# Testing your Rules

We went through many samples in the first six chapters of the book, and didn't have to pay out 'real' money if we made a mistake. We could happily play with the examples, make mistakes, and learn from them. This changes when we start writing 'real' business rules. How much money would a company lose if a rule that you wrote gave double the intended discount to a customer? Or, what if your airline ticket pricing rule started giving away first class transatlantic flights for one cent?

Of course, mistakes happen. This chapter makes sure that these costly mistakes don't happen to you. If you're going through the trouble of writing business rules, you will want to make sure that they do what you intend them to, and keep on doing what you intend, even when you or other people make changes, both now and in the future.

The chapter shows how to test your rules. It begins the testing by using Guvnor.

It then shows how to test rules against requirement documents using the **FIT**

(**Framework** for **Integrated** **Testing**), and then shows how to unit test rules by using Junit. But first of all, we will see how testing is not a standalone activity, but part of an ongoing cycle.

## Testing when building rules

In this book we play with the examples, and then throw them away when we've learned everything that we can from them. Real life isn't like that. It's a slightly morbid thought, but there's every chance that some of the business rules that you write will last longer than you do. Remember the millennium bug caused by programmers in the 1960's, assuming that nobody would be using their work in

40 years' time, and then being surprised when the year 2000 actually came along?

Rather than 'play and throw away', we're more likely to create production business rules in the following cycle:

1. Write your rules (or modify an existing one) based on a specification, or feedback from end users.
2. Test your rules to make sure that your new rules do what you want them to do, and ensure that you haven't inadvertently broken any existing rules.
3. Deploy your rules to somewhere other than your local machine, where end users (perhaps via a web page or an enterprise system) can interact with them.

You can repeat steps 1, 2, and 3 as often as required. That means, repeat as many times as it takes you to get the first version into production. Or, deploy now and modify it anytime later –in 1, 2, or 10 years time.

We covered step 1 (writing) in the first six chapters of this book. We will cover step 3 (deployment) in Chapter 11. Testing is what we'll cover in this chapter.

### Making testing interesting

Normal testing, where you inspect everything manually, is booooooooring! You might check everything the first time, but after the hundredth deployment you'll be tempted to skip your tests—and you'll probably get away with it without any problems. You'll then be tempted to skip your tests on the 101st deployment—still no problems. So, not testing becomes a bad habit either because you're bored, or because your boss fails to see the value of the tests.

The problem, then comes one Friday afternoon, or just when you're about to go on vacation, or some other worst possible time. The whole world will see any mistakes in the rules that are in production. Therefore, fixing them is a lot more time and money consuming than if you catch the error at the very start on your own PC.

What's the solution? **Automate** the testing. All of your manual checks are very repetitive—exactly the sort of thing that computers are good at. The sort of checks for our chocolate shipment example would be 'every time we have an order of 2000 candy bars, we should have 10 shipments of 210 bars and one shipment of 110 bars'.

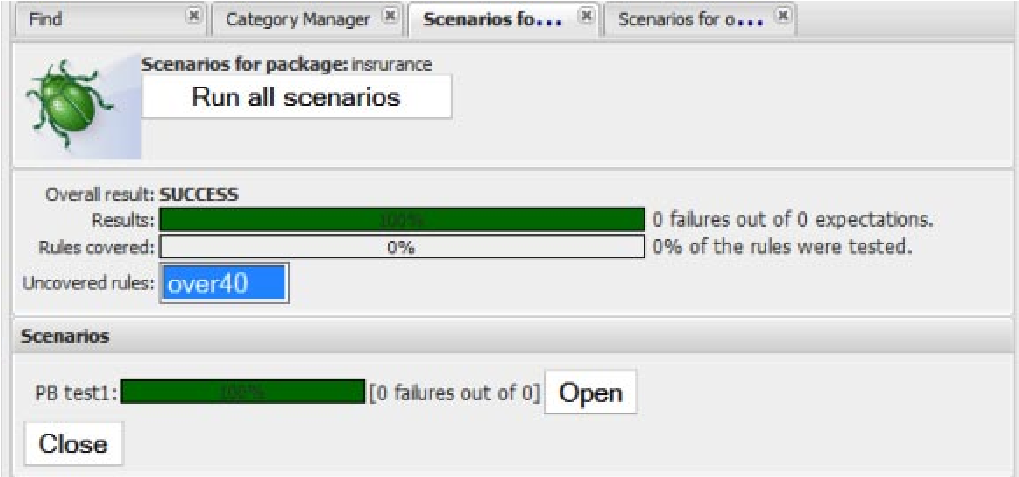
### Testing using Guvnor

Back in Chapters 3 and 4, we used the testing facilities in Guvnor to run our first rules. We had no other way of running rules that would eventually be deployed into the enterprise systems. This is another advantage of testing—we can instantly see whether our tests are correct, without having to wait for our rules to be deployed into the target system.

At a high level, Guvnor has two main screens that deal with testing:

* An individual test screen: Here you can edit your test by specifying the values that you want to input, and the values that you expect once your rules have fired
* A package or multiple tests screen (below): This allows you to run (later on) all of the tests in your package, to catch any rules that you may have inadvertently broken

Another way of saying this is: You write your tests for selfish reasons because you need them to ensure that your rules do what you want them to do. By keeping your tests for later, they automatically become a free safety net that catches bugs as soon as you make a change.



### Testing using FIT

Guvnor testing is great. But, often, a lot of what you are testing for is already specified in the requirements documents for the system. With a bit of thought in specifying various scenarios in your requirements documents, **FIT** allows you to automatically compare these requirements against your rules. These requirements documents can be written in Microsoft Word, or similar format, and they will highlight if the outputs aren't what is specified. Like Drools, FIT is an open source project, so there is no charge for either using it, or for customising it to fit your needs.

Before you get too excited about this, your requirements documents do have some compromises. The tests must specify the values to be input to the rules, and the expected result—similar to the examples, or scenarios, that many specifications already contain. These scenarios have to follow a FIT-specific format. Specification documents should follow a standard format anyway—the FIT scenario piece is often less than 10% of it, and it is still highly human-readable! Even better, the document can be written in anything that generates HTML, which includes Microsoft Word, Excel, OpenOffice, Google documents, and most of the myriad of editors that are available today.

Like the Guvnor testing, we can use FIT to test whether our individual requirements are being met when writing our rules. It is possible to run FIT automatically over multiple requirement documents to ensure that nothing has 'accidentally' broken as we update other rules.

#### Getting FIT

When you downloaded the samples for Chapter 6, you probably noticed three strange packages and folders that we didn't talk about at the time.

* **fit-testcase**: This folder resides just within the main project folder, and contains the FIT requirements documents that we're going to test against.
* **chap7**: This is a folder under **src/main/java/net/firstpartners**, and contains the **startpoint** (**FitRulesExample.java**) that we'll use to kickstart our FIT Tests.
* **FIT**: This folder is next to the **chap7** folder. It contains some of the 'magic plumbing' that makes FIT work. Most business users won't care how this works (you probably won't need to change what you find here), but we will take a look at it in more detail in case we want to customize exactly how FIT works.

If you built the previous example using Maven, then all of the required FIT software will have been downloaded for you. (Isn't Maven great?) So, we're ready to go.

#### The FIT requirements document

Open the word document **fit-testcase.doc** using Word, or OpenOffice. Remember that it's in the **fit-testcase** folder. **fit-testcase.doc** is a normal document, without any hidden code. The testing magic lies in the way the document is laid out. More specifically, it's in the tables that you see in the document. All of the other text is optional. Let's go through it.

##### Logo and the first paragraph

At the very top of the document is the Drools logo and a reference to where you can download FIT for rules code. It's also worth reading the text here, as it's another explanation of what the FIT project does and how it works.



None of this text matters, or rather FIT ignores it as it contains no tables. We can safely replace this (or any other text in this document that isn't in the table) with your company logo, or whatever you normally put at the top of your requirement documents.

FIT is a **GPL** (**General** **Public** **License**) open source software. This means you can modify it (as long as you publish your changes). In this sample we've modified it to accept global variables passed into the rules. (We will use this feature in step 3.)

The changes are published in the FIT plumbing directory, which is a part of the sample. Feel free to use it in your own projects.

##### First step—setup

The setup table prepares the ground for our test, and explains the objects that we want to use in our test. These objects are familiar as they are the Java facts that we've used in our rules.

There's a bit of text (worth reading as it also explains what the table does), but FIT ignores it. The bit that it reads is given in the following table:

|  |  |
| --- | --- |
| **net.firstpartners.fit.fixture.Setup** |  |
| **net.firstpartners.chap6.domain. CustomerOrder** | **AcmeOrder** |
| **net.firstpartners.chap6.domain. OoompaLoompaDate** | **nextAvailableShipmentDate** |

If you're wondering what this does, try the following explanation in the same table format:

|  |  |
| --- | --- |
| **Use the piece of plumbing called 'Setup'** |  |
| Create **CustomerOrder** and call it | **AcmeOrder** |
| Create **OoompaLoompaDate** and call it | **nextAvailableShipmentDate** |

There is nothing here that we haven't seen before. Note that we will be passing **nextShipmentDate** as a global so that it matches the global of a same name in our rules file (the match includes the exact spelling, and the same lower-and uppercase).

##### Second step—values in

The second part also has the usual text explanation (ignored by FIT) and table (the important bit), which explains how to set the values.

|  |  |
| --- | --- |
| **net.firstpartners.fit.fixture.Populate** |  |
| **AcmeOrder** Set initial balance | 2000 |
| **AcmeOrder** Set current balance | 2000 |

It's a little bit clearer than the first table, but we'll explain it again anyway.

|  |  |  |
| --- | --- | --- |
| **Use the piece of plumbing called Populate** | |  |
| **AcmeOrder** | Take the … we created earlier, and set it to have an initial balance of … | 2000 |
| **AcmeOrder** | Take the … we created earlier, and set it to have a current balance of … | 2000 |

##### Third step—click on the Go button

Our next part starts the rules. Or rather, the table tells FIT to invoke the rules. The rest of the text (which is useful to explain what is going on to us humans) gets ignored.

**net.firstpartners.fit.fixture.Engine**

Ruleset **src/main/java/net/firstpartners/chap6/ shipping-rules.drl**

Assert **AcmeOrder**

Global **nextAvailableShipmentDate**

Execute

The following table is the same again, in English:

**Use the piece of plumbing called 'Engine'**

Ruleset Use the rules in **shipping-rules.drl**

Assert Pass our **AcmeOrder** to the rule engine (as a fact)

Global Pass our **nextAvailableShipmentDate** to the rule engine (as a global)

Execute Click on the **Go** button

##### Fourth step—check the results

After running our rules, we check to see if the results are as we expected.

|  |  |
| --- | --- |
| **net.firstpartners.fit.fixture.Results** |  |
| **AcmeOrder** Get current balance | 0 |
| **AcmeOrder** Get initial balance | 2000 |

The following table explains whether the results are as we expected:

|  |  |  |
| --- | --- | --- |
| **Use the piece of plumbing called 'Results'** | |  |
| **AcmeOrder** | Check the … we created earlier, and make sure we now have a current balance of … | 0 |
| **AcmeOrder** | Check the … we created earlier, and make sure we have an initial balance of … | 2000 |

##### Clear (an optional step)

This optional step clears any of the previous steps. This means that, if we want, we can repeat steps 1 to 4 again in the same requirements document. **net.firstpartners.fit.fixture Clear**

##### Print a summary (an optional step)

Another optional step is to print a summary of how our tests went. This is in addition to the information that will append to each of the previous steps. **fit.Summary**

##### Footer

Finally, there is some more narrative in the footer. FIT ignores this, so you can remove or replace this if you want.

There is an important part in this footer. It's a note that the sample (that we're now using) is based on FIT, and also on a sample by Michael Neale from the Drools team—**http://fit-for-rules. sourceforge.net/**. This sample is based on GPL licenced code. My understanding (remember that I'm not a lawyer) is that you can use it internally within your organization; but if it is used outside, then you will need to publish any changes (if you made any) to the core FIT testing code.

#### Running FIT on our sample

So far, all we've seen is a Word document. Let's make the magic happen by running

FIT against this document. In this example we're running FIT through the JBoss IDE, but it would be easy for anyone with a moderate knowledge of Java (ask your friendly technical person) to make this work by double-clicking on a Windows icon. The steps for running FIT on our sample are given as follows:

1. FIT doesn't run against the Word document, but against the HTML version of the Word document. Save your Word document as HTML. (In Word, choose **File | Save As | Web Page**.) Name the file as **fit-testcase.htm**.

Our project already has this 'Save As HTML' step done, but don't forget to do it again if you make any changes to the FIT document, later.

1. Run the FIT test. In this case select **FitRulesExample** in the JBoss IDE (package explorer or navigator), and right-click on it. Select **Run As | Java Application** from the context menu that is dispalyed.
2. You should see a bunch of stuff in the console, beginning with **net firstpartners.chap7.FitRulesExample main** and ending with **Clearing domain objects**.
3. In between, you'll see the results of our rules firing, such as **28 right, 0 wrong, 23 ignored, 0 exceptions.** Remember that we automate our testing in order to automate our inspection of the results. So don't bother reading the console, other than to check that FIT has run successfully.
4. To see the results, open **fit-test-result.htm**. The filenames that we use (**fit-testcase.htm** and **fit-test-result.htm**) are set in the

**FitRulesExample.java** file, and can be changed to whatever you want.

**What just happened?**

**Fit-test-result.htm** is a copy of the original FIT test case, but it has been updated with the results of our tests. If everything goes well, all of our tables should be highlighted. As an example, the fourth table where we check the results is shown as follows:

|  |  |
| --- | --- |
| **net.firstpartners.fit.fixture.Results** |  |
| AcmeOrder Get current balance | 0 |
| AcmeOrder Get initial balance | 2000 |

The other main change has been our (optional) summary table. FIT has updated the output with a summary of the set of tests that it has just run. Most importantly, it's highlighted.

|  |  |
| --- | --- |
| **fit.Summary** |  |
| counts  input file | 28 right, 0 wrong, 23 ignored, 0 exceptions |
| C:\projects-drools\chap6-sample\fit-testcases\fit-testcase.htm |
| input update | Thu Sep 04 22:31:39 BST 2008 |
| output file | C:\projects-drools\chap6-sample\fit-testcases\fit-test-result.htm |
| run date | Thu Sep 04 22:43:44 BST 2008 |
| run elapsed time | 0:01.58 |

There is a small bug in the version of FIT used here (we didn't have 28 tests). But a test that fails will always be picked up, which leads us to the question, "What does a failed test look like?"

**What can go wrong?**

Let's imagine that our requirements document was slightly different—it said that our rules have run, and so we should still have a balance left on our order of 150 bars still to ship. In other words, table 4 of our requirements document should look something like this:

|  |  |
| --- | --- |
| **net.firstpartners.fit.fixture.Results** |  |
| AcmeOrder Get current balance | 150 |
| AcmeOrder Get initial balance | 2000 |

When we save the document using **Save as | HTML** and run our FIT tests again (using the steps above), we'll notice two differences. The first indication that a test has failed is in the console of the JBoss IDE, where we'll see the message: **25 right, 3 wrong, 23 ignored, 0 exceptions**. The second change is in the FIT document, where the results table (table 4) is highlighted in dark gray. Helpfully, it shows you the actual and expected values. The following figure shows you a failing FIT test:

|  |  |
| --- | --- |
| Net.firstpartners.fit.fixture.Results |  |
| AcmeOrder Get current balance | 150 expected  0 actual |
| AcmeOrder Get initial balance | 2000 |

Normally, we would revise our rules and run our FIT tests again until our requirements were met (hightlighted).

That was a failing rule. But what if something more serious happens that causes our rules to 'blow up', or throw an exception? We made a note while talking about the rules that it was important that the rule matches the name of our global variable.

What happens if it doesn't? Let's change the name of the variable in tables 1 and 3. The following table shows the setup using some made-up name:

|  |  |
| --- | --- |
| **net.firstpartners.fit.fixture.Setup** |  |
| net.firstpartners.chap6.domain.CustomerOrder | AcmeOrder |
| net.firstpartners.chap6.domain.OoompaLoompaDate | **someMadeupName** |

The following table shows how to execute the rule using some made-up name:

**net.firstpartners.fit.fixture.Engine**

Ruleset **src/main/java/net/firstpartners/chap6/shipping-rules.drl**

Assert **AcmeOrder**

Global **someMadeupName**

**Execute**

When we save the document using **Save as | HTML** and we run FIT again, we see an indication of problems in the console (**22 right, 3 wrong, 23 ignored, 2 exceptions**) with full details of the error in the updated FIT results document. It is shown as follows with some details removed, and others highlighted for clarity:

**org.drools.spi.ConsequenceException: java.lang. NullPointerException ….**

**Caused by: java.lang.NullPointerException**

**at net.firstpartners.chap6.Rule\_Add\_Next\_Available\_Shipment\_Date\_ 0.consequence(Rule\_Add\_Next\_Available\_Shipment\_Date\_0.java:10)**

**... 19 more**

When you first see this, there is too much detail (about 19 lines of technical-looking text, in this example), but the important bits are:

* The first line that tells us we have a **nullpointer** exception. As explained in the previous chapter, this is where a rule expects to have something available; but what it expects isn't there.
* The **Caused** **by** line indicates that the problem is in **Rule\_Add\_Next\_ Available\_Shipment**, and more particularly in the consequence (the 'when' part).

Looking at the When part of **Rule\_Add\_Next\_Available\_Shipment**, we see that it contains the code **nextAvailableShipmentDate.rollForward(7)**. This assumes that our **nextAvailableShipmentDate** has something passed in to it. But it doesn't, as we've just changed the FIT rules to use a different name. Hence, it is empty, and causes the **NullPointerException** that we see.

This process of deduction would be even more impressive if we hadn't deliberately broken the rule in the first place.

Even though we change the first (and not the third) table in our FIT document, we will still get an exception, but of a different kind: **IllegalArgumentException-No** **domain** **object** **for** **key** **'nextAvailableShipmentDate'** **exists**.

As before, the key to resolving the error is to not get scared by the large amount of detail provided, but to look for clues. In this case something is wrong relating to **nextAvailableShipmentDate**. 80% of the problems are caused by something being misspelt, or the wrong variable name being used (which is the problem in this case).

### The FIT plumbing

If you're interested only in testing against requirements, then you should presume that FIT for rules 'just works' and skip ahead to the next section (on unit testing). If you're somewhat technical, you're probably wondering how FIT works under the covers. If so, read on.

In general, the classes that were mentioned in the FIT template (for example, **net.firstpartners.fit.fixture.Setup**) are the plumbing that allows the FIT framework to understand our rules code. These files can be found in the **src/main/java/net/firstpartners/fit** folder.

FIT or FIT for rules? FIT is a general-purpose requirements-testing framework that can be adapted to test almost anything. Normally, 'adapted' means a bit of work to match FIT to your code. Fortunately, as most rules follow a standard pattern, Micheal Neale (of the Drools team) has already written the adapter for you as the 'FIT for Rules' project.

The most likely scenario is that 99% of what you need is already covered by this (modified) FIT for Rules framework, but you may come across something that you would like to add, or do differently. For example, the modified framework can pass global variables to the rules, whereas the original source cannot.

The full source is available for both, so these are just pointers to get you started:

1. When we start **FitRulesExample**, Java finds our **main()** method and jumps in. This method sets up parameters (such as input and output files) and calls the **FileRunner** file (**run** method).
2. **FileRunner** is a part of the FIT framework—the source code is available. But as a summary, it loads the input document that we specified and scans it for tables that may contain instructions to carry out a test.
3. You'll notice that the first line of all of the tables in our FIT document start with an instruction—Setup, Populate, Engine, Results, Clear, and Summary. When the FIT framework finds one of these, it tries to find the file, and loads it.
4. All of these instruction files follow a similar format. They 'extend' another file called **AbstractRulesTesting**, which is like a template. So when we are reading a file (for example, **Setup.java**), we must remember that it also contains all of the code from **AbstractRulesTesting**. (Because many files use this common template, it saves us a lot of typing.)
5. At this point, FIT has scanned the document, found the table, and loaded the instruction file (for example, **Setup.java**). It loops through each cell in the table and calls the **doCell()** method in the **Setup.java** (or other instruction) file.
6. The **doCell()** method checks to see what sort of cell it is. It may carry out different actions depending on whether it's the first row, the first column, or any other cell.

In this way, FIT loops through the document and sets up the facts to be tested, calls the rule engine, checks the results, and then prints a summary.

Remember that we would normally write the instruction file to understand our code. It just happens that we have a set of adaptors (in FIT for Rules) that understands most of the rules-related code. If we need to modify an adaptor, we can add pretty much any Java-based code that's required.

The actual mechanics of calling the rules from FIT are the same as our previous examples. What is different in FIT for Rules is that we express our tests in Word, and (via our adaptors) FIT understands what values to pass to the rules.

Now that we understand requirements testing, let's look at testing at the next level down—unit testing.

**What is unit testing?**

A good enterprise computer system should be built as if it was made of Lego bricks. Your rules are only a piece of the puzzle. You'll need to go back to the Lego box to get pieces that talk to the database, make web pages, talk to other systems that you may have in your company (or organisation), and so on. Just as Lego bricks can be taken apart and put together in many different ways, the components in a welldesigned system should be reusable in many different systems.

Before you use any of these components (or 'bricks') in your system, you will want to be sure that they work. For Lego bricks this is easy—you can just make sure that none of the studs are broken. For components this is a bit harder—often, you can neither see them, nor do you have any idea whether their inputs and outputs are correct. Unit testing makes sure that all of the component pieces of your application work, before you even assemble them.

You can unit test manually, but just like FIT requirements testing, you're going to 'forget' to do it sooner or later. Fortunately, there is a tool to automate your unit tests known as **Junit** (for Java; there are also versions for many other languages, such as .Net). Like Drools and FIT, Junit is open source. Therefore, we can use it on our project without much difficulty. Junit is integrated into the JBoss IDE and is also pretty much an industry standard, so it's easy to find more information on it. A good starting point is the project's home page at **www.Junit.org**.

The following points can help you to decide when to use unit testing, and when to use the other forms of testing that we talked about:

* If you're most comfortable using Guvnor, then use the test scenarios within Guvnor. As you'll see shortly, they're very close to unit tests.
* If the majority of your work involves detailing and signing off against the requirement documents, then you should consider using FIT for Rules.
* If you're most comfortable using Java, or some other programming language, then you're probably using (J)unit tests already—and we can apply these unit tests to rule testing.

In reality, your testing is likely to be a mix of two or three of these options.

**Why unit test?**

An important point to note is that you've already carried out unit testing in the rules that we wrote earlier. OK, it was manual unit testing, but we still checked that our block of rules produced the outcome that we expected. All we're talking about here is automating the process.

Unit testing also has the advantage of documenting the code because it gives a working example of how to call the rules. It also makes your rules and code more reusable. You've just proved (in your unit test) that you can call your code on a standalone basis, which is an important first step for somebody else to be able to use it again in the future.

You do want your rules to be reused, don't you?

#### Unit testing the Chocolate Shipments sample

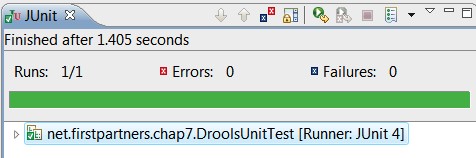
As luck would have it, our Chocolate Shipments example also contains a unit test. This is called **DroolsUnitTest.java**, and it can be found in the **test/java/net/ firstpartners/chap7** folder.

**DroolsUnitTest.java** lives in a similar, but parallel, set of folders. It begins with **test** instead of **main**. Having these parallel sets of folders is a convention used to separate our test code from our production code.

We want to run our tests before we deploy them to production, but not actually deploy those tests. By having the folders set up in this way, we fool Java into thinking that these tests live in the same package as the actual rules, yet they are still separate. This is done so that the tests don't accidentally get deployed (which is a good thing, as we don't want our tests to corrupt our production data).

Running the Junit test is similar to running the samples. In the JBoss IDE Navigator or package explorer, we select **DroolsUnitTest.java**, right-click on it, and then select **Run as | Junit test** from the shortcut menu.

All being well, you should see some messages appear on the console. We're going to ignore the console messages; after all, we're meant to be automating our testing, not manually reading the console. The really interesting bit should appear in the IDE— the Junit test result, similar to the screenshot shown below. If everything is OK, we should see the green bar displayed—success!



We've run only one unit test, so the output is fairly simple. From top to bottom we have: the time it took to run the test; the number of errors and failures (both zero—we'll explain the difference shortly, but having none of them is a good thing), the green bar (success!), and a summary of the unit tests that we've just run (**DroolsUnitTest**).

If you were running this test prior to deploying to production, all you need to know is that the green bar means that everything is working as intended. It's a lot easier than inspecting the code line by line.

However, as this is the first time that we're using a unit test, we're going

to step through the tests line by line. A lot of our Junit test is similar to

**MultipleRulesExample.java** that we ran in the previous chapter. For example, the unit test uses the same **RuleRunner** file to load and call the rules. In addition, the Junit test also has some automated checks (asserts) that give us the green bar when they pass, which we saw in the previous screenshot.

**What just happened?**

Probably the easiest way to understand what just happened is to walk through the contents of the **DroolsUnitTest.java** file.

Our unit code starts with the usual package information. Even though it is in a separate folder, Java is fooled into using the same package.

**package net.firstpartners.chap7;**

In our imports section (list of other files that we need), we have a mix of our domain objects (the facts such as **CustomerOrder**) that we used earlier for holding information. We also have the logging tools. What is new is the imports of **Assert** (part of our automatic checking tool) and importing the **junit** test (the template for our unit test).

**import static org.junit.Assert.assertEquals; import static org.junit.Assert.assertNotNull; import static org.junit.Assert.assertTrue; import java.util.HashMap; import net.firstpartners.chap6.domain.CustomerOrder; import net.firstpartners.chap6.domain.OoompaLoompaDate; import net.firstpartners.drools.RuleRunner; import org.apache.commons.logging.Log; import org.apache.commons.logging.LogFactory; import org.junit.Test;**

The start of the main part of the file may be renamed to **DroolsUnitTest**, but what it does is the same. The rules are still read from exactly the same file as before.

**public class DroolsUnitTest { private static Log log = LogFactory.getLog(DroolsUnitTest.class); private static final String NEXT\_AVAILABLE\_SHIPMENT\_DATE = "nextAvailableShipmentDate"; private static final String[] RULES\_FILES = new String[] { "src/ main/java/net/firstpartners/chap6/shipping-rules.drl" };**

Earlier, our starting point was called **main** so that Java knew where we wanted it to start when we pressed the green **Go** button. This time, our start method is called **testShippingRules** and it's marked with a **@Test** flag so that we know it's an entry point. We can have multiple tests, each marked with **@Test**. The Junit framework will test each one in turn.

The rest of this code snippet, which involves setting up and calling the business rules via **RuleRunner**, is exactly the same as our previous 'calling the rule engine' samples.

**@Test**

**public void testShippingRules() throws Exception {**

**// Initial order**

**CustomerOrder candyBarOrder = new CustomerOrder(2000);**

**HashMap<String, Object> startDate = new HashMap<String,**

**Object>();**

**startDate.put(NEXT\_AVAILABLE\_SHIPMENT\_DATE, new**

**OoompaLoompaDate(2009, 02, 03));**

**// Holidays**

**OoompaLoompaDate holiday2 = new OoompaLoompaDate(2009, 2, 10);**

**OoompaLoompaDate holiday1 = new OoompaLoompaDate(2009, 3, 17);**

**// Call the rule engine**

**Object[] facts = new Object[3]; facts[0] = candyBarOrder; facts[1] = holiday1; facts[2] = holiday2;**

**// A lot of the running rules uses the same code. The RuleRunner**

**(code**

**// in this project)**

**// keeps this code in one place. It needs to know**

**// - the name(s) of the files containing our rules**

**// - the fact object(s) containing the information to be passed in and**

**// out of our rules**

**// - a list of global values**

**new RuleRunner().runRules(RULES\_FILES, facts, startDate);**

In our previous example, once we called the rules, we printed the results out to the screen for manual inspection. This time things are different. We want to make this checking automatic. Hence, we have added following new lines in the final snippet, using **assertXXX** to check if the values that we get back from the rules are as expected:

**// Check that the results are as we expected assertEquals(**

**"No more bars should be left to ship", 0, candyBarOrder**

**.getCurrentBalance()); assertEquals(**

**"Our initial order balance should not be changed", 2100, candyBarOrder.getInitialBalance()); assertNotNull(**

**"Our list of shipments should contain a value", candyBarOrder.getShipments()); assertTrue(**

**"We should have some Cusomter Shipments", candyBarOrder.getShipments().size() > 5); }**

**}**

In general, our **assert** checks follow the format: **Assert(** **"message** **if** **the** **value** **is not** **as** **we** **expect"** **,** **valueWeExpected,** **valueWeGotWhenWeRanTheTest)**

* The first line (**assertEquals**) compares the number of candy bars that should still be left to ship after our rules have fired (should be 0)
* The second line (**assertEquals**) ensures that the initial order is not changed by the rules, and remains at 2100
* The next line (**assertNotNull**) ensures that the list of shipments that we made is not empty
* The final line (**assertTrue**) checks that we have more than five shipments made to a customer

Is it best to have multiple tests or multiple asserts within a single test? It is possible to have multiple tests, such as **someTest()** methods (each marked with **@Test**), and/or multiple tests using **assertXXX** within a method. A combination of both is probably the best. Multiple asserts in one method are great when your test is difficult to set up, but the test will stop at the first assert that turns out to be false. This means you can solve the first error, but your test will then stop at the next assert that fails. Having these asserts in separate test methods shows you instantly how many problem(s) you have—at the price of having some duplicated setup code.

There is a school of thought that advocates test-first design. Write your unit tests before you write any proper rules—this acts as your specification.

Of course, all of your tests will fail at the start. However, bit by bit, you must write the rules to make them work. That way, you know when you've done what you started out to do—no more and no less. And it means that you can never 'forget' to write your tests.

It may appear strange, but it works for many people. So, it's certainly worth giving a try.

**What if it goes wrong?**

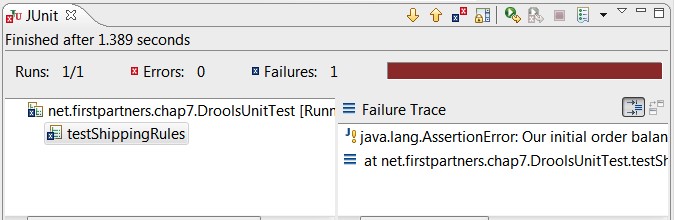
We were lucky that our tests worked the very first time. Unfortunately, this is almost impossible to achieve. For example, assume that we mistakenly wrote a rule that changed the initial balance.

**assertEquals(**

**"Our initial order balance should not be changed",**

**2100, candyBarOrder.getInitialBalance());**

In this case, when we come to check, our test will fail. We will get a red bar in our unit tests, detailing what has gone wrong (similar to the screenshot below). The message in our assert (**Our initial order balance should not be changed**) and other details (such as line numbers) are provided to help us trace what is going wrong. You'll also notice that the **Failures** count is now **1**.



#### Failures and errors

So what's the difference between failures and errors? Failures are things (such as the above assert) that we explicitly check for. Errors are the unexpected things that go wrong. Remember our **NullPointerException** from the previous section in FIT? That is, the problem that we face when something is empty that shouldn't be. That exception is shown as an error in Junit with a red bar (again), along with the details of the problem to help you fix it.

It's simple—green is good and red is bad. But remember, it's always better to catch mistakes early.

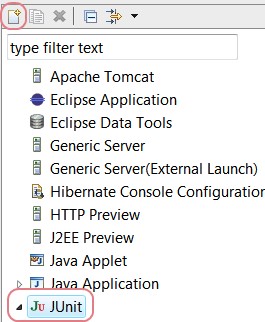
#### Testing an entire package

Typically, you write a unit test at the same time as writing a set of rules to confirm that the functionality is 'done'.

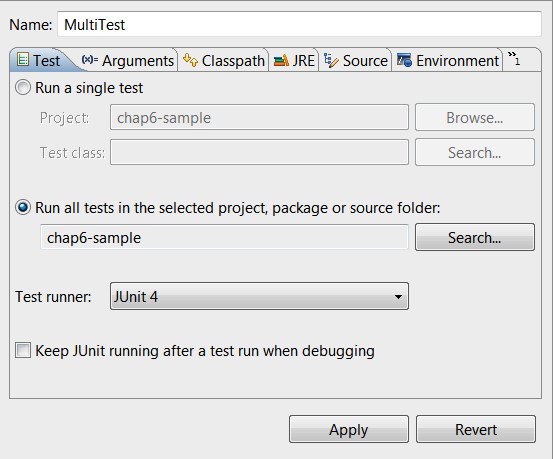
After you're 'done', you (or one of your team) should run all of the tests in the project to make sure that your new work hasn't broken any of the rules or code that already existed. There are a couple of ways to automatically do this overnight (as part of a build control tool such as Cruise Control) as part of your build scripts (Maven, the tool installed to build the samples in this book does this for you), or you can run all of the tests from the JBoss IDE (akin to the 'run all scenarios in package' that we saw in Guvnor).

It's pretty easy to run all of your unit tests in one go. All you have to do is this:

1. From the toolbar (at the very top of the JBoss IDE) select **Run**.
2. In the dialog box that is dispalyed, select **Junit** (near the lower-left of the screen) and then click on **New launch configuration** (the icon on the upper-left of the screen, as shown in the screenshot).



1. On the righthand side, fill in the following values:



1. Click on **Apply** (to save it, so that you don't have to go through all of the steps the next time), and then click on **Run**.

As before, the JBoss IDE will chug away for a couple of moments, and then the popup Junit screen will be displayed. As before, if any of the multiple tests fail, you'll see a red bar, along with details of all of the items that went wrong. When all of your tests pass, you can be sure that your rules are of top quality.

## Summary

Hopefully, you're now confident about testing. In this chapter we've seen how to test our rules using Guvnor, as well as using FIT for rule testing against requirements documents, and unit testing using Junit. Now that we can write and test advanced enterprise rules, let's see another rule format—rules in Excel (decision table) format.