Chapter 14

Customer Written Tests—

Automating the Acceptance Process

Your goal is to produce software that does what the customer asks for. There are many factors that make this difficult. There can be a gap between what the customer thinks is clearly being expressed and your understanding of their needs. Often, in the process of writing the software, you discover issues that were not made explicit. As the software evolves, the customer thinks of new requirements or wants to make adjustments to existing requirements.

Let's assume that if you could figure out what a customer wants then you could write the code to make it work. How can you be sure you understand what a customer wants? You and your client try to capture your common understanding in a sequence of activities called acceptance tests. The customer should write these tests. That doesn't mean that the client will be writing code. In the framework we'll explore in this chapter, the customer designs HTML tables and you provide a little bit of code to glue the tables to the parts of your application that the tables will be exercising.

Acceptance tests help the customer discover and the developers understand what is expected of a particular user story. In XP, we look at user stories as describing a conversation. They are not hard and fast requirement documents. Acceptance tests are a way of asking the customer how they will determine that you have satisfied the goals of the user story. As with all facets of XP, a customer can add acceptance tests to a story if they think the costs of doing so are justified.

Acceptance tests are more useful if they can be automated. You want to be able to run the tests all the time. Acceptance tests point you in the direction of tasks that can be used to split the user stories. You don't need to write your acceptance tests before you write your production code in the same way that you write unit tests first. Acceptance tests will, however, focus your efforts on the customer requirements.

In this chapter we'll work through a couple of examples of acceptance tests using the Fit framework. Even though it is early on in the life of this framework, there are compelling reasons for considering its use. Fit provides a format for acceptance tests

with two chief advantages. First, customers are comfortable enough with spreadsheets and tables that they can write their own acceptance tests. Second, the approach is formal enough that developers can write the fixtures that tie the tests to the code being tested. In other words, with relatively little work on the part of customers or developers, acceptance tests can be written and run automatically so that both the customers and the developers can track the progress towards completing user stories.

THE FIT FRAMEWORK

There are four pieces to writing and automatically running acceptance tests.

- First, the customer has to write the acceptance tests. In the case of the Fit framework, the customer will be creating tables on a web page using either HTML or a Wiki.
- Second, the developers will write the code that makes the acceptance tests pass. This is the shipping code that the developers will release to the customers every couple of weeks.
- Third, the developers will need to write a little bit of code that maps from the tests the customers write to the application being tested. These fixtures will extend classes in the Fit framework and will exercise the shipping code according to the customer specifications.
- Finally, there needs to be a framework that parses the HTML or Wiki tables and makes the appropriate calls in the test fixtures created in the third step. These are provided by the Fit framework and can be run from the command line or remotely using scripts written in Perl, Python, Ruby, or other languages.

Writing Acceptance Tests as HTML Tables

There are several types of tests that can be run from the Fit framework. We'll take a quick look at three of them. Other types of tests are being developed for the Fit framework, but you can design a wide array of tests using a combination of the three basic types described in this section. In all cases the intent is that the tables express what the customer is interested in verifying in a form that is friendly to both customers and developers.

First, imagine that you've designed a new calculator program that you'd like to test. One test might be something like checking that 36 + 12 = 48. Suppose that you aren't really interesting in testing addition. This first test is designed to test the results of buttons being pushed and information being entered into text fields in a GUI.

enter	36	
Press	plusButton	
enter	12	
press	equalsButton	
check	result	48

A second test may be used when you need to quickly set some values and check the results of performing various actions. For example, we could set our first input to

36, our second input to 12, and then check what we get back for the sum, difference, product, and difference. We could then repeat the test when the values are 30 and 5. Here's what that test could look like in a table form.

firstNumber	secondNumber	sum()	difference()	product()	quotient()
36	12	48	24	432	3
30	5	35	25	150	6

Again, the test is easy to read. In this case the first two columns will be treated as input. The final four columns will call methods with the names that are specified in the first row of the table and the results will be compared with the values specified in the test data row.

A third type of test is used to look at the characteristics of an object. Perhaps you have an object that has been returned by a search and you want to check its state. As a simple example, suppose you have booked a flight from Cleveland to Seattle and have been given the flight locator id 123ABC. Then a table might look like this.

airline	id	departure City	destination City	flight Number	departure Date
BigAir	123ABC	Cleveland	Seattle	429	11/27/02

Of course your record contains more information. We didn't ask for all of the information available in this one table. We could also create a second table that queries some of the other information like this.

airline	id	departure Time	arrival Time	meal Served	movie	number Of Stops
BigAir	123ABC	11:30 EST	13:00 PST	lunch	none	none

These tests show that the name of the airline and the id should identify the record and that the rest of the information can be compared with the data stored in the object.

Supporting the HTML Tables with Fixtures

The next step is to process these tests so that the results can be reported back. To a computer there is no apparent difference between the tables for one type of test and those for another. In Fit, the code that is responsible for interpreting one or more tables is called a fixture. To map a table to a given fixture you'll list the Java class used to process the table in the first row of the table. For example, a subclass of `ColumnFixture` is required to process the calculator example that adds, subtracts, multiplies, and divides. Suppose that this class is called `CalculateThis` and that it is part of the `answers` package. Then your table would look like this.

answers.CalculateThis					
firstNumber	secondNumber	sum()	difference()	product()	quotient()
36	12	48	24	432	3

The CalculateThis class will need to have public variables firstNumber and secondNumber. In our example, we can declare these variables to be ints. CalculateThis will also need public methods named sum(), difference(), product(), and quotient(). The variables don't take any arguments and they return ints that can be compared with the values in the table.

The first calculator example will use the ActionFixture class that comes with the framework. The ActionFixture class uses the keywords start, press, enter, and check. The start keyword points to the class that understands the method names that are in the second column after the words Press, enter, and check. Let's assume that the class in our example is CalculateGUI in the answers package. We have to give names to the methods accepting entered input. Now the table looks like this.

fit.ActionFixture		
start	answers.CalculateGUI	
enter	number	36
press	plusButton	
enter	number	12
press	equalsButton	
check	result	48

The CalculateGUI class then would need no argument methods such as those named plusButton and equalsButton. It would also need a method named number that accepts, in this case, an int. Finally, the result method needs to return an int.

Together the tables and the corresponding classes combine to define the acceptance tests. The fixtures should contain little more than the calls into the production code. The next step is to write the functionality into your application.

Writing the Code to Pass the Tests

Remember that the focus of your efforts is still delivering working code that meets the customer's requirements. The Fit framework is designed to make it easy to write the acceptance tests that call into your code in the same way that JUnit is a framework that allows you to easily write unit tests. Once you are accustomed to the Fit framework, it won't take you long to extend a column or row fixture or to write the code that you need to get an action fixture working.

Acceptance tests should not change the way you write the actual code. Acceptance tests will help you partition the user stories into tasks. Pairs of developers can work on these tasks by writing unit tests and then the code that makes the unit tests pass. Often the acceptance tests can help suggest unit tests that you might write. The fixtures should not determine the interface of the code being tested in the same way that unit tests do. When it is time to deploy your code, you will strip out the acceptance test HTML, Java source, and Java class files.

Running Acceptance Tests with Fit

You need to process your acceptance tests by passing them as input into an application. The runner we'll use in this chapter is FileRunner. You'll run it from the command line by making sure that fit.FileRunner and the classes in your fixture are in the classpath.

You'll also need to pass in the path to the input and output files. We'll look at the actual syntax in depth later in this chapter.

You'll get an indication of the results of processing the tests in the Terminal window. Exceptions often indicate that a variable, method, or class required by the table isn't provided or accessible. The problem also might be with the type returned by the method. You should check that your classpath has been set correctly and make sure that you have compiled your fixture. When bouncing between Java code and HTML, it's easy to forget that HTML only needs to be saved while Java code needs to be both saved and compiled.

The results of running the acceptance tests should be easy to read. The resulting HTML file is color coded to highlight areas that you need to address. Yellow areas are intended for developers. This is where you'll see warnings that certain parts of the acceptance tests haven't been implemented in the fixtures. Red highlighting indicates that the tests ran without exception but that the results weren't what was expected. Not only will the cell be highlighted in red, but you also will see a report that tells you what the actual result was and what the expected result was. Green highlighting is reserved for tests that ran without any problems and returned the expected result.

Exploring Fit

You'll find the Fit wiki at http://fit.c2.com. A wiki is a special web site where every page is editable by anybody. This means that as the community gains more experience with Fit, more examples and helpful observations are added to the site. Page through the site and explore the different styles of acceptance tests that you can write using Fit. On the DownloadNow page, early downloads support Java, .Net, and Python with plans to support Lisp, Ruby, Perl, and C++. In addition, Bob and Micah Martin have released Fitnesse at www.fitnesse.org. Fitnesse provides a wiki framework for easily writing and running acceptance tests.

For this chapter you'll need the Java version of the framework. Download it and expand it. You only need the fit.jar file to run the framework, but you'll find the supporting documentation helpful. Source files are included for the framework in the fit package and for several of the examples that you'll find on the site in the eg package. Under the Documents directory you'll find some of the pages that can also be found on the site. Because the on-line site is a wiki, the pages you download are a snapshot of the site at a given time. The Reports directory contains some of the acceptance test pages from the wiki. These can be processed locally from the command line. On the wiki, the tests can be processed by clicking a hyperlink to a CGI script.

While on-line, follow the Cook's tour for developers. Get a quick view of the Row, Column, and Action fixtures. Take a moment to look at the *Field Guide To Fixtures* to figure out which fixtures are best suited to different tasks that you are trying to accomplish. You'll notice that the advice for *Making Fixtures* is similar to the advice for using JUnit to write test first code. The ideas in Fit and JUnit are different but related. With JUnit you write the test first and then write the code that makes the test code compile and then pass. With Fit your customer writes the test. You write the fixture that makes the test not throw any exceptions. Then you write the code that gets the tests to pass. The differences in intent and practice will become more evident as you play with the two testing frameworks.

FROM USER STORIES TO ACCEPTANCE TESTS

Because we haven't had a way of automating acceptance tests, until now, much of our experience has been with customers playing with the application and saying, "That looks right." Sometimes the customer is able to formalize the steps we need to take to test the code. Even in these cases, because the tests needed to be run manually, they weren't run frequently.

Fit allows us to automate the processing of our acceptance tests. This means that the developers can run the tests while working on a user story to help measure progress. A customer can also either run the tests or view the results of the most recently generated tests. You'll run the tests by passing arguments to the Java class `fit.FileRunner` to specify the input HTML file and the corresponding location and name of the output file.

Test results can be easily published. Since you are generating HTML files, these can be shared with the customer by putting them on a web site or attaching them to or including them in an e-mail. You can also customize a CGI script or create a Java server page (JSP) or servlet to allow the developers or the customers to run the acceptance tests remotely by clicking a link on a web page. As the community explores Fit, more of this will be available to you in an easily downloadable form.

Our First Example User Story

Let's begin with a user story that tests that the value returned in a transaction is correct. Our customer is Brian, the information technology (IT) manager of a fictional grocery chain called Cheaper Buy the Dozen. Brian has explained that each item in the story has a unit amount. Any time a customer buys twelve of any item, they get a 5% "case" discount on that set of twelve items.

Brian captures this as the following user story.

Case Discount

The price for multiple copies of the same item is the number of items multiplied by the unit cost with a 5% discount on any set of twelve purchased.

Initial Acceptance Tests for the Case Discount Story

We think we know what Brian means, but we ask him to come up with acceptance tests that will tell him that we've done what he wants. He thinks a moment, and says, "Let's say that a pound of coffee costs $8.00. If I buy one pound then the item total should be $8.00. If I buy five pounds then the item total should be $40.00. If I buy twelve pounds then we need to figure out the discount. Multiplying $8.00 times twelve gives us $96.00. Now 5% of $96 is $4.80. This means that the price for a dozen pounds of coffee is $96 minus the $4.80, or $91.20."

"Okay," you suggest, "Why don't we summarize your requirements in this table."

Unit Price	Number of Items	Item Total
$8.00	1	$8.00
$8.00	5	$40.00
$8.00	12	$91.20

"That's nice," says Brian. "Now that I see it like this, I think there's a couple of cases I'd like to add."

"Like what?" you ask.

"Well, if I buy more than one dozen but less than two dozen, I want to make sure I only get a discount on the first dozen. So if I buy seventeen pounds of coffee I should get $131.20—the price with a discount for the first dozen plus the price for the next five pounds."

"What else?" you prompt.

"Well, I want to make sure that I get discounts correctly calculated if I buy two dozen or more. So let's check that the price for two dozen pounds is twice $91.20, or $182.40, and that the price for twenty-nine pounds is $40 more than that, or $222.40. While we're at it, lets check that the price for one hundred dozen pounds is $9,120.00. It's kind of ridiculous—we'd never sell this quantity out of our stores but it will make me feel better that we're getting the calculation correct."

"Great," you agree, "This will check our logic on the dozen discounts pretty thoroughly."

"There's one more case I'd like to check," says Brian.

"What's that?" you ask.

Brian answers, "I'd like to see what happens if 5% of the unit price is a fraction of a cent. Let's say I sell candy corns at five cents a piece. A dozen of them would be $0.60 minus the 5% discount of $0.03. Oh, I guess that's okay."

"No," you chime in, "I see where you're going. If you sell them for four cents a piece, then a dozen of them would be $0.48 minus the 5% discount of $0.024. Now you'd end up pricing them at $0.456 per dozen."

"Right," answers Brian, "I'd really like them to be priced at $0.46 per dozen. It sounds small, but these differences add up."

"So," you summarize, "if all of your prices end in a five or a zero this will never happen, but if they end in other digits then we'll have to deal with these fractions of a penny."

Brian thinks another moment and answers, "I want to leave flexibility in pricing, so let's add your case in as a test case. Whenever we get a fraction of a penny, I'd like to round up."

"Okay," you agree, "would you mind if we give prices in terms of pennies in our tests?"

"No, that's fine," says Brian. You present him with the following updated table.

Unit Price in Pennies	Number of Items	Total Price in Pennies
8 00	1	8 00
8 00	5	40 00
8 00	12	91 20
8 00	17	131 20
8 00	24	182 40
8 00	29	222 40
8 00	1200	9 120 00
4	12	46

Benefits of Having Acceptance Tests

Compare your current understanding of what Brian wants with what you understood when you first saw this user story.

Case discount

The price for multiple copies of the same item is the number of items multiplied by the unit cost with a 5% discount on any set of twelve purchased.

Armed with Brian's tests, you now have a much better idea of how the case discount should apply in his stores. More than that, you now have an agreement with him that when you have passed all of these tests, he will consider this story completed. As with all other aspects of XP, Brian can change the requirements or add to the tests, but he will take the costs into consideration before doing so.

The table also helps Brian understand what he needs to see in order to consider this user story complete. If he can quickly look at this table and see which of these cases are passing and failing, he'll have a good indication of the status of this user story. We used JUnit to run our unit tests and indicate which tests failed and why. As smart as Brian is, he's not a developer. We should be able to present the results to him in a customer-friendly way. The strategy of the Fit framework is simple. If a result is correct, it is colored green. If it is incorrect it is colored red and the returned and the expected values are displayed for the customer to inspect. If the fixture isn't yet in place to support the acceptance tests then the cell will be colored yellow with a message designed for developers not customers. We'll see examples of all three of these test results soon.

FORMALIZING THE ACCEPTANCE TESTS

The Fit framework is designed to take tables like the one we created in the last section as input. There are several types of fixtures used to process tables differently. We'll begin with the column fixture. We'll tweak the table we created so that we can process it using the Fit framework. We'll then begin to process the table and create the fixture needed to respond to these inputs.

Mapping to the Fixture

The main advantage of the table we set up with Brian is that it clearly communicates to him and to us what his requirements are. In the table the first column is an integer that

represents the price of a single unit and the second column is an integer that represents the number of units purchased. You can view the contents of these columns as inputs into some method responsible for determining the total cost for that item. The third column is a bit different. It contains the expected value of the total cost for the item. You can think of it as the expected return value for the method.

For the column fixture, these are the two types of columns. Either a column corresponds to a variable in the corresponding Java class or it corresponds to a method. The values of the variables are read from the table. The values in the method column are read from the table and compared with the value of the corresponding methods. We indicate methods by ending their names with parentheses.

In our example, let's create an HTML file called CaseDiscountFitTest.html. Let the corresponding Java class be called CaseDiscountFixture.java and place it in a package called register. It is possible that several tables use the same fixture or that a single HTML file contains more than one table that uses different fixtures. Each table needs to contain identification of the fixture that is intended to process it. The top line of a table is the name of the fixture used to process it. To make it look nicer, we use the `colspan` attribute to force the top row to span all of the columns. Our current example looks like Figure 14-1.

Acceptance Tests for User Story: Case Discount

The price for multiple copies of the same item is the number of items multiplied by the unit cost with a 5% discount on any set of twelve purchased.

register.CaseDiscountFixture		
unitPrice	numberPurchased	itemTotal()
800	1	800
800	5	4000
800	12	9120
800	17	13120
800	24	18240
800	29	22240
800	1200	912000
4	12	46

Figure 14-1 Acceptance Tests for User Story: Case Discount

Notice that the first line of the table includes the qualified name of the Java class, `register.CaseDiscountFixture`. The first two columns are labeled `unitPrice` and `numberPurchased` and, therefore, correspond to public variables in the `CaseDiscountFixture` class. The third column is labeled `itemTotal()` and corresponds to a public method in the `CaseDiscountFixture` class.

The client is expected to be able to produce these tables. They can use a spreadsheet program or word processor to produce the tables and export the HTML. They can write the actual HTML itself. Even the most nontechnical clients can easily follow the template to create their own tables. In our case, Brian quickly created the following HTML that generated the page in Figure 14-1.

```
<html>
  <head>
    <title> Acceptance Tests User Story: Case Discount </title>
  </head>
  <body>
    <h1> Acceptance Tests for User Story: Case Discount </h1>
    <p> The price for multiple copies of the same item is the
        number of items multiplied by the unit cost with a 5%
         discount on any set of twelve purchased.
    </p>
    <table BORDER>
      <tr>
        <td colspan = 3> register.CaseDiscountFixture </td>
      </tr>
      <tr>
        <td> unitPrice </td>
        <td> numberPurchased </td>
        <td> itemTotal() </td>
      </tr>
      <tr>
        <td> 800 </td>
        <td>  1 </td>
        <td> 800 </td>
      </tr>
      <tr>
        <td> 800 </td>
        <td>  5 </td>
        <td> 4000 </td>
      </tr>
      <tr>
        <td> 800 </td>
        <td>  12 </td>
        <td> 9120 </td>
      </tr>
      <tr>
        <td> 800 </td>
        <td>  17 </td>
        <td> 13120 </td>
```

```
      </tr>
      <tr>
        <td> 800 </td>
        <td>  24 </td>
        <td> 18240 </td>
      </tr>
      <tr>
        <td> 800 </td>
        <td>  29 </td>
        <td> 22240 </td>
      </tr>
      <tr>
        <td> 800 </td>
        <td>  1200 </td>
        <td> 912000 </td>
      </tr>
      <tr>
        <td> 4 </td>
        <td>  12 </td>
        <td> 46 </td>
      </tr>
    </table>
  </body>
<html>
```

Running the Tests

We will process the HTML file by running the `FileRunner` class in the `Fit` package. We can do this from the command line or from within an IDE. When you first downloaded the Fit framework, you should have run the setup tests specified on the `RunMeFirst` page of the Wiki. To run FileRunner you had to specify the classpath and an input file and an output file. Let's look at each of those parts in turn.

We'll begin by creating a directory named "dozen." Inside of this new directory we'll copy the fit.jar file and create additional directories labeled classes, docs, src, and tests. All source files are compiled into the classes directory. Within the tests directory we'll create subdirectories labeled acceptance and unit to hold the source files. At this point we only created a single file named CaseDiscountFitTest.html. Place it inside of /dozens/tests/acceptance/testsource. Our dozen directory should look like Figure 14-2.

Your classpath needs to include two things. First it must point to the classes that make up the Fit framework. In the `RunMeFirst` example you just pointed at the Classes directory. You could also just use the fit.jar file and include it in the classpath. The classpath must also include the fixtures used to process the tables. In this setup, the classpath should just include fit.jar and the classes directory.

The input file is CaseDiscountFitTest.html. We'll start by running the acceptance tests from the command line just inside the dozen directory. This means that the path to the input file is tests/acceptance/testsource/CaseDiscountFitTest.html. We'll write the output to a file of the same name in the testresults directory. This means that the path to the output file is tests/acceptance/testresults/CaseDiscountFitTest.html.

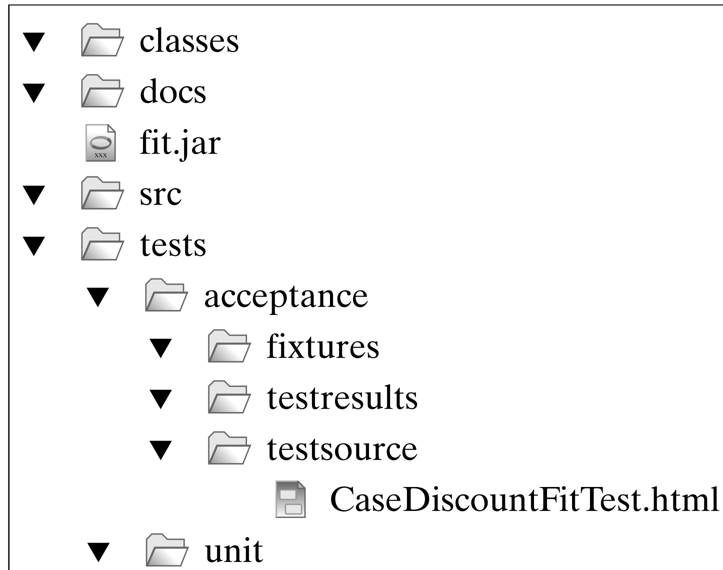

Figure 14-2 The Dozen Directory

Now we can run the acceptance tests. Open a terminal window and navigate inside of the dozen directory. Run the acceptance tests with this command:

```
java -classpath fit.jar:classes fit.FileRunner
  tests/acceptance/testsource/CaseDiscountFitTest.html
  tests/acceptance/testresults/CaseDiscountFitTest.html
```

On a Windows machine replace : with ; and replace / with \. We get the following feedback in the terminal window.

```
0 right, 0 wrong, 0 ignored, 1 exceptions
```

This doesn't surprise us. The table specifies that the fixtures needed to run the tests are found in the class `register.CaseDiscountFixture`. So far, this class does not exist. In fact, if we look at the generated HTML file in the testresults directory, we can see that it provides us with this information (see Figure 14-3).The top row of the table is presented with a yellow background color to indicate to developers that something hasn't yet been implemented. In this case a `ClassNotFoundException` is reported because the framework can't find the class `register.CaseDiscountFixture`. Our next step is to take care of this exception.

Creating a Stub Fixture

As a first step, inside of the /tests/acceptance/fixtures directory create a subdirectory named register. Inside of this register directory, create a file CaseDiscountFixture.java. The class `CaseDiscountFixture` needs to extend the class `fit.ColumnFixture` in order to properly process our table. Our first version of CaseDiscountFixture.java looks like this.

register.CaseDiscountFixture

```
java.lang.ClassNotFoundException: register.CaseDiscountFixture
        at java.net.URLClassLoader$1.run(URLClassLoader.java:195)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:183)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:294)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:281)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:250)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:115)
        at fit.Fixture.doTables(Fixture.java:69)
        at fit.FileRunner.process(FileRunner.java:29)
        at fit.FileRunner.run(FileRunner.java:22)
        at fit.FileRunner.main(FileRunner.java:17)
```

unitPrice	numberPurchased	itemTotal()
800	1	800
800	5	4000
800	12	9120
800	17	13120
800	24	18240
800	29	22240
800	1200	912000
4	12	46

Figure 14-3 Message Not Created Yet

```
package register;
import fit.ColumnFixture;
public class CaseDiscountFixture extends ColumnFixture {
}
```

Compile CaseDiscountFixture.java so that the .class file is written to the classes directory. Now we can rerun the Fit framework as before. This time the results are a bit different. You can view the generated HTML page for more details on the test results, but you should see this display in the terminal window.

```
0 right, 0 wrong, 24 ignored, 3 exceptions
```

It may seem that things are getting worse—but we are actually making progress. Before creating the `CaseDiscountFixture` class we had no ignored tests and now there are twenty-four. This actually indicates that now the tests are being found. Earlier there was one exception and now there are three. The initial version of `CaseDiscountFixture` also addressed the `ClassNotFoundException`. Now we have one `NoSuchFieldException` for both `unitPrice` and another for `numberPurchased`, and a `NoSuchMethodException` for `itemTotal()`.

As a next step, let's add variables for unitPrice and numberPurchased. The variables need to be public as they are being called from outside of the register package. They need to be compatible with ints because we're passing in integral values. Add the highlighted lines to CaseDiscountFixture.java.

```
package register;
import fit.ColumnFixture;
public class CaseDiscountFixture extends ColumnFixture{
  public int unitPrice;
  public int numberPurchased;
}
```

Now, compile the code as before and rerun the Fit framework. This time the results reported in the terminal window are predictably better.

```
0 right, 0 wrong, 8 ignored, 1 exceptions
```

As before, the generated HTML page contains more details if you need them. The itemTotal() cell is colored yellow and contains the stack trace for the NoSuchMethodException.

What remains is to address the NoSuchMethodException. You could make a good argument in favor of doing this now or leaving it for later. On the side of leaving it as is, the developers and the client can clearly see by this report that itemTotal() has not been implemented yet. No one needs to be worried about tests that pass or fail, because we haven't gotten around to this task yet. On the other hand, we could create a trivial version of itemTotal() and most or all of the tests would fail. Failing tests also indicate an area that needs to be addressed.

Stubbing out itemTotal() shouldn't take us very long. In one sense, it's as simple as creating a public method named itemTotal() that takes no parameters and returns an int. We do want to be careful to make sure that we aren't putting in the logic to make these tests pass. The CaseDiscountFixture class is part of the testing framework. It should defer the programming logic to classes that will become part of the production code. As Bob Martin notes on the Fit framework Wiki, acceptance tests are not the same as unit tests. Our task in the fixture isn't to get one after another of the acceptance tests running. For now we'll return the number –1 from itemTotal(). Here's the new version of CaseDiscountFixture.java.

```
package register;
import fit.ColumnFixture;
public class CaseDiscountFixture extends ColumnFixture{
  public int unitPrice;
  public int numberPurchased;
  public int itemTotal(){
    return -1;
  }
}
```

Again, compile the revised version of CaseDiscountFixture.java and run the acceptance tests. This time this message in the terminal window lets us know that all of the tests are run, there are no exceptions, and that all of the tests have failed.

```
0 right, 8 wrong, 0 ignored, 0 exceptions
```

The generated HTML file shown in Figure 14-4 highlights all of the entries in the third column in red. You can see that each of the cells in that column is split into two pieces: one for the expected result that was entered in the table and the other for the actual result that was returned from the method.

Writing the Production Code

All of the production code that you write should have a written test first. This means that you'll need to write unit tests in addition to the acceptance tests. Unit tests specify the interface of the code being tested in a way that isn't possible or desirable with acceptance tests. You don't want to allow your customers to make technical decisions about how the code should be written. These decisions often come out of a suite of unit tests. Your customer is making business decisions about what your software should do. You should consult the unit-testing tutorial in Chapter 10 for more details on that process.

In this case, the acceptance tests suggest something about the structure of the code. The tests themselves lead us to unit tests that we might write. Also, the acceptance test fixtures need to call into the production code somewhere. You'll find it best if the test fixture is in the same package as the code it's calling into. This means that you will minimize the number of changes to the access level of methods being called.

Let's create the class `ItemTest` inside of the `register` package. We'll create a new object of type `Item` that has a `unitPrice` of 800 pennies just as in our acceptance tests. We'll add one item to our purchases and verify that the total price for this item is 800. Here's the code for ItemTest.java.

```
package register;
import junit.framework.TestCase;
public class ItemTest extends TestCase{
  public void testCostOfSingleItemIsUnitPrice(){
    Item item = new Item(800);
    item.addToOrder(1);
    assertEquals(800,item.totalItemCost());
  }
}
```

We go through the usual steps of writing enough code until this class compiles. This requires that we create an `Item` class that has a constructor that takes an `int` representing the unit price as a parameter. It also requires that we create an `addToOrder()` method that takes an `int` representing the number purchased as a parameter. Finally, this unit test implies that there must be a method called `totalItemCost()` that has no parameters and returns an `int` representing the total cost of purchasing the prescribed number of copies of this particular item. After a little refactoring, the code for Item.java looks like this:

```
package register;
class Item {
  private int unitPrice;
  private int numberPurchased;
  Item(int unitPrice) {
```

register.CaseDiscountFixture		
unitPrice	numberPurchased	itemTotal()
800	1	800 *expected* -1 *actual*
800	5	4000 *expected* -1 *actual*
800	12	9120 *expected* -1 *actual*
800	17	13120 *expected* -1 *actual*
800	24	18240 *expected* -1 *actual*
800	29	22240 *expected* -1 *actual*
800	1200	912000 *expected* -1 *actual*
4	12	46 *expected* -1 *actual*

Figure 14-4 Initial Failing Tests

```
    this.unitPrice = unitPrice;
  }
  void addToOrder(int numberOfAdditionalItems){
    numberPurchased += numberOfAdditionalItems;
  }
  int totalItemCost() {
    return numberPurchased * unitPrice;
  }
}
```

Tying the Fixture to the Production Code

Now that we are passing one unit test, our `Item` class has taken shape enough that we can tie our acceptance test fixture to it. We could have waited until this point to fill out the `itemTotal()` method in `CaseDiscountFixture` instead of having returned a –1 to avoid the `NoSuchMethodException`. Now that we know how to fix `itemTotal()`, we initialize a new `Item`, add to the number of items purchased, and then return the total price for that item as calculated by the instance of `Item`.

It is important to note that the `ColumnFixture` class is very thin. Its job is to call into the main codebase. This should feel like writing a good GUI for a program. Very little of the business logic should sit in the GUI. A second note is that if you find yourself needing to place some amount of logic in the fixture, then you need to write this code test first as well.

We've now tied the acceptance test through its fixture to the production code. The altered code is highlighted below.

```
package register;
import fit.ColumnFixture;
public class CaseDiscountFixture extends ColumnFixture{
  public int unitPrice;
  public int numberPurchased;
  public int itemTotal(){
    Item item = new Item(unitPrice);
    item.addToOrder(numberPurchased);
    return item.totalItemCost();
  }
}
```

Now when you rerun the acceptance tests, you'll see that two of the acceptance tests are now passing. Here's the message in the terminal window.

```
2 right, 6 wrong, 0 ignored, 0 exceptions
```

Figure 14-5 shows the results that you can see in the generated HTML file. Now the tests that passed have a green background while the tests that failed have a red background and list the expected and actual values.

Passing More Acceptance Tests

We can see from the acceptance test results that we are only passing those tests when we purchase fewer than a dozen items. We need to create a more complex pricing

register.CaseDiscountFixture		
unitPrice	numberPurchased	itemTotal()
800	1	800
800	5	4000
800	12	9120 *expected* 9600 *actual*
800	17	13120 *expected* 13600 *actual*
800	24	18240 *expected* 19200 *actual*
800	29	22240 *expected* 23200 *actual*
800	1200	912000 *expected* 960000 *actual*
4	12	46 *expected* 48 *actual*

Figure 14-5 Test Indicates What Needs Fixing

scheme than just returning the product of the unit price and the number of items purchased. Of course, before we add any production code we have to write another unit test. Let's just see what will happen if we buy a dozen of an item. We'll add that test to `ItemTest.java` and refactor the test code a little by adding a `setUp()` method. Here's ItemTest.java.

```
package register;
import junit.framework.TestCase;
public class ItemTest extends TestCase{
  private Item item;
  protected void setUp(){
    item = new Item(800);
  }
  public void testCostOfSingleItemIsUnitPrice(){
    item.addToOrder(1);
    assertEquals(800,item.totalItemCost());
  }
  public void testCostOfADozenIsDiscounted(){
    item.addToOrder(12);
    assertEquals(9120,item.totalItemCost());
  }
}
```

Everything compiles and runs, but the second test fails. The discounted cost should be 9120 but the calculated cost remains 9600. We return to `Item` and add the logic to factor in the discount. We quickly get the test to pass and then refactor the code so that it looks like this:

```
package register;
class Item {
  private int unitPrice;
  private int numberPurchased;
  private double percentDiscount = .05;
  Item(int unitPrice) {
    this.unitPrice = unitPrice;
  }
  void addToOrder(int numberOfAdditionalItems){
    numberPurchased += numberOfAdditionalItems;
  }
  int totalItemCost() {
    return totalCostWithoutDiscount()
           - discountForDozensPurchased();
  }
  private int numberPurchasedByTheDozen() {
    return 12 * ((int) (numberPurchased/12) );
  }
  private int totalCostWithoutDiscount(){
    return unitPrice * numberPurchased;
  }
  private int discountForDozensPurchased(){
    return ( (int) (unitPrice
                 * numberPurchasedByTheDozen()
                 * percentDiscount));
  }
}
```

Notice that `Item` has no public methods or variables. In fact, `Item` itself isn't public and doesn't have a public constructor. We can expose this class and its methods later if the need arises. The classes used by JUnit and Fit have to be public. The Fit variables and methods that are referenced by the HTML tables also must be public. By placing the class that extends the `ColumnFixture` class in the same package as `Item`, we aren't required to expose very much of the class being tested. At this point our directory structure looks like that shown in Figure 14-6.

With test-driven development we run unit tests all the time. We write a little code, compile, and then run the unit tests continuously. How often should we run the acceptance tests? The advantage of having automated acceptance tests is that we can run them frequently. We have a suspicion that by passing the last unit test, we've probably passed all but the last acceptance test. When we run the acceptance tests to check where we are we get the following surprising result.

```
8 right, 0 wrong, 0 ignored, 0 exceptions
```

In the `discountForDozensPurchased()` method we inadvertently rounded the amount of the discount down to the nearest penny when casting the calculated value back to an int. This had the effect of rounding the total item cost up to the nearest penny as Brian had specified. By running the acceptance tests we saved ourselves some time and work by creating functionality that already existed.

The Client and the Acceptance Tests

We have completed our work on Brian's first user story and can demonstrate our success. We can now go back to Brian and show him his acceptance tests running. He accepts that somehow we've taken care of all of his goals in this particular user story. If he'd like, he can add another test or two if he feels there's some area of this story that he's left unexplored. For now, he accepts that we've done what we promised, but he doesn't seem satisfied.

"What's wrong?" you ask.

"I don't know," Brian answers, "it just feels like magic."

"What do you mean?"

"Well," says Brian, "I came up with some tests and you said you'd make them all pass. Now you show me a third column colored green and tell me that you've made them all pass. How do I know you didn't just color the third column green?"

"Well, I could change the code back to when everything was breaking," you suggest.

Brian isn't satisfied with this answer. "No," he says, "that wouldn't help. I want a test that convinces me the system really is calculating the values correctly."

It's hard not to take Brian's objection personally. It sounds as if he's saying that he doesn't trust you. He's not saying that—he's saying that the passing acceptance tests you showed him aren't convincing him right now. He needs reassurance that this process really is testing what you say it's testing. Brian is an expert in his domain. You've asked him to sign off on a user story when his acceptance tests are passing and it is the automation that is giving him pause.

Let's look back at how we got to this point. You asked Brian to come up with test cases that would show that you were correctly calculating the cost of one or more of a

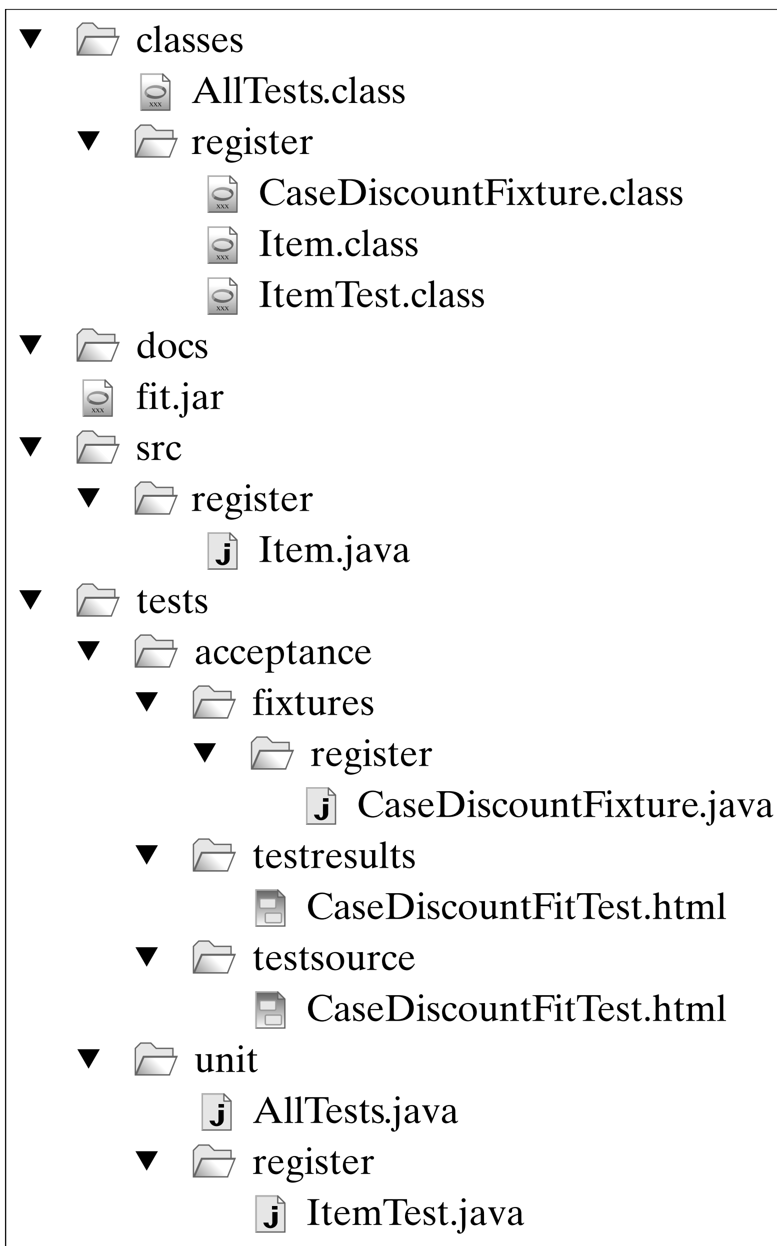

Figure 14-6 Directory Structure

particular item. He came up with a representative set of test cases and then you told him that you would run them for him. This should remind you a bit of a strategy we employed when unit testing. Often we need to see a test fail before we can make it pass—just to have confidence that the test is exercising the code it is intended to test. Brian never got to see failing tests. We'll show Brian a failing test and then we'll set up a system that will help him see tests during the process.

You tell Brian, "I have an idea."

"What?" he asks.

"What if you change the HTML file so that the third column isn't the value you expect. Then the tests should fail in those rows."

"That's great," says Brian. "I can change the HTML myself and then we'll run the tests. Let's change the first three. Coffee still costs 800 pennies for one pound. When I buy one pound let's change the third column to say that the total cost should be 100, and when I buy five pounds let's change the third column to 500. Then let's leave the total amount alone in the third row, but change the amount purchased to three dozen pounds. Now let's rerun the tests."

You open up a terminal window and rerun the tests. As expected you get this result.

```
5 right, 3 wrong, 0 ignored, 0 exceptions
```

"Whoa," says Brian. "That's really nice."

"What do you mean?" you ask.

"That summary. I've never seen it before."

"I've shown you the web page where you can look for green and red to see which tests pass."

Brian says, "That's great, but I'd also like to see the summary. Eventually we're going to have a lot of tests. If everything is passing, I'd rather not have to look through all of the tables."

"Okay," you answer, "it's actually pretty easy to do. We just need to add a table to the end of the page with a single cell that contains `fit.Summary`. When the page is processed this will be replaced with the summary data for this page."

Brian adds the highlighted lines below to the end of CaseDiscountFitTest.html.

```
    </table>
    <h1> Summary of tests run on this page </h1>
    <table BORDER>
      <tr>
        <td> fit.Summary </td>
      </tr>
    </table>
  </body>
<html>
```

We rerun the tests and view the resulting web page (see Figure 14-7). Brian is beaming. The three incorrect values are highlighted in red, as he wanted. He can see the difference between the actual and expected values for the first three rows. The third row also correctly lists the total expected for three dozen pounds, even thought that's not a value he had originally tested. At the bottom of the page is the summary that he requested.

Summary of tests run on this page

fit.Summary	
counts	5 right, 3 wrong, 0 ignored, 0 exceptions
input file	/Users/daniel/Dev/dozen/tests/acceptance/testsource/CaseDiscountFitTest.html
input update	Tue Nov 26 10:19:18 EST 2002
output file	/Users/daniel/Dev/dozen/tests/acceptance/testresults/CaseDiscountFitTest.html
run date	Tue Nov 26 10:19:25 EST 2002
run elapsed time	0:00.17

Figure 14-7 Summary of Tests Run on This Page

"Really," says Brian, "I'd like the summary at the top of the page." You start to explain to him that this won't work and then stop. Let him try it. You know that he'll be more convinced that the framework is doing something by exploring a bit. He moves the `fit.Summary` table to the top of the page and gets the results that there were 0 right, 0 wrong, 0 ignored, and 0 exceptions.

"Hang on," says Brian, "it doesn't look like anything ran." He thinks for a moment and then gets it. "I can't have the summary before the tests are run because at that point nothing has run. There are no results to report."

Brian then puts the HTML file back in order. He corrects the table entries so that the values are correct and he puts the `fit.Summary` table back on the bottom. He reruns the tests and sees that all of the tests have passed. With this summary he can tell how recently the tests were run and what the current status of this story is.

Brian asks you to run the tests once a day and post the HTML files to a web site that he's set up. We agree that any time new code is integrated, we'll run the acceptance tests and that once a day we'll post the results to his web site. We also make a note to either use a variant of the CGI script on the Fit wiki site or a JSP page or servlet to allow Brian to process the tests remotely whenever he likes.

TESTING GUIs

"Now," says Brian, "let's work on the actual register. For the most part the cashiers will be scanning the items in and the register will be keeping a running record. If a cashier scans in an item that the register doesn't recognize, then the cashier needs to be able to enter the unit price and the number of items purchased."

"Actually," you respond, "that sounds like two stories. Story one has to do with having some list that matches bar codes, item numbers, and unit prices so that the register

can scan a bar code and know the item and its price. Story two has something to do with items not on the list being manually entered."

"Maybe," says Brian, "but I would think there are more stories than that. I'd like to keep the list that matches bar codes to items separate from the list that matches items to prices. I may be updating the price list pretty regularly. Then I want to give the cashier some indication if either the item can't be scanned or it can be scanned but there's no price on file for it. Finally, I want to allow the cashier to manually enter prices for any item."

"That seems like a lot," you say.

"It is," agrees Brian. "Let's start with the last one. If everything else fails, I can still provide each cashier with a price list on paper and let them enter the amounts by hand." He writes down this user story.

Miscellaneous Item

A user can enter an item by hand by pressing the "MISC" button and then entering the unit price and number of items purchased.

Designing the Acceptance Tests

With this starting place, you are ready to have Brian flesh out the acceptance tests. Then you can write the fixtures and the code that will make them pass. Remembering your earlier experience, you are careful to involve Brian in this process.

"Describe the process to me," you prompt.

"Well, they will hit the 'Misc' button and they will see a prompt that says 'Enter Unit Price'. They will then enter the unit price. The display will reflect this unit price. Next, they'll press the 'Enter' button. They'll then see a prompt that says 'Enter number of items.' They will then enter this number. The display will reflect this number. Then they'll press the 'Done' button."

"Then what?"

"Then, they'll see the total cost for that item. Then they can enter another item or finish the transaction by pressing the 'End' button."

You show him a table that you think captures what he just said.

Press	miscButton	
Check	display	Enter Unit Price
Enter	unitPrice	800
check	display	800
press	enterButton	
check	display	Enter number of items
enter	numberOfItems	
check	display	5
press	doneButton	
check	display	4000
check	totalCost	4000
press	endButton	

Brian thinks that the table looks pretty good. He suggests two changes. First, he would like a running total to be available in a separate display and second, he wants to make sure you can handle more than one item in a transaction and more than one transaction.

You and Brian reread the user story.

Miscellaneous Item

A user can enter an item by hand by pressing the 'MISC' button and then entering the unit price and number of items purchased.

You and Brian agree that this story is about entering a single item and defer the additions that have to do with handling more than one item in a transaction and keeping track of a running total.

Stubbing Out a Fixture for the Tests

You can see that this table is different than the one we worked with earlier. In that one each column corresponded to variables that would be input into the system being tested and methods that included the value that we expected to be returned from the method. In this one the first column consists of one of four keywords: start, press, enter, or check.

The start keyword is accompanied, in the second column, by the name of a class that is instantiated. The rest of the commands will be handled by this instance of the class. The press keyword is meant to simulate a button press. The second column contains the name of the method called by the press. The enter keyword is meant to simulate entering values into a GUI via a text field. The second column contains the name of the method corresponding to the text field and the third column contains the value being entered. Finally, the check keyword indicates that this row is being tested against a value returned by the fixture. The second column contains the name of the method that will return the value and the third column contains the value being compared.

Our first step is to make a few adjustments to the table. With Brian's help, we add the two lines at the top of the table to specify the class used to process the table and the class that is instantiated to run the methods listed in the table. Brian also decides to tweak the process a little bit. The revised table looks like this.

fit.ActionFixture		
start	register.MiscItemFixture	
press	miscButton	
check	display	Enter Unit Price
enter	unitPrice	800
check	display	800
press	enterButton	
check	display	Misc Grocery 800

press	timesButton	
check	display	Enter number of items
enter	numberOfItems	5
check	display	5
press	doneButton	
check	display	4000
check	totalCost	4000

We'll create the file MiscItemFixture.java in the /tests/acceptance/fixtures/register/directory. It must extend the `Fixture` class in the `fit` package. We can quickly stub it out with the methods that it needs to contain. You can read those off of the table one at a time.

```
package register;
import fit.Fixture;
public class MiscItemFixture extends Fixture {
  public void miscButton(){}
  public void enterButton(){}
  public void doneButton(){}
  public void timesButton(){}
  public int totalCost(){
    -1;
  }
  public String display(){
    return "";
  }
  public void unitPrice(int unitPrice){}
  public void numberOfItems(int numberOfItems){}
}
```

Compile this class and run the acceptance tests in the file MiscItemFitTest.html. The report is that none of the tests are right, seven of them are wrong, and that no exceptions were thrown. In an `ActionFixture` the tests correspond to the rows that begin with the keyword check. Our next step is to write the code that gets these tests to pass.

Problems Passing the Acceptance Tests

We'll pass the acceptance tests in a way that has problems not caught by these tests. Don't skip to the next section where a more correct solution is presented. It's important to see the problems that can arise when you take care of programming logic in your acceptance test fixtures.

Let's first look at the production code class `ManualEntry`. It was written using the test first method to take care of many of the tasks specified in the acceptance tests. Only the highlighted areas are used for real input or calculation. Everything else is used to set the display or access variables.

```
package register;
class ManualEntry {
```

```
  private String display;
  private Item currentItem;
  ManualEntry(){
    setDisplay("Enter Unit Price");
  }
  void setDisplay(String display){
    this.display = display;
  }
  String getDisplay(){
    return display;
  }
  void createItemWithPrice(int price){
    currentItem = new Item(price);
    setDisplay("Misc Grocery " + price);
  }
  void buyMoreThanOne(){
    setDisplay("Enter number of items");
  }
  void setNumberOfItems(int numberPurchased){
      currentItem.addToOrder(numberPurchased);
     setDisplay(""+ getTotalCost());
  }
  int getTotalCost(){
    return currentItem.totalItemCost();
  }
}
```

Here's the test fixture that calls into `ManualEntry`. Notice that it does a little more than it should. Here's the code for MiscItemFixture.java.

```
package register;
import fit.Fixture;
public class MiscItemFixture extends Fixture{
  private ManualEntry manualEntry;
  private int unitPrice;
  private int numberOfItems;
  public void miscButton(){
    manualEntry = new ManualEntry();
  }
  public void enterButton(){
    manualEntry.createItemWithPrice(unitPrice);
  }
  public void doneButton(){
    manualEntry.setNumberOfItems(numberOfItems);
  }
  public int totalCost(){
    return manualEntry.getTotalCost();
  }
  public String display(){
    return manualEntry.getDisplay();
```

```
    }
    public void unitPrice(int unitPrice){
      this.unitPrice = unitPrice;
      manualEntry.setDisplay(""+unitPrice);
    }
    public void numberOfItems(int numberOfItems){
      this.numberOfItems = numberOfItems;
      manualEntry.setDisplay(""+numberOfItems);
    }
    public void timesButton(){
      manualEntry.buyMoreThanOne();
    }
  }
```

All of the tests pass. The unit tests pass and the acceptance tests pass. For the most part, the acceptance tests call into the class being tested. There are two main exceptions. Consider the code for the methods `unitPrice()` and `numberOfItems()`. They both do two things that should raise a red flag: they set the value of an instance variable and they explicitly set the value of a variable instead of calling methods in the `ManualEntry` class. The acceptance test has separated the steps of typing in the unit price and pressing the enter button. These steps should be handled in our production code and not in our test fixture.

This example along with its mistakes may seem a little artificial. Of course, you reason, the functionality of those methods has to be in `ManualEntry` and not in the fixture. On the other hand, it is easy to make this mistake. You need to be vigilant and keep any logic that belongs in your production code out of your test code.

The Revised Fixture and Production Code

Before reading the code presented in this section, try to refactor MiscItemFixture.java to eliminate the instance variables `unitPrice` and `numberOfItems`. One technique is to comment out `unitPrice` and see what else needs to change. You'll see that you have to change the way you implement the methods `enterButton()` and `unitPrice()`. Similarly, when you comment out `numberOfItems` you'll find that you need to change the way you implement the methods `doneButton()` and `numberOfItems()`. Here's one possible refactoring of MiscItemFixture.java:

```
package register;
import fit.Fixture;
public class MiscItemFixture extends Fixture {
  private ManualEntry manualEntry;
  public void miscButton(){
    manualEntry = new ManualEntry();
  }
  public void enterButton(){
    manualEntry.createItemWithPrice();
  }
  public void doneButton(){
```

```
    manualEntry.setNumberOfItems();
  }
  public int totalCost(){
    return manualEntry.getTotalCost();
  }
  public String display(){
    return manualEntry.getDisplay();
  }
  public void unitPrice(int unitPrice){
    manualEntry.enterUnitPrice(unitPrice);
  }
  public void numberOfItems(int numberOfItems){
    manualEntry.enterNumberOfItems(numberOfItems);
  }
  public void timesButton(){
    manualEntry.buyMoreThanOne();
  }
}
```

Now the acceptance test fixture is thin. It just delegates all calls into the class being tested. The downside is that these two classes are highly coupled. This means that changes to `ManualEntry` may require changes to `MiscItemFixture`. In this case, we've made changes in `MiscItemFixture` that require changes to `ManualEntry`. You should make the changes by writing unit tests for `ManualEntry`. The tests will check some of the behavior specified by the acceptance tests.

Here's one possible version of ManualEntry.java. Most of the changes have been highlighted.

```
package register;
class ManualEntry {
  private String display;
  private Item currentItem;
  private int numberOfItems;
  private int unitPrice;
  ManualEntry(){
    setDisplay("Enter Unit Price");
  }
  void setDisplay(String display){
    this.display = display;
  }
  String getDisplay(){
    return display;
  }
  void createItemWithPrice(){
    currentItem = new Item(unitPrice);
    setDisplay("Misc Grocery " + unitPrice);
  }
  void buyMoreThanOne(){
    setDisplay("Enter number of items");
```

```
    }
    void setNumberOfItems(){
         currentItem.addToOrder(numberOfItems);
        setDisplay(""+ getTotalCost());
    }
    int getTotalCost(){
      return currentItem.totalItemCost();
    }
    void enterNumberOfItems(int numberOfItems){
      this.numberOfItems = numberOfItems;
      setDisplay(""+ numberOfItems);
    }
    int getNumberEntered(){
      return numberOfItems;
    }
    void enterUnitPrice(int unitPrice){
      this.unitPrice = unitPrice;
      setDisplay(""+ unitPrice);
    }
    int getUnitPrice(){
      return unitPrice;
    }
  }
```

Now the changes to what is displayed are in the production code and not in the test fixture.

RUNNING ALL OF THE TESTS

As your suite of acceptance tests grows you will want a way of running all of them, all of the time. One method is to put them all in a single HTML file. Although allowable, this is an unwieldy solution. You and your customers will need to scroll down through extremely long files looking for particular results. Another solution is to create a test that calls all of the other tests and reports back the results. If you need more details, you can look at the particular tests that returned incorrect results.

The AllFiles Fixture

Several fixtures have been created for running more than one test file. We'll use the `AllFiles` fixture that is included in the `eg` package. Note that this means that `AllFiles` is not included in the fit.jar file. In order to use `AllFiles` you need to add it to your classpath. You can create a jar file eg.jar that contains the `eg` package or you can just add the Classes directory in the Fit distribution to your classpath. Because `AllFiles` is not currently part of the framework, you'll need to check that it is still in the distribution.

Create an HTML file named AllTests.html and place it in the tests/acceptance/test-source directory. This time you'll include a row with the path to any HTML file

containing acceptance tests that you want to run as part of this suite. You are allowed to use wild cards to include all of the files in a certain location or with a certain naming pattern. For example, we can run both the CaseDiscountFitTest.html and the MiscItemFitTest.html by specifying that we are running *FitTest.html. Here's the AllTests.html file.

```
<html>
  <head>
    <title> Acceptance Tests for all User stories</title>
  </head>
  <body>
    <h2> Acceptance Tests for all User stories:  </h2>
    <table BORDER>
      <tr>
        <td colspan = 2>
          eg.AllFiles
        </td>
      </tr>
      <tr>
        <td colspan = 2>
          tests/acceptance/testsource/*FitTest.html
        </td>
      </tr>
    </table>
    <h2> Summary of tests run on this page </h2>
    <table BORDER>
      <tr>
        <td> fit.Summary </td>
      </tr>
    </table>
  </body>
<html>
```

Notice that we've also included a call to `fit.Summary` in a second table. This provides a more informative summary of the tests run by the call to `AllFiles`.

Results of Running the AllFiles Fixture

You need to know what you do and don't get from running this fixture. You do get a comprehensive report as shown in Figure 14-8.

The top table in Figure 14-8 lists the results for each of the files matched by the expressions you input. One advantage of running this test suite is that while you have your heads down and are focused on the tests in MiscItemFitTest.html you can make sure you aren't breaking anything anywhere else.

The bottom table lists the total files run in the counts variable and the summary results for tests in the counts run variable. The number of tests run should continue to climb. It's a matter of taste, but the number ignored and the number that appears in the exceptions should never be nonzero for very long. The ratio of right to wrong should increase. You can't insist that acceptance tests run at 100% all of the time. With unit

Acceptance Tests for all User stories:

eg.AllFiles	
tests/acceptance/testsource/*FitTest.html	
CaseDiscountFitTest.html	8 right, 0 wrong, 0 ignored, 0 exceptions
MiscItemFitTest.html	7 right, 0 wrong, 0 ignored, 0 exceptions

Summary of tests run on this page

fit.Summary	
counts	2 right, 0 wrong, 0 ignored, 0 exceptions
counts run	15 right, 0 wrong, 0 ignored, 0 exceptions
input file	/Users/daniel/Dev/dozen/tests/acceptance/testsource/AllFiles.html
input update	Thu Nov 28 07:15:42 EST 2002
output file	/Users/daniel/Dev/dozen/tests/acceptance/testresults/AllFiles.html
run date	Thu Nov 28 07:15:59 EST 2002
run elapsed time	0:00.25

Figure 14-8 Acceptance Tests for All User Stories

tests, everything stops until you fix a broken test. But you also don't write code until you write a test that breaks. Acceptance tests communicate customer requirements. It is not unexpected that when a customer contributes a new set of requirements that many of the acceptance tests won't work.

Running this fixture does not generate new HTML files for each of the files for which tests are run. You can see the summary results here. If you then decide to look at the latest information in the tests/acceptance/testresults/MiscItemFitTest.html you'll notice that this is still the same version. If you want details for failing tests, you need to process that individual page. The comprehensive tests point you in the direction of files that need to be looked at further.

Next Steps

Now create your own tests and add them to the AllTests.html file. In this tutorial we've explored examples of column fixtures and action fixtures. You can now create a row fixture and see how it is different from the other two. Work with your customer to help create tables that capture their requirements. Automate the testing process in a way that allows you to provide feedback to your customer, your instructor, and the other developers on your team.

Finally, it may feel as if you have devoted a lot more effort to testing than to writing the application code. It is true that between unit tests and acceptance tests you may actually end up with more lines of code used in testing than in your application. Your application will tend to be more directed, lean, and modular. The test code will benefit the application you are building. In addition, you should know that in an actual commercial development environment your application would still require more testing, which is often created by an independent team.