Concept

Negative testing is the testing of cases that are expected to fail.

Exercise 9.11 Which of the test cases mentioned in the previous exercises are positive tests and which are negative? Make a table of each category. Can you think of further positive tests? Can you think of further negative ones?

9.4 Test automation

Concept

test automation simplifies the process of regression testing.

One reason why thorough testing is often neglected is that it is both a time-consuming and a relatively boring activity, if done manually. You will have noticed, if you did all exercises in the previous section, that testing thoroughly can become tedious very quickly. This particularly becomes an issue when tests have to be run not just once but possibly hundreds or thousands of times. Fortunately, there are techniques available that allow us to automate repetitive testing, and so remove much of the drudgery associated with it. The next section looks at test automation in the context of *regression testing*.

9.4.1 Regression testing

It would be nice if we could assume that correcting errors only ever improves the quality of a program. Sadly, experience shows that it is all too easy to introduce further errors when modifying software. Thus, fixing an error at one spot may introduce another error at the same time.

As a consequence, it is desirable to run *regression tests* whenever a change is made to software. Regression testing involves rerunning tests that have previously passed, to ensure that the new version still passes them. It is much more likely to be performed if it can be automated. One of the easiest ways to automate regression tests is to write a program that acts as a *test rig*, or *test harness*.

9.4.2 Automated testing using JUnit

Support for regression testing is integrated in BlueJ (and many other development environments) using a testing system called JUnit. JUnit is a testing framework devised by Erich Gamma and Kent Beck for the Java language, and similar systems are now available for many other programming languages.

JUnit JUnit is a popular testing framework to support organized unit testing and regression testing in Java. It is available independent of specific development environments, as well as integrated in many environments. JUnit was developed by Erich Gamma and Kent Beck. You can find the software and a lot of information about it at `http://www.junit.org`.

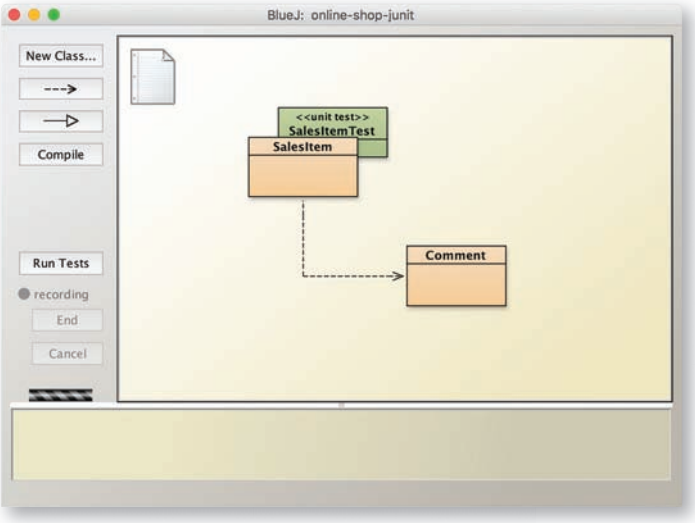


Figure 9.2
A project with a test class

To start investigating regression testing with our example, open the *online-shop-junit* project. This project contains the same classes as the previous one, plus an additional class, **`SalesItemTest`**.

`SalesItemTest` is a test class. The first thing to note is that its appearance is different from what we have seen previously (Figure 9.2). It is annotated as a **`<<unit test>>`**, its color is different from that of the ordinary classes in the diagram, and it is attached to the **`SalesItem`** class (it will be moved with this class if **`SalesItem`** is moved in the diagram).

You will note that Figure 9.2 also shows some additional controls in the main window, below the *Compile* button. These allow you to use the built-in regression testing tools. If you have not used JUnit in BlueJ before, the testing tools are switched off, and the buttons will not yet be visible on your system. You should switch on these testing tools now. To do this, open the *Interface* tab in your *Preferences* dialog, and ensure that the *Show unit testing tools* option is selected.

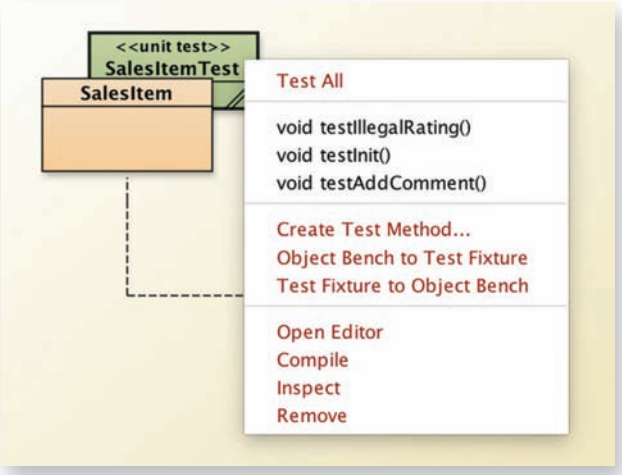
A further difference is apparent in the menu that appears when we right-click the test class (Figure 9.3). There are three new sections in the menu instead of a list of constructors.

Using test classes, we can automate regression testing. The test class contains code to perform a number of prepared tests and check their results. This makes repeating the same tests much easier.

A test class is usually associated with an ordinary project class. In this case, **`SalesItemTest`** is associated with the **`SalesItem`** class, and we say that **`SalesItem`** is the *reference class* for **`SalesItemTest`**.

In our project, the test class has already been created, and it already contains some tests. We can now execute these tests by clicking the *Run Tests* button.

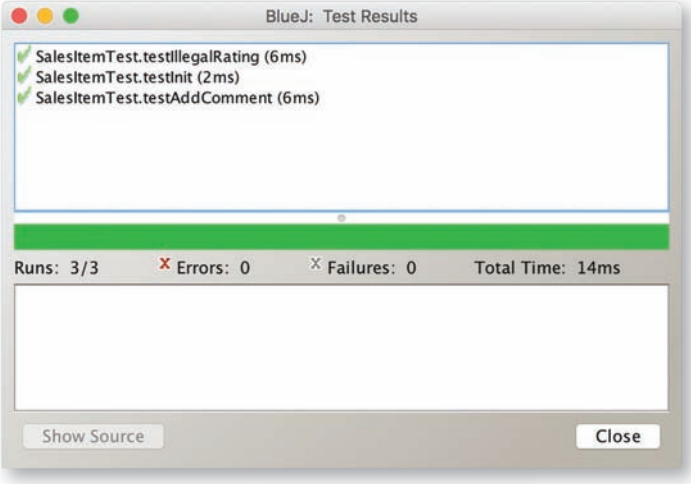
Figure 9.3
The pop-up menu for a test class



Exercise 9.12 Run the tests in your project, using the *Run Tests* button. You should see a window similar to Figure 9.4, summarizing the results of the tests.

Figure 9.4 shows the result of running three tests named **`testAddComment`**, **`testInit`**, and **`testIllegalRating`**, which are defined in the test class. The ticks immediately to the left of the test names indicate that the tests succeeded. You can achieve the same result by selecting the *Test All* option from the pop-up menu associated with the test class, or running the tests individually by selecting them from the same menu.

Figure 9.4
The Test Results window



Test classes are clearly different in some way from ordinary classes, and if you open the source code of **SalesItemTest**, you will notice that it has some new features. At this stage of the book, we are not going to discuss in detail how test classes work internally, but it is worth noting that although the source code of **SalesItemTest** could have been written by a person, it was, in fact, *automatically generated* by BlueJ. Some of the comments were then added afterwards to document the purpose of the tests.

Each test class typically contains tests for the functionality of its reference class. It is created by using the right mouse button over a potential reference class and selecting *Create Test Class* from the pop-up menu. Note that **SalesItem** already has a test class, so this additional menu item does not appear in its class menu, but the one for **Comment** does have this option, as it currently has no associated test class.

The test class contains source code both to run tests on a reference class and to check whether the tests were successful or not. For instance, here is one of the statements from **testInit** that checks that the price of the item is 1000 at that point:

```
assertEquals(1000, salesIte1.getPrice());
```

When such tests are run, BlueJ is able to display the results in the window shown in Figure 9.4.

In the next section, we shall discuss how BlueJ supports creation of tests so that you can create your own automated tests.

Exercise 9.13 Create a test class for the **Comment** class in the *online-shop-junit* project.

Exercise 9.14 What methods are created automatically when a new test class is created?

9.4.3 Recording a test

As we discussed at the beginning of Section 9.4, test automation is desirable because manually running and re-running tests is a time-consuming process. BlueJ makes it possible to combine the effectiveness of manual unit testing with the power of test automation, by enabling us to record manual tests, then replay them later for the purposes of regression testing. The **SalesItemTest** class was created via this process.

Suppose that we wanted to thoroughly test the **addComment** method of the **SalesItem** class. This method, as we have seen, adds customer comments if they are valid. There are several tests we would like to make, such as:

- adding a first comment to an empty comment list (positive)
- adding further comments when other comments already exist (positive)
- attempting to add a comment with an author who has already submitted a comment (negative)
- attempting to add a comment with an invalid rating (negative)

The first of these already exists in the **SalesItemTest** class. We will now describe how to create the next one using the *online-shop-junit* project.

A test is recorded by telling BlueJ to start recording, performing the test manually, and then signaling the end of the test. The first step is done via the menu attached to a test class. This tells BlueJ which class you wish the new test to be stored in. Select *Create Test Method . . .* from the **SalesItemTest** class's pop-up menu. You will be prompted for a name for the test method. By convention, we start the name with the prefix "test." For example, to create a method that tests adding two comments, we might call that method **testTwoComments**.[1]

Concept

An **assertion** is an expression that states a condition that we expect to be true. If the condition is false, we say that the assertion fails. This indicates an error in our program.

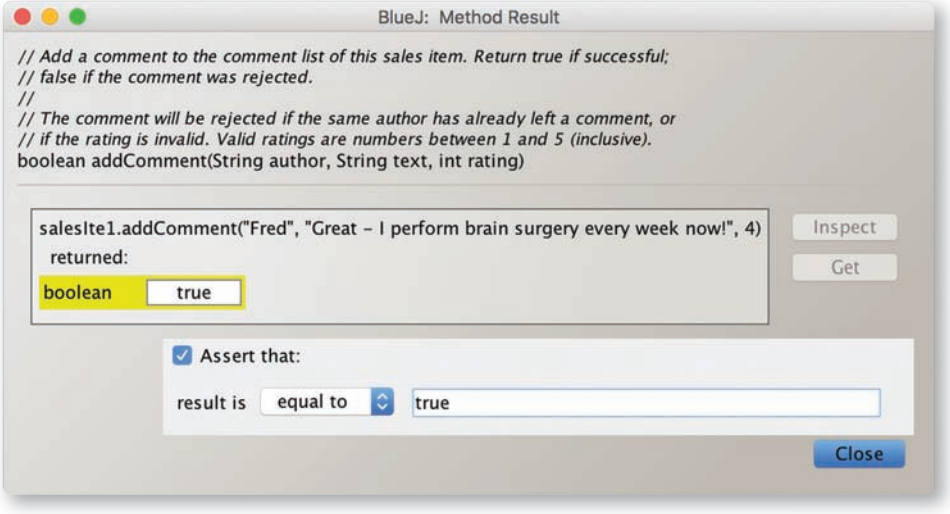
Once you have entered a name and clicked *OK,* a red recording indicator appears to the left of the class diagram, and the *End* and *Cancel* buttons become available. *End* is used to indicate the end of the test-creation process and *Cancel* to abandon it.

Once recording is started, we just carry out the actions that we would with a normal manual test:

- Create a **SalesItem** object.
- Add a comment to the sales item.

Once **addComment** has been called, a new dialog window will appear (Figure 9.5). This is an extended version of the normal method result window, and it is a crucial part of the automated testing process. Its purpose is to allow you to specify what the result of the method call *should be.* This is called an *assertion*.

Figure 9.5
The Method Result dialog with assertion facility



[1] Earlier versions of JUnit, up to version 3, required the method names to start with the prefix "test." This is not a requirement anymore in current versions.

In this case, we expect the method return value to be *true*, and we want to include a check in our test to make sure that this is really the case. We can now make sure that the *Assert that* checkbox is checked, enter *true* in the dialog, and select the *Close* button.

- Add a second comment to your sales item. Make sure the comments are valid (they have unique authors and the rating is valid). Assert that the result is true for the second comment addition as well.
- We now expect two comments to exist. To test that this is indeed the case, call the **`getNumberOfComments`** method and assert that the result is 2.

This is the final stage of the test. We then press the *End* button to stop the recording. At that point, BlueJ adds source code to the **`SalesItemTest`** class for our new method, **`testTwoComments`**, then compiles the class and clears the object bench. The resulting generated method is shown in Code 9.2.

Code 9.2
An automatically generated test method

```
@Test
public void testTwoComments()
{
    SalesItem salesIte1 = new SalesItem("Java book", 12345);
    assertEquals(true, salesIte1.addComment("James Duckling",
                                            "Great book!", 4));
    assertEquals(true, salesIte1.addComment("Fred", "Like it", 3));
    assertEquals(2, salesIte1.getNumberOfComments());
}
```

As can be seen, the method contains statements that reproduce the actions made when recording it: a **`SalesItem`** object is created, and the **`addComment`** and **`getNumberOfComments`** methods are called. The call to **`assertEquals`** checks that the result returned by these methods matches the expected value. You can also see a new construct, **`@Test`**, before the method. This is an *annotation* that identifies this method as a test method.

The exercises below are provided so that you can try this process for yourself. They include an example to show what happens if the actual value does not match the expected value.

Exercise 9.15 Create a test to check that **`addComment`** returns **`false`** when a comment from the same author already exists.

Exercise 9.16 Create a test that performs negative testing on the boundaries of the rating range. That is, test the values 0 and 6 as a rating (the values just outside the legal range). We expect these to return **`false`**, so assert **`false`** in the result dialog. You will notice that one of these actually (incorrectly) returns **`true`**. This is the bug we uncovered earlier in manual testing. Make sure that you assert **`false`** anyway. The assertion states the *expected* result, not the *actual* result.

Exercise 9.17 Run all tests again. Explore how the *Test Result* dialog displays the failed test. Select the failed test in the list. What options do you have available to explore the details of the failed test?

Exercise 9.18 Create a test class that has **`Comment`** as its reference class. Create a test that checks whether the author and rating details are stored correctly after creation. Record separate tests that check whether the **`upvote`** and **`downvote`** methods work as expected.

Exercise 9.19 Create tests for **`SalesItem`** that test whether the **`findMostHelpfulComment`** method works as expected. Note that this method returns a **`Comment`** object. During your testing, you can use the *Get* button in the method result dialog to get the result object onto the object bench, which then allows you to make further method calls and add assertions for this object. This allows you to identify the comment object returned (e.g., by checking its author). You can also assert that the result is *null* or *not null*, depending on what you expect.

9.4.4 Fixtures

Concept

A **fixture** is a set of objects in a defined state that serves as a basis for unit tests.

As a set of test methods is built up, it is common to find yourself creating similar objects for each one. For instance, every test of the **`SalesItem`** class will involve creating at least one **`SalesItem`** and initializing it, often by adding one or more comments. An object or a group of objects that is used in more than one test is known as a *fixture.* Two menu items associated with a test class enable us to work with fixtures in BlueJ: *Object Bench to Test Fixture* and *Test Fixture to Object Bench.* In your project, create two **`SalesItem`** objects on the object bench. Leave one without any comments, and add two comments to the other. Now select *Object Bench to Test Fixture* from **`SalesItemTest`**. The objects will disappear from the object bench, and if you examine the source code of **`SalesItemTest`**, you will see that its **`setUp`** method looks something like Code 9.3, where **`salesItem1`** and **`salesItem2`** have been defined as fields.

Code 9.3
Creating a fixture

```
/**
 * Sets up the test fixture.
 *
 * Called before every test case method.
 */
@Before
public void setUp()
{
    salesItem1 = new SalesItem("Java book", 12345);
    salesItem2 = new SalesItem("Harry Potter", 1250);
    salesItem2.addComment("Fred", "Best book ever", 5);
}
```