you also assert the intent of the API. Each time a println is added, try to add an assert as well. If you are not sure what to assert, then think through what intent of the API the printout is supposed to help you confirm. As the println code is removed from the test code, the assert statements remain and continue to ensure silently that the test subject is behaving as expected. If the expected value is not known, then add the assert statement as soon as the expected value is known. A good approach to adding asserts when the values are not known is to run the test, then assert that the value is whatever shows up on the console. As the data varies, the test will expose misunderstandings about the API, leading to better assertions.

Solving Pitfall 1.3: System.out Becomes Assert

The solution System.out Becomes assert is the best means to solve this pitfall. (A related solution, assert the intent from Pitfall 1.1: No assert, can also be helpful.) Tests trapped in this pitfall usually have the same issues that code trapped in Pitfall 1.1 do, namely the intent of the API is not asserted. As you are working through this solution references will be made to the intent of the API being put into the asserts in places where the intent of the printout is not clear. In those cases it might be helpful to refer back to that solution for more context. The steps to that solution can be found earlier in this chapter as the solution to Pitfall 1.1.

Many developers become console blind because of having too many debugging statements in their code. As the program runs, debugging statements come spraying out of the console as fast as the console can write text. The same thing happens with test code that is stuck in this pitfall. As the printouts come streaming by on the console, the developer is forced to pay very close attention and then must be able to look through all the output to make sure that what was expected is what actually happened. There is no way that a developer will catch everything in a long stream of printouts, and a developer cannot possibly provide as much detail checking as a program can. With asserts in place, however, the test can actually make sure that whatever is being looked for in the output is found. And, as an added bonus, nothing will show up on the console unless there is a failure. This approach leads to more repeatable and reliable tests. Using asserts allows the computer to review the state of the test subject and make sure that the expected state of the subject (that is, what is documented in the API) is the actual state. When tests rely on the output to the console and a human to review that output, there are often a lot of missed bugs.

Step-by-Step

1. Start with the simplest existing test method.
2. For each System.out.println, try to discern what the intent of the output is.
   a. This is absolutely the hardest part of this solution. Sometimes it's just not apparent why some piece of data is printed out or what the value should be.
   b. There are usually two flavors of printouts—one is just to see what is there and to check if an object is null; the other is to print the expected value and the actual value so that they can be compared. The first case is harder to convert to an assert; the second is easier.
3. For each simple review discovered in the second step (an "I just want to see it" printout), place an assertNotNull.
4. For each println with a value to compare use an assertEquals.
   a. Typically, this shows up as a printout of the expected value, followed by the value derived in the test.
5. Review the intent of the API being tested with these changes, and look for unchecked aspects that should be tested.
6. Deploy and test.
7. Repeat these steps for each test that is trapped in the pitfall.
   a. Often when applying this solution you will notice many pieces of the API that are not being tested. Add tests to cover the missing pieces while applying this solution.

Example

In this example, the CachedCustomerList JUnit test will be fixed so that it is no longer trapped in the pitfall. The first step in applying this solution is to find the simplest existing test method to start the cleanup on. Going back to the CachedCustomerListTest listing earlier, the simplest test method is testCustomerMap. The code for that method is listed here.

```
/**
* Test getting the customer map from the cache.
```

```
     */
    public void testCustomerMap() {
        CustomerPO one = cache.getCustomer(ONE_SSN);
        System.out.println("Customer = " + one);
        System.out.println("ssn should be " + ONE_SSN);
        }
```

The next step is to figure out what intent is supposed to be tested by examining the values that are being printed out. In the two printouts in testCustomerMap, it is fairly straightforward to see what is intended to be tested. Because the customer with ONE_SSN is being asked for, that is what should be returned.

The next step in the solution is to provide an assert for each intent that is being tested. In the current code for testCustomerMap there are two printouts. It is not clear what the first printout is for, but we will assume it is there just to make sure the value is not null (System.out.println ("Customer = " + one)). The other printout is giving a value to be compared. This method should have one assertNotNull call and one assertTrue, so that the code looks like this.

```
public void testCustomerMap() {
        CustomerPO one = cache.getCustomer(ONE_SSN);
        assertNotNull("The customer with ssn "
                + ONE_SSN + " should not be null", one);
        assertEquals("The ssn should be " + ONE_SSN,
                     ONE_SSN, one.getSSN());
    }
```

On the first pass at inserting the asserts, leave the printouts in place in case the asserts fail. If the asserts do fail, you just might have uncovered a long-standing bug that escaped the output review testing that was happening.

The next step is to check for other untested intent. For each intent discovered that is not being tested, insert a new test method to cover that intent. The intent of the getCustomer API is fully tested so we should move on to the next method. The method to test the list of customers is next. The existing test method is listed here.

```
    /**
     * Test getting the customers from the cache.
     */
    public void testGetCustomers() throws Exception {
        List customers = cache.getCustomers();
        // there is no check here (except visually by the developer)
        // of what the count is, and it's not altogether apparent
        System.out.println("Count = " + customers.size());
        System.out.println("Customers = " + customers);
    }
```

The next step is to identify the reason for the printout—that is, what intent of the API is being reviewed. The first printout is showing what the actual size is, but there is no reference to the expected size. For this part of the test, though, we can easily know the expected size by looking at the setUp method to see how many customers were put into the cache. The other printout is not obvious. Is the intent to look at the content of the collection or just another way to see the length of the list, or is the test runner supposed to be looking at the order of the customer objects to make sure it is correct? Because we cannot tell exactly what the intent of the printouts was, we just press on and assert the intent of the API.

```
  public void testCustomerSize() throws Exception {
        List customers = cache.getCustomers();
        // Check the size of the list
        assertEquals(
            "There should be 3 customers",
            3,
            customers.size());
    }

    public void testCustomerOrder() throws Exception {
        List customers = cache.getCustomers();
        CustomerPO one = (CustomerPO)customers.get(0);
        CustomerPO two = (CustomerPO)customers.get(1);
        CustomerPO three = (CustomerPO)customers.get(2);
        assertEquals(ONE_SSN, one.getSSN());
        assertEquals(TWO_SSN, two.getSSN());
        assertEquals(THREE_SSN, three.getSSN());
    }
```

The name of the method testGetCustomers was changed to testCustomerSize in keeping with its new focus. A new test method was also added to assert that the order of the customer objects is correct.

The final step is to repeat the process for the rest of the methods in the test and to continue to review the intent of the test subject to make sure everything that should be tested is being tested.

Example 2: Cactus

This example tests the Populate servlet from Petstore. The Populate servlet takes data from an XML file and puts it into the database so there is a reference set of data loaded. The servlet takes initialization parameters from the web.xml file and uses them to find the data and load the database. The servlet is invoked from the welcome page after the Petstore is first installed. Listing 1.8 illustrates the initial test code to invoke the loading.

The first step to make this code better is to find out what is the intent of the test. This can be very hard if there are no comments in the test. Often, the source code for the subject must be reviewed to be able to discover the actual expected behavior. The PopulateServlet reads an XML file and pipes the data into a database; then, on success, it redirects the user to the main page (or the page on which the user clicked the Sign On button).

In the endSimplePopulate test method, the entire response is being printed. It is impossible to discern intent from that printout; there is just too

```
public class PopulateServletCactusTest extends ServletTestCase {
    private PopulateServlet subject = null;

... more tests and JUnit required methods here

    public void beginSimplePopulate(WebRequest theRequest)
        throws Exception {
        // set the parameters
        theRequest.addParameter("success", "//petstore/main.screen");
        theRequest.addParameter("forcefully", "false");
    }

    public void testSimplePopulate() throws Exception {
        try {
            subject.init(config);
            subject.doPost(request, response);
        } catch(ServletException se) {
            System.out.println("This should not happen, an " +
                               "exception should not be thrown " +
                               se.getMessage());
        }
    }

    public void endSimplePopulate(WebResponse theResponse)
        throws Exception {
        System.out.println("theResponse = " +
                           theResponse.getText());
    }
... more tests here
    public void setUp() throws Exception {
        subject = new PopulateServlet();
    }
}
```

Listing 1.8 PopulateServletCactusTest.

much there, so instead the method should assert that the intent of the post method has been achieved. A couple of things should be checked: The title of the response should be checked to make sure that the store-front is being returned, and the database should be checked to make sure that the data is loaded properly. To test the title of the response it's best to use the HttpUnit integration in Cactus. The HttpUnit framework provides many simple-to-use APIs that make finding titles, buttons, and so on simpler. The endSimplePopulate method would change to look like this.

```
public void endSimplePopulate(com.meterware.httpunit.WebResponse
                              theResponse) throws Exception {
    assertEquals("Welcome to the BluePrints Petstore",
                 theResponse.getTitle());
}
```

Checking the database is a bit more involved. The test needs to fetch some data from the database and assert that what is expected is what is there. To do that, a bit of JDBC code should be added to the test; specifically, there needs to be code to get a connection, execute a select statement, retrieve the results, and clean up. In this test, one method is written to perform all that and return what was found in a HashMap keyed on the column name and strings as values.

```
public void testSimplePopulate() throws Exception {
    try {
        // init the subject
        subject.init(config);
        subject.doPost(request, response);
    } catch(ServletException se) {
        fail("A servlet exception should not be thrown: " +
             se.getMessage());
    }
    String query = "select * from item where itemid = 'EST-1'";
    // get the data from the item table
    HashMap data = fetchData(query);
    // assert that it's what is expected
    assertEquals("FI-SW-01", (String)data.get("productid"));
    query = "select * from item_detail where itemid = 'EST-1'";
    // get some of the details
    data = fetchData(query);
    // assert the values
    assertEquals("16.50", data.get("listprice"));
}
```

This test does not assert everything that could be asserted about the data that was imported, but it does at least check part of it. Exhaustive tests are often more trouble than they are worth. It is often better to focus on testing the breadth of the API first.

Pitfall 1.4: Unfocused Test Method

Pitfall 1.4: Unfocused Test Method describes test methods that are unfocused in nature and tend to become unwieldy as they grow to test more and more of the API. It usually comes from experienced developers who have become lazy about building the setUp code for the test, so they cram all the testing they can into one large, complex test method. While this sometimes results in a well-tested subject, the tests become unwieldy and are less likely to be maintained over time.

The symptoms of this pitfall usually involve reluctance to modify a test method because it is too big. If during a typical maintenance cycle on the subject of a test you are tempted to get rid of the test because it's too large or complex, rather than update it, then the test is probably stuck in this pitfall. Keep in mind that as the complexity of the test code grows, so does the time taken to maintain the tests, which can lead to the tests being abandoned. Another symptom is the time it takes to execute each test method. As a test method attempts to test more, the time it takes for that method to execute grows. As the execution time grows, developers will run the tests less often, eventually resulting in the tests not being run at all. Tests that are not run are not very useful.

This pitfall usually takes the form of a few test methods that grow to many lines of code over time. The tests grow to test the entire API of the subject in only a few test methods. Here is an example of a test method stuck in this pitfall.

```
public void testCustomerAPI() throws Exception {
    InitialContext ic = new InitialContext();
    UserTransaction ut = (UserTransaction)ic.
        lookup(USER_TRANS_REF);
    ut.begin();
    CustomerHome cHome = (CustomerHome)ic.lookup(CUSTOMER_REF);
    Customer customer = cHome.findByPrimaryKey(new Long(14));
    assertEquals(new Long(14), customer.getId());
    // this should be in an address test method
    assertEquals("14 Mulberry Lane", customer.getAddress().
                getStreet());
    Collection invoices = customer.getInvoices();
    assertEquals(3, invoices.size());
    Invoice inv = (Invoice)invoices.iterator().next();
    // this should be in an invoice test method
    assertTrue(!(new BigDecimal(0.0)).equals(inv.getOrderTotal()));
    assertEquals("Inigo", customer.getFirstName());
```

```
        assertEquals("Montoya", customer.getLastName());
        assertEquals("555-33-4455", customer.getSsn());
        customer.setFirstName("Binigo");
        assertEquals("Binigo", customer.getFirstName());
        ut.commit();
    }
```

This method is testing several aspects of the Customer API as well as some of the Invoice API. By contrast, a good test method typically tests one method or aspect of an API. Some methods are quite complex, and a test method would have to be very large to test the whole intent of a single piece of the API. In cases like that, there are two choices: Refactor the subject so that its API is easier to test, or build several tests to cover all the different aspects of the complex method. It is usually better to refactor the subject (or, for our purposes, find a solution to the pitfall), but sometimes the source is not easy to change and the test has to accommodate.

Example

The test subject in this example is a Customer Entity Bean. The bean maintains information about customers—their name, their address, and other personal data The bean has two container-managed relationships: a one-to-one relationship with the customer's billing address and a one-to-many relationship with the list of invoices describing the orders the customer has placed. The customer interface is demonstrated in Listing 1.9; we will discuss the intent of particular pieces of this API later in the solution.

```
public interface Customer extends EJBLocalObject {
    public Long getId();

    public String getFirstName();
    public void setFirstName(String param);

    public String getLastName();
    public void setLastName(String param);

    public String getFullName();

    public String getSsn();
    public void setSsn(String param);

    public Address getAddress();
```

Listing 1.9 Customer bean API. *(continues)*

```
    public void setAddress(Address param);

    public void addInvoice(Invoice invoice)
        throws InvoiceException;
    public void removeInvoice(Invoice invoice)
        throws InvoiceException;
    public Collection getInvoices();
    public void setInvoices(Collection invoices);
}
public interface CustomerHome extends EJBLocalHome {
    public Customer create() throws CreateException;

    public Customer lookupCustomer(CustomerPO customer,
                                   AddressPO address)
        throws FinderException;

    public Customer findByPrimaryKey(Long pk) throws FinderException;
}
```

Listing 1.9 *(continued)*

The Customer bean API is tested by the following test class (Listing 1.10). There is only one test method, and it is very unfocused.

```
public class CustomerCactusTest extends ServletTestCase {
    private final String USER_TRANS_REF =
        "java:comp/UserTransaction";
    private final String CUSTOMER_REF = "invoice.Customer";
    private final String ADDRESS_REF = "invoice.Address";

    private UserTransaction ut = null;
    private Customer subject = null;
    private Address address = null;

    public CustomerCactusTest(String name) {
        super(name);
    }

    public static Test suite() {
        return new TestSuite(CustomerCactusTest.class);
    }

    public void testCustomerAPI() throws Exception {
        InitialContext ic = new InitialContext();
        UserTransaction ut = (UserTransaction)ic.
            lookup(USER_TRANS_REF);
        // need a user transaction so that we can traverse the
```

Listing 1.10 Cactus test for the Customer bean.

```
        // container-managed relationships, usually the transaction
        // would be opened and managed by a session facade.
        ut.begin();
        CustomerHome cHome = (CustomerHome)ic.lookup(CUSTOMER_REF);
        Customer customer = cHome.findByPrimaryKey(new Long(14));
        assertEquals(new Long(14), customer.getId());
        // this should be in an address test method
        assertEquals("14 Mulberry Lane", customer.getAddress().
                    getStreet());
        Collection invoices = customer.getInvoices();
        assertEquals(3, invoices.size());
        Invoice inv = (Invoice)invoices.iterator().next();
        // this should be in an invoice test method
        assertTrue(!(new BigDecimal(0.0)).
                         equals(inv.getOrderTotal()));
        assertEquals("Inigo", customer.getFirstName());
        assertEquals("Montoya", customer.getLastName());
        assertEquals("555-33-4455", customer.getSsn());
        customer.setFirstName("Binigo");
        assertEquals("Binigo", customer.getFirstName());
        ut.commit();
    }

    public void setUp() throws Exception {
    }

    public void tearDown() throws Exception {
    }
}
```

Listing 1.10 *(continued)*

The first clue that this test is unfocused is the inclusion of the JNDI name for other beans in this test. While it is not unreasonable to need the JNDI name for other beans, it is an indicator that the test might be unfocused. The biggest problem in this test class is the testCustomerAPI method. This test method is all over the place, asserting things about most of the API of the Customer bean. It starts off testing things about the customer but then moves to the invoices. Each test method should be focused on one aspect or intent of the API and should assert that one thing instead of several different, seemingly random parts of the API. Another thing to note is that a test class should focus on one subject class. Any tests that are written for the Invoice bean should be in a different test class altogether.

In some cases, the code being tested is just too complex to test in just a few asserts. As argued earlier, if the subject code is doing too much, the subject code should probably be refactored into more methods that each do less work. In some cases it will not be possible for you to refactor the test subject code. Whether you are able to change the subject or not, you should strive to keep the test methods as focused as possible so that your test code will stay out of this pitfall. The best way to keep the test methods focused is to limit the number of asserts in the test to one or two. We will see more on how to do this in the solution to this pitfall.

Solving Pitfall 1.4: Keep It Simple

The solution to Pitfall 1.4: Unfocused Test Method is simple: Keep it simple. Long test methods are hard to maintain and keep in sync with the ever-changing API of the subject. Instead, test methods should be short and have clearly defined goals. For each test method, only one aspect or intent of the subject's API should be asserted, leaving other asserts to other test methods. Make sure in applying this solution that each new test method is named after the kind of test being performed or the intent that is being tested. Another issue to consider in applying this solution is that lots of simple tests are likely to be more thorough than a few big tests. When a test method has a clearly defined piece of the subject to test, that method can be more focused. If, however, the method is just testing the subject, then the asserts can be haphazard in nature and fail to check critical aspects of the state of the subject.

Another thing to think about when applying this solution is the construction of the subject(s) of the tests. As the large tests are broken into more, smaller tests, the overhead of calling setUp and tearDown grows. If this overhead becomes too much, then the "run quickly and often" goal of unit tests can be compromised. In these cases, consider using a test decorator to initialize a common subject that can be used by many different tests.

Keep in mind that each test should be focused on one part of the intent of the subject API. As you are applying this solution to your tests, you might find it useful to look at the solution to Pitfall 1.1: No assert for pointers on capturing the intent of the subject's API. When you are applying this solution to your Cactus tests, keep in mind that getting to your test in the first place is a distributed method call (it takes some time to get set up and executed) so make sure that you test everything that makes sense to test in each test method.

Step-by-Step

1. For each large and complex test method, extract groups of asserts that relate to common intent of the subject API.
   a. "Large" and "complex" are, of course, subjective; the question to keep in mind is "Will this test method be likely to be abandoned

as the subject's API is changed?" If the answer is yes, then the method is probably too complex.

2. Place each related group of asserts into a separate test method.
   a. Name these new methods after the intent that is being asserted.
3. Further decompose the remaining complex test methods.
   a. As a general rule, each test method should have one assert.
4. Deploy and test.
5. Repeat these steps for each test that is trapped in the pitfall.

Example

In this example, a Cactus test is cleaned up from two very complex test methods to many, much simpler methods that test focused pieces of the API. The Invoice bean listed here will be the subject of the example test to which we will apply the solution. The bean manages orders for customers. A single invoice has one customer and the list of line items for that invoice. The initial test is haphazard in its approach to checking the invoice API, as you can see in Listing 1.11, but the fixed test will make sure that the line items are managed properly and that the customer-related methods work as expected.

```
public interface Invoice extends EJBLocalObject {
    public Long getId() throws EJBException;
    // Customer relationship
    public Customer getCustomer() throws EJBException;
    public void setCustomer(Customer customer) throws EJBException;
    // LineItem relationship
    public void addLineItem(LineItem newItem)
        throws EJBException, InvoiceException;
    public void removeLineItem(LineItem newItem)
        throws EJBException, InvoiceException;
    public Collection getLineItems() throws EJBException;
    public void setLineItems(Collection items) throws EJBException;
    // totaling this invoice
    public BigDecimal getOrderTotal() throws EJBException;
}
public interface InvoiceHome extends EJBLocalHome {
    public Invoice create(Customer customer) throws CreateException;
    public BigDecimal totalForCustomer(Customer customer);
    public Invoice findByPrimaryKey(Long pk) throws FinderException;
}
```

Listing 1.11 The API for the Invoice bean.

This entity is responsible for keeping track of the invoices for individual customers. The entity is also able to sum all orders that a customer has ever made. The original test code in Listing 1.12 tries to test everything in only two test methods. The test methods are too long, overly complex, and not all that thorough as a result.

```
public class InvoiceCactusTest extends ServletTestCase {
    // for JBoss this must be "UserTransaction" instead
    private final String TRANS_REF =
        "java:comp/UserTransaction";
    private static final String INVOICE_REF = "invoice.Invoice";
    private static final String CUSTOMER_REF = "invoice.Customer";
    private static final String LINEITEM_REF = "invoice.LineItem";

    private InvoiceHome iHome = null;
    // the subject of the tests in this class
    private Invoice invoice = null;
    // an invoice primary key for use in one of the tests
    private Long invoicePrimaryKey = null;
    // auxiliary objects
    private CustomerHome cHome = null;
    private Customer customer = null;
    private UserTransaction ut = null;

    public InvoiceCactusTest(String name) {
        super(name);
    }

    public static void main(String[] args) {
        String name = InvoiceCactusTest.class.getName();
        junit.textui.TestRunner.main(new String[] {name});
    }

    public static Test suite() {
        return new TestSuite(InvoiceCactusTest.class);
    }

    public void testBean() throws Exception {
        assertNotNull(invoice.getCustomer());
        assertNotNull(invoice.getLineItems());
        Collection lineItems = invoice.getLineItems();
        List removedItems = new ArrayList();
        assertEquals(4, lineItems.size());
        Iterator itr = lineItems.iterator();
        int count = 0;
        while(itr.hasNext()) {
            // remove the even items to test removal
```

Listing 1.12 InvoiceCactusTest.

```
            LineItem lineItem = (LineItem)itr.next();
            if((count % 2) == 0) {
                invoice.removeLineItem(lineItem);
                removedItems.add(lineItem);
            }
        }
        assertEquals(2, invoice.getLineItems().size());
        BigDecimal total = invoice.getOrderTotal();
        assertTrue(total.doubleValue() > 57.00 &&
                   total.doubleValue() < 59.00);
        itr = removedItems.iterator();
        while(itr.hasNext()) {
            LineItem lineItem = (LineItem)itr.next();
            invoice.getLineItems().add(lineItem);
        }
        assertEquals(4, invoice.getLineItems().size());
        total = invoice.getOrderTotal();
        assertTrue(total.doubleValue() > 231.00 &&
                   total.doubleValue() < 233.00);
    }

    public void testHome() throws Exception {
        customer = cHome.create();
        customer.setFirstName("Buzz");
        customer.setLastName("Lightyear");
        invoice = iHome.create(customer);
        assertNotNull(invoice.getCustomer());
        Invoice anInvoice = iHome.findByPrimaryKey(invoicePrimaryKey);
        assertEquals(invoicePrimaryKey, anInvoice.getId());
        BigDecimal total = iHome.totalForCustomer(customer);
        assertTrue(total.doubleValue() > 231.00 &&
                   total.doubleValue() < 233.00);
    }
. . . the setUp and tearDown methods go here . . .
}
```

Listing 1.12 *(continued)*

The testBean method is all over the place, testing many different pieces of the invoice API. A much more effective way to test the entity is to test each intent in a different method. The first step of this solution is to group the asserts according to the intent that is to be tested. The local interface for this bean has three basic functions: manage the relationship with the customer, manage the relationship with the line items this invoice contains, and sum the list of line items to get the total for this invoice. The tests should make sure that these three parts of the API are functioning as expected. The home interface for this bean is very simple: one finder, one create, and one business method. The test should make sure that these aspects of the API are tested as well.

The testBean method is more or less testing each aspect of the API for the entity, but the depth of testing is not consistent. The asserts should be grouped into the three basic areas. The next step, after grouping the asserts, is to move them into another test method named after the function or intent they are supposed to be testing. The asserts can be cut and pasted into the new methods. Here is the code after being put through these first two steps.

```
public void testCustomerRelationship() throws Exception {
    // check the customer relationship
    assertNotNull(invoice.getCustomer());
}

public void testLineItemRelationship() throws Exception {
    // check the lineitem relationship
    assertNotNull(invoice.getLineItems());
    Collection lineItems = invoice.getLineItems();
    List removedItems = new ArrayList();
    assertEquals(5, lineItems.size());
    Iterator itr = lineItems.iterator();
    int count = 0;
    while(itr.hasNext()) {
        LineItem lineItem = (LineItem)itr.next();
        // remove the even items (0, 2, 4) to test removal
        if((count % 2) == 0) {
            invoice.removeLineItem(lineItem);
            removedItems.add(lineItem);
        }
    }
    assertEquals(2, invoice.getLineItems().size());
    // test adding items
    itr = removedItems.iterator();
    while(itr.hasNext()) {
        LineItem lineItem = (LineItem)itr.next();
        invoice.getLineItems().add(lineItem);
    }
    assertEquals(5, invoice.getLineItems().size());
}

public void testTotaling() throws Exception {
    BigDecimal total = invoice.getOrderTotal();
    assertTrue(total.doubleValue() > 231.00 &&
               total.doubleValue() < 233.00);
}
```

Notice that this is basically the same code from the testBean method; it's just been grouped according to the functional area of the API that it was testing.

The next step in the solution is to further decompose the complex test methods into simpler test methods (aim for one assert per test method). The next method to attack would be the testLineItemRelationship method because it has several asserts. One aspect of the bean that can be pulled into another test is the totaling of the invoice. Specifically, if a line item is removed, the total should change. Here is a test to make sure that the total does change when items are removed.

```
public void testTotalingDiffers() throws Exception {
    BigDecimal total = invoice.getOrderTotal();
    removeEvenItems(invoice);
    BigDecimal smallerTotal = invoice.getOrderTotal();
    // total should be larger than smallerTotal
    assertTrue(total.compareTo(smallerTotal) > 0);
}
```

The rest of the testLineItemRelationship method needs to be broken up into several test methods. Listing 1.13 shows the code resulting from breaking up the original testBean method.

```
public void testCustomerRelationship() throws Exception {
    // check the customer relationship
    // this test will let us know if anything has
    // gone wrong in the container-managed relationships
    assertNotNull(invoice.getCustomer());
}

public void testLineItems() throws Exception {
    // check the lineitem relationship
    // this test will let us know if anything has
    // gone wrong in the container-managed relationships
    assertNotNull(invoice.getLineItems());
}

public void testLineItemsCount() throws Exception {
    // 5 line items are put in during setUp
    Collection lineItems = invoice.getLineItems();
    // make sure they are there
    assertEquals(5, lineItems.size());
}

public void testLineItemRemoval() throws Exception {
   Collection lineItems = invoice.getLineItems();
```

Listing 1.13 Fixed code from the original testBean method. *(continues)*

```
        Iterator itr = lineItems.iterator();
        // remove the first element
        while(itr.hasNext()) {
            LineItem lineItem = (LineItem)itr.next();
            invoice.removeLineItem(lineItem);
            break;
        }
        // make sure there are four items left
        assertEquals(4, invoice.getLineItems().size());
    }

    public void testTotaling() throws Exception {
        BigDecimal total = invoice.getOrderTotal();
        assertTrue(total.doubleValue() > 231.00 &&
                   total.doubleValue() < 233.00);
    }

    public void testTotalingDiffers() throws Exception {
        BigDecimal total = invoice.getOrderTotal();
        removeEvenItems(invoice);
        BigDecimal smallerTotal = invoice.getOrderTotal();
        // total should be larger than smallerTotal
        assertTrue(total.compareTo(smallerTotal) > 0);
    }
```

Listing 1.13 *(continued)*

The single unfocused testBean method is now cleaned up to be six separate test methods. The test code is much easier to follow and is more focused on the particular aspect that is being tested. The names of the test methods are also more descriptive.

This is a good time to bring up TestDecorators in Cactus. Keep in mind that a typical implementation of a JUnit test decorator in your Cactus tests will be run on the client side, not on the server side. This means that any access to a local EJB interface will fail because the test starts in a different JVM. In order to use test decorators they must interact with remote interfaces. You can accomplish this in several different ways. Placing a session façade is a popular way to provide a remote interface to the underlying entities. The typical problem with this approach, however, is that the session then needs specific methods to support testing. An alternative approach is to deploy your entities twice, once under their local interfaces and once with remote interfaces when doing testing. Placing remote interfaces over your entities is a maintenance burden so it is not without cost. Using the session façade is the recommended approach.

Pitfall 1.5: Failure to Isolate Each Test

Pitfall 1.5: Failure to Isolate Each Test usually takes the form of some external script that must be run before the tests will succeed. Examples of external scripts include things like initialization data (or scripts that clean out the database) in an SQL script that must be run against the database. Another typical form is for the test methods to have order dependencies. For example, one test might create an object, the next method makes a couple of state changes, and the last method deletes the object. In either form, the tests will become unreliable if they remain stuck in this pitfall because, over time, the order of the methods will change and they will start to fail. Often, developers will forget to run the database script often enough that they get fed up with the tests. Tests that are not reliable will cease to be used because they show too many false bugs and catch too few actual bugs.

Loading data into the database via a script prior to running a test suite is a very typical practice. The problem with this approach, though, is that the test is no longer self-sufficient. The developer (and anyone else running the tests) could be required to run the script before running the tests. If the test fails because the database script has not been run, the results can be confusing and lead to questions about the validity of the test or even the code being tested.

CACTUS AND MOCK OBJECT APPLICATION SERVERS

Some in the unit testing community go so far as to say that the application server configuration is another example of configuration or setup that should be part of the test. This is another place where the discussion of Mock Objects is prevalent. In some cases, people have developed entire mock application servers so that the tests can run in isolation from the application server configuration. While that makes for very isolated tests, the amount of work required to maintain these Mock Objects is often as much or more than the work required to maintain the tests.

Cactus takes the opposite approach to the application server. Instead of seeing it as a factor to be isolated and removed from the test, Cactus embraces the application server. Instead of trying to supply Mock Objects for all the server-side services, Cactus provides the actual server-side objects to the tests. This approach does introduce a dependency on the application server and might cause tests to fail as a result of bugs in the underlying application server. If there are bugs in the underlying application server, it is probably better to know that up front instead and thus be able to work around them than to wait until the application is deployed to the test team for the bug to be exposed.

This pitfall usually develops because a developer got lazy about building sufficient test subject objects in the setUp method. Building good test subjects often takes time and sometimes even digging around in the subject code to make sure that the correct set of initial conditions is met to perform the test. It is often argued that this time is not worth the trouble because writing an SQL script to load the database could be done in half the time. That might be true for the first set of tests, but over time the data file becomes a constant source of trouble.

Another driver behind this pitfall is a lack of knowledge or experience with Cactus or JUnit. Inexperienced developers often put a bunch of test data into the database and then run their tests. The test runs once and then is not touched again for a while. In the meantime, other testing has modified the data in the database enough that the tests fail. False failures are a constant headache and often lead to unit testing being dropped from a project because developers and managers do not see value in the tests because of the maintenance overhead.

This pitfall also shows up in tests that add or remove data from the database. If each test does not clean up after it runs, then the data that the test changed must be reset, or the test might not run the next time. In other cases, this issue shows up as one test that can inadvertently delete the data on which another test depends. If the first test does not clean up, the second test will fail because its data has been removed.

The important thing to keep in mind is that the tests need to be isolated enough to be run without manual intervention by the test runner. In other words, requiring that the application server be running for the tests to succeed is fine, but requiring that an SQL script be run after each test is not. The difference between the two is subtle. If the application server is not running the failure is obvious, and it is easy to see from the failure that Cactus could not contact the server-side piece of the test. If, on the other hand, the data has not been reinitialized in the database, the failure might show up as a NullPointerException. Testers never like to see NullPointerExceptions; if they do happen, the testers can and often do lose confidence in the application.

Example

In this example, we will test a Customer Entity EJB. This bean manages the typical information needed to manage a customer, the name, address, and so on, as well as the list of invoices for goods the customer has purchased. The API the example will test is found in Listing 1.14. The intent of the particular parts of the API will be discussed as they are tested in the example. Because this is an Entity Bean, we will use Cactus to test.

```
public interface Customer extends EJBLocalObject {
    public Long getId() throws EJBException;
    public String getFirstName() throws EJBException;
    public void setFirstName(String param) throws EJBException;
    public String getLastName() throws EJBException;
    public void setLastName(String param) throws EJBException;
    public String getSsn() throws EJBException;
    public void setSsn(String param) throws EJBException;
    public Address getAddress() throws EJBException;
    public void setAddress(Address param) throws EJBException;
    public Collection getInvoices() throws EJBException;
    public void setInvoices(Collection invoices) throws EJBException;
}
public interface CustomerHome extends EJBLocalHome {
    public Customer create() throws CreateException;
    public Customer lookupCustomer(CustomerPO customer,
                                   AddressPO address)
        throws FinderException;
    public Customer findByPrimaryKey(Long pk) throws FinderException;
}
```

Listing 1.14 Customer bean API.

The Cactus test (in Listing 1.15) for the Customer bean API is stuck in this pitfall. Notice that testLookupCustomer depends on testCreateCustomer being executed first. Also, the testSetAddress method relies on data being in the database from some external process (probably an SQL script). Here is the code.

```
public class CustomerCactusTest extends ServletTestCase {
    private final String CUSTOMER_REF = "invoice.Customer";
    private final String ADDRESS_REF = "invoice.Address";
    private final String INVOICE_REF = "invoice.Invoice";

    private InvoiceHome iHome = null;
    private CustomerHome cHome = null;
    private Customer subject = null;
    private Address address = null;
    private AddressHome aHome = null;

    public CustomerCactusTest(String name) {
        super(name);
    }

    public static Test suite() {
        return new TestSuite(CustomerCactusTest.class);
```

Listing 1.15 CustomerCactusTest. *(continues)*

```
    }

    /**
     * This method tests that a newly created customer is in the
     * right state. The create method call will fail if the customer
     * named 'Elmer Fudd' is not deleted from the database
     * manually after each test run.
     * @throws Exception
     */
    public void testInitialCustomer() throws Exception {
        InitialContext ic = new InitialContext();
        cHome = (CustomerHome) ic.lookup(CUSTOMER_REF);
        CustomerPO custPO = new CustomerPO("Elmer", "Fudd");
        Customer customer = cHome.create(custPO);
        assertNotNull(customer);
        assertNotNull(customer.getId());
        assertNull(customer.getAddress());
        assertTrue(customer.getInvoices().size() == 0);
        assertEquals("Elmer", customer.getFirstName());
        assertEquals("Fudd", customer.getLastName());
    }

    /**
     * This method tests creating and adding an invoice. This method
     * will fail if the customer 'Elmer Fudd' is not in the
     * database already. There is an order dependency between this
     * method and testInitialCustomer.
     * @throws Exception
     */
    public void testAddInvoice() throws Exception {
        InitialContext ic = new InitialContext();
        cHome = (CustomerHome) ic.lookup(CUSTOMER_REF);
        iHome = (InvoiceHome) ic.lookup(INVOICE_REF);
        CustomerPO custPO = new CustomerPO("Elmer", "Fudd");
        Customer customer = cHome.lookupCustomer(custPO, null);
        Invoice inv = iHome.create(customer);
        assertTrue(customer.getInvoices().size() == 1);
    }

    /**
     * This method tests looking up a customer. This method requires
     * that the testInitialCustomer method has already been run or
     * that 'Elmer Fudd' is in the database.
     * @throws Exception
     */
    public void testLookupCustomer() throws Exception {
        InitialContext ic = new InitialContext();
        cHome = (CustomerHome) ic.lookup(CUSTOMER_REF);
        CustomerPO po = new CustomerPO("Elmer", "Fudd");
```

Listing 1.15 *(continued)*

b 449156 Ch01.qxd 6/16/03 8:48 AM Page 49

```
        Customer customer = cHome.lookupCustomer(po, null);
        assertNotNull(customer);
        assertEquals("Elmer", customer.getFirstName());
        assertEquals("Fudd", customer.getLastName());
    }

    /**
     * This method tests setting the address on an existing customer.
     * It will fail if 'Pancho Villia' is not already in the database.
     * @throws Exception
     */
    public void testSetAddress() throws Exception {
        InitialContext ic = new InitialContext();
        cHome = (CustomerHome) ic.lookup(CUSTOMER_REF);
        aHome = (AddressHome) ic.lookup(ADDRESS_REF);
        Address address = aHome.create();
        address.setStreet("128 Wherever Ln.");
        address.setCity("Wizzleville");
        address.setState("CO");
        address.setZipCode("80345");
        CustomerPO po = new CustomerPO("Pancho", "Villia");
        Customer customer = cHome.lookupCustomer(po, null);
        customer.setAddress(address);
        assertNotNull(customer.getAddress());
    }

    public void setUp() throws Exception {
    }

    public void tearDown() throws Exception {
    }
}
```

Listing 1.15 *(continued)*

The order-dependent methods run on Sun's Windows JVM and Apple JVM in the correct order (probably because of their order in the file), but that is nothing to rely on. What will happen when the application is moved to big-iron servers and runs on IBM's JVM? Who knows?—that is the point. The tests might or might not be run in the correct order. Other JVMs might provide methods in different order than the Sun JVM, and that would cause the tests to fail.

On the surface, this might not look like much, just moving the methods around until they execute in the correct order. Or the really savvy JUnit developer might suggest creating a custom version of the test suite that will execute the tests in the order they are added and using the custom suite instead of the default reflection-based suite. While these workarounds are fine for a small project (say, one or two developers), they will not work on a bigger project. Members of the team other than the author of the test will come to hate this test over time because there will be failed tests unless they delete 'Elmer Fudd' from the database after each run of the tests. That manual step defeats the purpose of having repeatable, automatic unit tests. Tests must not be allowed to continue in this sorry state.

Solving Pitfall 1.5: Use setUp and tearDown and Introduce Test Decorators

There are two ways to solve this pitfall. The first is to use the setUp and tearDown methods that Cactus calls automatically before and after each test is run. The second, Introduce Test Decorators, is a twist on setUp and tearDown that can be used if the performance of having setUp and tearDown executed with each test becomes burdensome.

Better isolation (and better tests) is achieved by factoring out dependencies between tests. If the setUp method creates the subject and tearDown deletes the subject, then there is no opportunity for the tests to depend on each other. Each run of each test is starting over with a clean slate. If tests are isolated in this way, problems will be easier to pin on the subject code instead of some part of the testing apparatus. The failures need to be tied to the subject for the test to have value.

This solution also helps tests be more isolated from each other by getting rid of manual processes that are required for the tests to run. In small teams, manual processes are easy to communicate and are thus not as much of an issue, but in big teams manual processes are hard to communicate so that everyone on the team is completely comfortable with what needs to be done. Over time, the tests that require manual steps will be abandoned because when the manual process is not followed exactly, the tests fail. Unit tests should not have this kind of dependency. After all, they are supposed to be individual, independent, repeatable units.

The manual processes are usually captured well in test decorators. Whenever you have a manual process that you are building to support your tests, you should instead be building a test decorator to do all that you would do manually. Once the decorator is coded, the manual process is there for everyone else to use. We will see more about test decorators later in this solution.

Step-by-Step

1. Review the tests for creation, initialization, and deletion of subjects.
   a. Look especially for commonality between the initialization or creation.
   b. If the creation differs significantly, consider creating a new test class or custom setUp and tearDown methods that you can call manually.
2. Review the tests for order dependencies.
   a. Look for assumed instances.
3. Review manual processes needed to make the tests runnable.
   a. Look for SQL scripts that must be run or other external scripts required to make the tests pass.
4. Create a setUp and tearDown method pair.
   a. Consider using test decorators instead if the setUp and/or tear-Down methods will take a long time to complete.
5. Add an instance variable to the test class for each distinct kind of subject.
   a. Each kind of subject will be used to test a different part of the API; for example, one subject could be useful for testing error response in the API, and another subject could be useful for testing the deletion behavior.
6. Move the create or initialization code from the tests to the setUp method.
   a. Initialize one subject for each distinct kind found in the first few steps.
   b. Find commonalities between the subjects, and initialize the subjects so that they can be reused in many tests.
   c. If there are many different sets of initialization code that must be built, consider creating a new test class for each kind of subject.
7. Move (or add) the delete code to the tearDown method.
8. Deploy and test.

Example

In this example, we will see applying the solution to the test class that we used to test the Customer bean earlier. As you will recall, the test is trapped in this pitfall and prone to unreliability. We will take that code through the steps and see how to fix the code so that it is more reliable and a better gauge of the quality of the test subject.

The first step in the process is to identify code that does creation and/or deletion in the tests. The testInitialCustomer method creates a Customer bean and then initializes it. The next step in the solution is to review the test for order dependencies. The testLookupCustomer method relies on the testInitialCustomer method being executed first so that the data is in the database. The third step is to look for manual steps that must be made for the tests to execute properly. The testSetAddress method relies on a preexisting customer being in the database. Now that we have reviewed the tests for candidate code for the setUp method, it is time to move that code into the setUp method.

The next step in the process is to add a setUp and tearDown method pair and then fill the setUp method with the code identified in the previous steps. For each distinct customer instance that will be tested, we need an instance variable. Two variables are added to the class, and the setUp method is changed to look like the code in Listing 1.16.

```
public class CustomerCactusTest extends ServletTestCase {
    ...
    private Customer elmerFudd;
    private Customer panchoVillia;
   ...

    public void setUp() throws Exception {
        InitialContext ic = new InitialContext();
        cHome = (CustomerHome) ic.lookup(CUSTOMER_REF);
        aHome = (AddressHome) ic.lookup(ADDRESS_REF);
        iHome = (InvoiceHome) ic.lookup(INVOICE_REF);
        CustomerPO custPO = new CustomerPO("Elmer", "Fudd");
        elmerFudd = cHome.create(custPO);
        custPO.setFirstName("Pancho");
        custPO.setLastName("Villia");
        panchoVillia = cHome.create(custPO);
    }
```

Listing 1.16 New setUp method with initialization code.

```
    public void tearDown() throws Exception {
        elmerFudd.remove();
        indigoMontoya.remove();
        panchoVillia.remove();
    }
```

Listing 1.17 New tearDown method.

Notice that the setUp method in Listing 1.16 now creates all the data needed by the tests. No longer will the tests rely on an SQL script being run in order to pass. Notice also that the lookup for each of the homes is done here in the setUp method. Remember that putting this code in the setUp method does not save us any execution time—it is just convenient.

Because JNDI lookups can be expensive, you might consider a test decorator to do the JNDI lookup if the time to do the lookups in your test is prohibitively long (see Example 2, which follows). Also notice that in this code, the JNDI initial context does not require any initialization parameters. Remember that Cactus tests run on the server and because it is running on the server the InitialContext is able to root itself in the JNDI tree there.

The next step is to remove all this data created in the setUp in the tearDown method. Deleting the data is very important so that the tests can be repeatable. If the data is left in the database, all the create method calls in the setUp code will fail because there are customers with the same name already in the database (arguably, you would not want to build a real customer management system so that you could have only one customer per name; the point is that the tests need to clean up after themselves). The tearDown code is Listing 1.17.

The final step is to deploy and retest. Before we move on, though, let's take a look at some of the test methods and how they have changed. Specifically, let's look at the two test methods that had an order dependency. The initial code is Listing 1.18.

```
    public void testInitialCustomer() throws Exception {
        InitialContext ic = new InitialContext();
        cHome = (CustomerHome) ic.lookup(CUSTOMER_REF);
        CustomerPO custPO = new CustomerPO("Elmer", "Fudd");
        Customer customer = cHome.create(custPO);
        assertNotNull(customer);
        assertNotNull(customer.getId());
```

Listing 1.18 Original order-dependent test methods. *(continues)*

```
        assertNull(customer.getAddress());
        assertTrue(customer.getInvoices().size() == 0);
        assertEquals("Elmer", customer.getFirstName());
        assertEquals("Fudd", customer.getLastName());
    }

    public void testLookupCustomer() throws Exception {
        InitialContext ic = new InitialContext();
        cHome = (CustomerHome) ic.lookup(CUSTOMER_REF);
        CustomerPO po = new CustomerPO("Elmer", "Fudd");
        Customer customer = cHome.lookupCustomer(po, null);
        assertNotNull(customer);
        assertEquals("Elmer", customer.getFirstName());
        assertEquals("Fudd", customer.getLastName());
    }
```

Listing 1.18 *(continued)*

The new methods do not have the order dependency because they are able to rely on the beans created and destroyed by the setUp and tearDown methods. The new code is found in Listing 1.19.

The original method and the fixed method do not vary by many lines of code. The only real change is that the creation and JNDI lookup code was moved to the setUp method. This simple change makes all the difference. Now these two tests can run reliably and repeatably.

```
    public void testInitialCustomer() throws Exception {
        assertNotNull(elmerFudd);
        assertNotNull(elmerFudd.getId());
        assertNull(elmerFudd.getAddress());
        assertTrue(elmerFudd.getInvoices().size() == 0);
        assertEquals("Elmer", elmerFudd.getFirstName());
        assertEquals("Fudd", elmerFudd.getLastName());
    }

    public void testLookupCustomer() throws Exception {
        CustomerPO po = new CustomerPO("Elmer", "Fudd");
        Customer customer = cHome.lookupCustomer(po, null);
        assertNotNull(customer);
        assertEquals("Elmer", customer.getFirstName());
        assertEquals("Fudd", customer.getLastName());
    }
```

Listing 1.19 New order-independent test methods.

b 449156 Ch01.qxd 6/16/03 8:48 AM Page 55

Example 2: Introduce Test Decorators

Test decorators have been mentioned a couple of times in this solution as something to look into if the setUp and tearDown methods are taking too long to execute. The nice thing about test decorators is that the setUp method is executed once, then the suite of test methods contained by the decorator is run and the test decorator's tearDown method is called. This approach allows the setUp and tearDown functionality to be executed only once for the whole set of tests. A decorator is introduced in the suite method on the test class. Here is the new code for that method.

```
public static Test suite() {
    return new CustomerBeanTestDecorator(
        new TestSuite(CustomerCactusTest.class));
}
```

The test decorator encapsulates the tests from the Cactus test class so that they are executed after the test decorator's setUp method is called. Then, after the last test is finished, the decorator's tearDown method is called. The code for the two methods is listed here.

```
public void setUp() throws Exception {
    Context ic = getInitialContext();
    CustomerHome cHome = (CustomerHome) ic.lookup(CUSTOMER_REF);
    CustomerPO custPO = new CustomerPO("Elmer", "Fudd");
    elmerFudd = cHome.create(custPO);
    custPO.setFirstName("Indigo");
    custPO.setLastName("Montoya");
    indigoMontoya = cHome.create(custPO);
    custPO.setFirstName("Pancho");
    custPO.setLastName("Villia");
    panchoVillia = cHome.create(custPO);
}

public void tearDown() throws Exception {
    elmerFudd.remove();
    indigoMontoya.remove();
    panchoVillia.remove();
}
```

With this decorator in place, the three Customer beans that are used in the tests will be created and removed only once per run of the test suite. The code was moved from the setUp and tearDown methods on the CustomerCactusTest class to the decorator for that purpose.

An important concept to make note of is that the decorator’s setUp and tearDown methods are executed on the client side of Cactus, not on the server side. This means that the home and bean interface must be remote for this code to work. In this example, the Customer bean being tested was local only in the original code. The Customer bean must be redeployed with remote interfaces for this example to work.

Another way that decorator can be used is to automate the manual tasks required to prepare the environment for the tests to run. As stated earlier, the typical manual task loads test data into a database. That can easily be done from within a setUp method on a test decorator. Once the manual step is captured in a test decorator, the test will be more reliable and repeatable. The code on the Web site has an example of a test decorator that loads and removes data from the database by running an SQL script through JDBC into the database in the setUp and tearDown methods of a test decorator.

Listing 1.20 is an example test decorator that uses JDBC to load the content of a file and execute it against the database. This decorator could be used to eliminate the manual task of running an SQL script against the database. If you have a large data set, this way is probably more practical than using the create methods. The EJB create methods are not really intended to be used to do batch-oriented processing.

```
public class CustomerLoaderTestDecorator extends TestSetup {
    private final String DB_URL =
            "jdbc:mysql://localhost/book?user=book";
    private final String DB_USER = "book";
    private final String DB_PASS = "book";
    //load the driver class
    private static final Class mySQLDriverClass = Driver.class;

    public CustomerLoaderTestDecorator(Test test) {
        super(test);
    }

    public void setUp() throws Exception {
        // AddData.sql is delimited by ';'
        InputStream is =
            getClass().getClassLoader().getResourceAsStream(
                "AddData.sql");
        executeBuffer(is);
    }

    public void tearDown() throws Exception {
        // RemoveData.sql is delimited by ';'
        InputStream is =
            getClass().getClassLoader().getResourceAsStream(
```

Listing 1.20 CustomerLoaderTestDecorator.

```
"RemoveData.sql");
        executeBuffer(is);
    }

    private void executeBuffer(InputStream is) throws Exception {
        StringBuffer sql = new StringBuffer(128);
        DriverManager.registerDriver(new Driver());
        Connection conn = DriverManager.getConnection(DB_URL,
                   DB_USER, DB_PASS);
        int nextChar = is.read();
        while(-1 != nextChar) {
            if(';' == nextChar) {
                Statement stmt = conn.createStatement();
                stmt.execute(sql.toString());
                stmt.close();
                stmt = null;
                sql.setLength(0);
            } else if('\n' != nextChar) { // skip new lines
                sql.append((char)nextChar);
            }
            nextChar = is.read();
        }
    }
}
```

Listing 1.20 *(continued)*

Applying this decorator to the test will remove the need to use the create calls and the remove calls from the tearDown method.

Pitfall 1.6: Failure to Isolate Subject

Test subjects that rely on other classes to function properly (which are most classes) are harder to test because the underlying classes might have bugs. Bugs in an underlying class might cause the test to fail even though the bug is not in the subject of the test. In a perfect world, no test failures would be caused by code other than the test subject. In that kind of scenario, the test would pass or fail based solely on the bugs (or lack of bugs) in the test subject. While the ideal scenario is hard to achieve, it is worth pursuing. Developers often don't even think about trying, and thus they end up in this pitfall.

Developers usually fall into this pitfall because they fail to understand that a unit test is just that; it's supposed to test a single unit of functionality at a time. Often tests end up testing the class under consideration and everything on which the test depends. That makes it very hard to isolate where the problem is and defeats part of the reason for having unit tests—essentially, the unit tests become integration tests. Instead, each of the underlying or support classes should have its own unit tests.

This pitfall usually shows itself as frequently broken tests, especially when a new version of a framework or library is incorporated. If the library is not being unit tested, then every user of the library is forced to debug it with his or her own tests. This is an inefficient process. The client code often looks broken when the underlying framework actually has the bug.

The form of this pitfall is seen in tests that work fine one day and then fail when some underlying .jar file is replaced. On small projects with one or two developers, this rarely happens because there is a lot of communication between the developers and very little happens that the whole team does not know about. On bigger teams, however, this is common. Developers for one part of the application need a later version of some class and upgrade and may not know the effect on others until after unit tests start failing. (One might argue that the unit tests should be run before checking in the new .jar file, but that is another pitfall for another time.)

The issue the developer faces is figuring out if the problem lies in the subject or the underlying classes from the new .jar file. Sometimes the developer does not even know about the new classes, just that the tests stop running properly. If the tests are properly isolated, then bugs in the subject will be exposed by the tests for the subject, and any bugs in the underlying classes will be exposed by unit tests for those classes.

DESIGNING WITH TESTING IN MIND

Classes are more testable when they are designed with testing in mind. For example, imagine that a test subject relies on an instance of the java.util.Map interface in order to function and that the subject directly allocates an instance of my.great.coll.lib.HashMap to use in its implementation. Without thought to testing, this is a perfectly reasonable and typical practice. When we want to test the class and we do not want to rely on the custom map implementation (because we don't want bugs there to cause false test failures), we have no opportunity to replace the implementation of the custom map with one that we know works. If the test subject were slightly altered to allow the test to provide the implementation of the map, then the subject could be better isolated. If the subject can be better isolated, then test failures are more likely to be an indication of a bug in the subject instead of possibly being with the custom implementation of the map.

There is value in integration tests, especially the integration between concurrent development teams. Cactus/JUnit is a great framework to use to build these kinds of tests. Problems arise when integration tests are used as unit tests. Bugs in the underlying code can mask problems in the subject and vice versa. Both kinds of tests are valuable and should be written.

Example

This pitfall is illustrated with tests for an Invoice class. The Invoice class provides the total dollar amount of the invoice by summing each line item. Here is the code for the Invoice class (Listing 1.21), followed by the code for the JUnit test for the invoice (Listing 1.22).

```
public class Invoice  {
    private List items;

    public Invoice() {
        super();
    }

    public void addLineItem(LineItem item) {
        getLineItems().add(item);
    }

    public void removeLineItem(LineItem item) {
        getLineItems().remove(item);
    }
```

Listing 1.21 Invoice. *(continues)*

```
public Collection getLineItems() {
        if(null == items) {
            items = new ArrayList();
        }
        return items;
    }

    public void setLineItems(Collection items) {
        this.items = new ArrayList(items.size());
        Iterator itr = items.iterator();
        while(itr.hasNext()) {
            this.items.add(itr.next());
        }
    }

    public BigDecimal getInvoiceTotal() {
        Iterator itr = items.iterator();
        BigDecimal total = (new BigDecimal((double)0.0)).setScale(2);
        while(itr.hasNext()) {
            LineItem item = (LineItem)itr.next();
            total = total.add(item.getLineTotal());
        }
        return total;
    }
}
```

Listing 1.21 *(continued)*

```
public class InvoiceTest extends TestCase {
    private static final int QUANTITY = 3;
    private static final double COST = 12.34;
    private static final int COUNT = 10;

    private Invoice subject = null;

    public InvoiceTest(String name) {
        super(name);
    }

    public static Test suite() {
        return new TestSuite(InvoiceTest.class);
    }

    public void testInvoiceTotal() throws Exception {
        BigDecimal total = subject.getInvoiceTotal();
        BigDecimal expected = new BigDecimal(COUNT *
```

Listing 1.22 InvoiceTest.

b 449156 Ch01.qxd 6/16/03 8:48 AM Page 61

```
COST * QUANTITY);
        // setting up a value that can be used to assert the intent
        // of the API documented with the method.
        expected = expected.setScale(2, BigDecimal.ROUND_HALF_UP);
        // assert the intent
        assertEquals(expected, total);
    }

    public void setUp() throws Exception {
        subject = new Invoice();
        List items = new ArrayList();
        LineItem item = null;
        for(int i = 0;i < COUNT;i++) {
            item = new LineItem(subject);
            item.setUnitPrice(new BigDecimal(COST));
            item.setQuantity(new Integer(QUANTITY));
            items.add(item);
        }
        subject.setLineItems(items);
    }

    public void tearDown() throws Exception {
        subject = null;
    }
}
```

Listing 1.22 *(continued)*

The setUp method is creating a new invoice for the test and adding several line items to initialize the subject. This test indirectly tests the getLineTotal method from the LineItem class as well as the intended method on Invoice. It is important that the getLineTotal method is tested; however, that test needs to be in the unit test for the LineItem class, not in the test for Invoice. If the getLineTotal method has a bug, then the invoice test will fail—not due to a bug in the Invoice class, but rather due to the bug in the LineItem class.

Do not misunderstand—there is value in testing the interaction between classes like Invoice and LineItem. The integration tests should be run as such, so that any test failures can be attributed to the way that Invoice and LineItem interact instead of being falsely attributed to a bug in either class. With tests in place that are distinct, bugs like this are identified early and fixed more quickly, which is, of course, the whole point of doing testing.

Solving Pitfall 1.6: Introduce Mock Objects

The most common way to solve this pitfall is to use the Mock Objects Pattern. With Mock Objects, the tests become inside and outside tests. The Mock Objects become pseudo-tests on the inside, and the JUnit test body becomes the test from the outside. With this strategy, the interaction between the subject and the underlying objects can be tested as well as the results of that interaction on the subject itself. The basic idea is to replace the objects that the test subject depends on with Mock Objects that provide the same API but yield hard-coded data and simplified functionality to reduce the likelihood that a bug in the underlying code will affect the test.

Step-by-Step

1. Identify all the classes required to implement the subject.
2. For each class or interface, build a Mock Implementation.
3. For classes, write a subclass and override the required methods to return canned data.
   a. Make sure to provide initialization methods to specify what parameters to expect and what values to return when these values are provided.
4. In the test, initialize the Mock Object with the expected parameters and the values to return.
5. Initialize the subject with the Mock Object.
6. Call the method being tested.
7. Assert whatever state change is expected in the subject.
8. Assert state changes expected in the Mock Object.
9. Deploy and test.

Example

This example will test an Invoice class that is able to sum the line items it contains. The invoice asks each line item for its total value and sums these numbers to arrive at an order total. Because the invoice depends so heavily on the LineItem class, the test for the invoice will be testing the LineItem class even though it is not intended to do so. The InvoiceTest test class is stuck in this pitfall, as you can see in Listing 1.23, as it is testing both the invoice and, indirectly, the line item code.

```
public class InvoiceTest extends TestCase {
    private static final int QUANTITY = 3;
    private static final double COST = 12.34;
    private static final int COUNT = 10;

    private Invoice subject = null;

    public InvoiceTest(String name) {
        super(name);
    }

    public static Test suite() {
        return new TestSuite(InvoiceTest.class);
    }

    public void testInvoiceTotal() throws Exception {
        BigDecimal total = subject.getInvoiceTotal();
        BigDecimal expected = new BigDecimal(COUNT * COST *
                    QUANTITY);
        // setting up a value that can be used to assert the intent
        // of the API documented with the method.
        expected = expected.setScale(2, BigDecimal.ROUND_HALF_UP);
        // assert the intent
        assertEquals(expected, total);
    }

    public void setUp() throws Exception {
        subject = new Invoice();
        List items = new ArrayList();
        LineItem item = null;
        for(int i = 0;i < COUNT;i++) {
            BigDecimal total = (new BigDecimal(COST)).
                setScale(2, BigDecimal.ROUND_HALF_UP);
            BigDecimal quantity = (new BigDecimal(QUANTITY)).
                setScale(2, BigDecimal.ROUND_HALF_UP);
            total = total.multiply(quantity).
                setScale(2, BigDecimal.ROUND_HALF_UP);
            items.add(new MockLineItem(subject, total));
        }
        subject.setLineItems(items);
    }

    public void tearDown() throws Exception {
        subject = null;
    }
}
```

Listing 1.23 InvoiceTest.

The testTotal method is testing what it should test in calling the get-InvoiceTotal method. Because the implementation of that method is so heavily dependent on the LineItem class, it is also testing, indirectly, the getLineTotal method on the LineItem class. Bugs in the LineItem class could show up as test failures in Invoice's tests.

The first step in solving this pitfall is to identify the underlying classes that Invoice needs for its implementation. We have identified the LineItem class already. ArrayList and Long are also used to implement the Invoice; however, because these two classes are part of the JDK we will assume they work and not try to replace them with Mock Objects.

The next step is to build a mock implementation of the LineItem class. Because LineItem is a class and not an interface, a subclass of LineItem is used that overrides the methods that Invoice uses. The implementation will return hard-coded data. If the Invoice class used an interface for LineItem instead, the mock implementation would simply implement that interface. Here is the code for the mock implementation of the LineItem class.

```
public class MockLineItem extends LineItem {
    BigDecimal expected;

    public MockLineItem(Invoice invoice, BigDecimal expected) {
        super(invoice);
        this.expected = expected;
    }

    // this is the only method that actually does anything
    public BigDecimal getLineTotal() {
        return expected;
    }
}
```

The next step is to change the test so that it uses instances of the Mock Object instead of the actual object in the test. Because our test is creating LineItems in the setUp method, we modify that method to create Mock-LineItems instead. Here is the code for the changed setUp method.

```
    public void setUp() throws Exception {
        subject = new Invoice();
        List items = new ArrayList();
        LineItem item = null;
        for(int i = 0;i < COUNT;i++) {
            BigDecimal total = (new BigDecimal(COST)).
                setScale(2, BigDecimal.ROUND_HALF_UP);
            BigDecimal quantity = (new BigDecimal(QUANTITY)).
                setScale(2, BigDecimal.ROUND_HALF_UP);
            total = total.multiply(quantity).
            setScale(2, BigDecimal.ROUND_HALF_UP);
        items.add(new MockLineItem(subject, total));
    }
    subject.setLineItems(items);
}
```

The final step is to change the test methods to assert the correct state changes. The expected value has not changed since we populated our test subject with the same data, so the testInvoiceTotal method remains unchanged and looks like this.

```
public void testInvoiceTotal() throws Exception {
    BigDecimal total = subject.getInvoiceTotal();
    BigDecimal expected = new BigDecimal(COUNT * COST * QUANTITY);
    // setting up a value that can be used to assert the intent
    // of the API documented with the method.
    expected = expected.setScale(2, BigDecimal.ROUND_HALF_UP);
    // assert the intent
    assertEquals(expected, total);
}
```

Now the invoice test subject will be tested without reliance on the LineItem class. Any test failures can more easily be attributed to the Invoice class instead of the LineItem class. An interesting thing to note is that, in this example, none of the actual test methods had to change, only the setUp of the subject. Because we still expect the same behavior and we are initializing with the same data, the set of asserts in the original test is just what we want. This is often the case when putting Mock Objects in place if there was a good set of asserts to begin with.