9.3 Unit testing within BlueJ

Concept

unit testing refers to tests of the individual parts of an application, such as methods and classes.

The term *unit testing* refers to a test of the individual parts of an application, as opposed to *application testing*, which is testing of the complete application. The units being tested can be of various sizes. They may be a group of classes, a single class, or even a single method. It is worth observing that unit testing can be done long before an application is complete. Any single method, once written and compiled, can (and should) be tested.

Because BlueJ allows us to interact directly with individual objects, it offers unique ways to conduct testing on classes and methods. One of the points we want to stress in this section is that it is never too early to start testing. There are several benefits in early experimentation and testing. First, they give us valuable experience with a system; this can make it possible to spot problems early enough to fix them at a much lower cost than if they had not been uncovered until much later in the development. Second, we can start building up a series of test cases and results that can be used over and over again as the system grows. Each time we make a change to the system, these test cases allow us to check that we have not inadvertently introduced errors into the rest of the system as a result of the changes.

In order to illustrate this form of testing within BlueJ, we shall use the *online-shop* project, which represents an early stage in the development of software for an online sales shop (such as Amazon.com). Our project contains only a very small part of this application, namely the part that deals with customer comments for sales items.

Open the *online-shop* project. Currently, it contains only two classes: **`SalesItem`** and **`Comment`**. The intended functionality of this part of the application—concentrating solely on handling customer comments—is as follows:

- Sales items can be created with a description and price.
- Customer comments can be added to and removed from sales items.
- Comments include a comment text, an author's name, and a rating. The rating is in the range of 1 to 5 (inclusive).
- Each person may leave only one comment. Subsequent attempts to leave a comment by the same author are rejected.
- The user interface (not implemented in this project yet) will include a question asking, "Was this comment helpful to you?" with Yes and No buttons. A user clicking Yes or No is known as *upvoting* or *downvoting* the comment. The balance of up and down votes is kept for comments, so that the most useful comments (the ones with the highest vote balance) can be displayed at the top.

The **`Comment`** class in our project stores information about a single comment. For our testing, we shall concentrate on the **`SalesItem`** class, shown in Code 9.1. Objects of this class represent a single sales item, including all comments left for this item.

As part of our testing, we should check several parts of the intended functionality, including:

- Can comments be added and removed from a sales item?
- Does the **`showInfo`** method correctly show all the information stored about a sales item?

- Are the constraints (ratings must be 1 to 5, only one comment per person) enforced correctly?
- Can we correctly find the most helpful comment (the one with the most votes)?

We shall find that all of these can be tested conveniently using the object bench within BlueJ. In addition, we shall see that the interactive nature of BlueJ makes it possible to simplify some of the testing by making controlled alterations to a class under test.

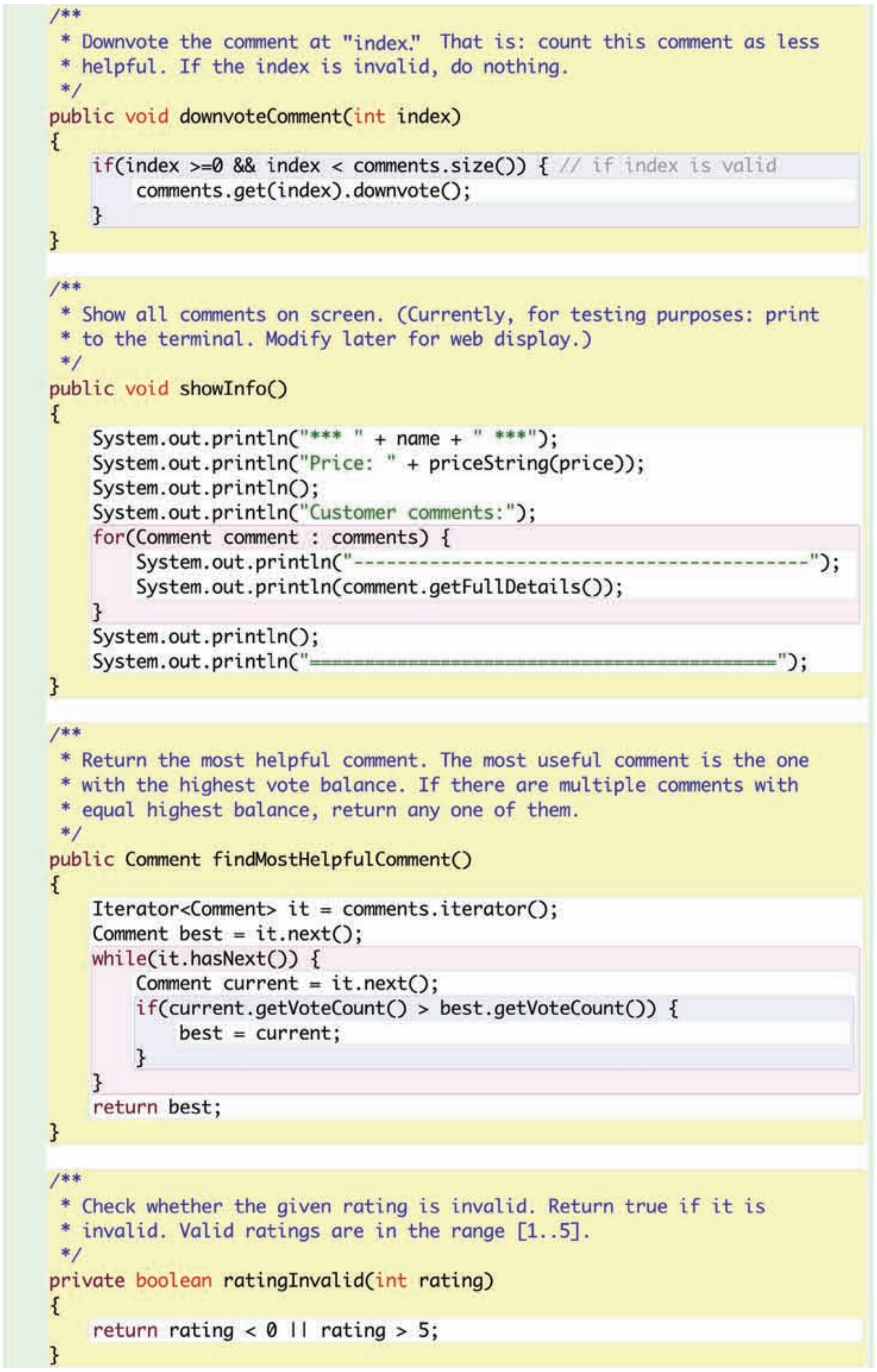
Code 9.1
The **SalesItem** class

```
import java.util.ArrayList;
import java.util.Iterator;

/**
 * The class represents sales items on an online e-commerce site (such as
 * Amazon.com). SalesItem objects store all information relevant to this item,
 * including description, price, customer comments, etc.
 *
 * NOTE: The current version is incomplete! Currently, only code dealing with
 * customer comments is here.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.1 (2016.02.29)
 */
public class SalesItem
{
    private String name;
    private int price;  // in cents
    private ArrayList<Comment> comments;

    /**
     * Create a new sales item.
     */
    public SalesItem(String name, int price)
    {
        this.name = name;
        this.price = price;
        comments = new ArrayList<>();
    }

    /**
     * Return the name of this item.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Return the price of this item.
     */
    public int getPrice()
    {
        return price;
    }
```

Code 9.1 continued
The `SalesItem` class

```
/**
 * Return the number of customer comments for this item.
 */
public int getNumberOfComments()
{
    return comments.size();
}

/**
 * Add a comment to the comment list of this sales item. Return true if
 * successful; false if the comment was rejected.
 *
 * The comment will be rejected if the same author has already left a
 * comment, or if the rating is invalid. Valid ratings are numbers between
 * 1 and 5 (inclusive).
 */
public boolean addComment(String author, String text, int rating)
{
    if(ratingInvalid(rating)) {  // reject invalid ratings
        return false;
    }

    if(findCommentByAuthor(author) != null) {
        // reject mutiple comments by same author
        return false;
    }

    comments.add(new Comment(author, text, rating));
    return true;
}

/**
 * Remove the comment stored at the index given. If the index is invalid,
 * do nothing.
 */
public void removeComment(int index)
{
    if(index >=0 && index < comments.size()) { // if index is valid
        comments.remove(index);
    }
}

/**
 * Upvote the comment at "index."  That is: count this comment as more
 * helpful. If the index is invalid, do nothing.
 */
public void upvoteComment(int index)
{
    if(index >=0 && index < comments.size()) { // if index is valid
        comments.get(index).upvote();
    }
}
```

Code 9.1 continued
The `SalesItem` class



```
/**
 * Downvote the comment at "index." That is: count this comment as less
 * helpful. If the index is invalid, do nothing.
 */
public void downvoteComment(int index)
{
    if(index >=0 && index < comments.size()) { // if index is valid
        comments.get(index).downvote();
    }
}

/**
 * Show all comments on screen. (Currently, for testing purposes: print
 * to the terminal. Modify later for web display.)
 */
public void showInfo()
{
    System.out.println("*** " + name + " ***");
    System.out.println("Price: " + priceString(price));
    System.out.println();
    System.out.println("Customer comments:");
    for(Comment comment : comments) {
        System.out.println("-------------------------------------------");
        System.out.println(comment.getFullDetails());
    }
    System.out.println();
    System.out.println("===========================================");
}

/**
 * Return the most helpful comment. The most useful comment is the one
 * with the highest vote balance. If there are multiple comments with
 * equal highest balance, return any one of them.
 */
public Comment findMostHelpfulComment()
{
    Iterator<Comment> it = comments.iterator();
    Comment best = it.next();
    while(it.hasNext()) {
        Comment current = it.next();
        if(current.getVoteCount() > best.getVoteCount()) {
            best = current;
        }
    }
    return best;
}

/**
 * Check whether the given rating is invalid. Return true if it is
 * invalid. Valid ratings are in the range [1..5].
 */
private boolean ratingInvalid(int rating)
{
    return rating < 0 || rating > 5;
}
```

Code 9.1 continued
The `SalesItem` class

```
    /**
     * Find the comment by the author with the given name.
     *
     * @return The comment if it exists; null if it doesn't.
     */
    private Comment findCommentByAuthor(String author)
    {
        for(Comment comment : comments) {
            if(comment.getAuthor().equals(author)) {
                return comment;
            }
        }
        return null;
    }

    /**
     * For a price given as an int, return a readable String representing the
     * same price. The price is given in whole cents. For example for
     * price==12345, the following String is returned: $123.45
     */
    private String priceString(int price)
    {
        int dollars = price / 100;
        int cents = price - (dollars*100);
        if(cents <= 9) {
            return "$" + dollars + ".0" + cents;  // zero padding
        }
        else {
            return "$" + dollars + "." + cents;
        }
    }
}
```

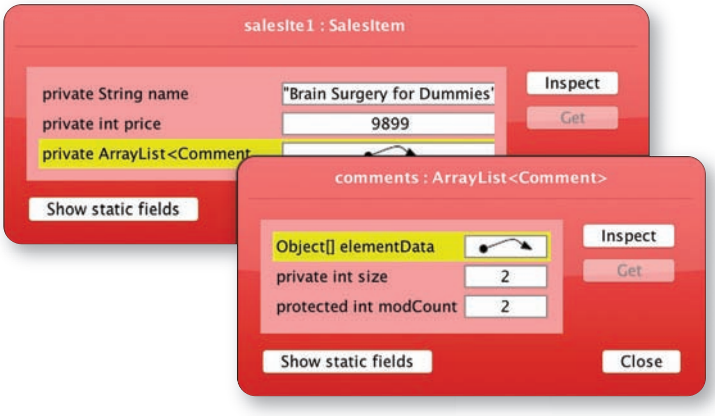
9.3.1 Using inspectors

When testing interactively, using object inspectors is often very helpful. In preparation for our testing, create a **`SalesItem`** object on the object bench and open its inspector by selecting the *Inspect* function from the object's menu. Select the **`comments`** field and open its inspector as well (Figure 9.1). Check that the list has been created (is not null) and is initially of size 0. Check also that the size grows as you add comments. Leave the comment-list inspector open to assist with subsequent tests.

An essential component of testing classes that use data structures is checking that they behave properly both when the data structures are empty and—if appropriate—when they are full. Testing for full data structures only applies to those that have a fixed limit, such as arrays. In our case, where we use an **`ArrayList`**, testing for the list being full does not apply, because the list expands as needed. However, making tests with an empty list is important, as this is a special case that needs special treatment.

A first test that can be performed on **`SalesItem`** is to call its **`showInfo`** method before any comments have been added. This should correctly show the item's description and price, and no comments.

Figure 9.1
Inspector for the `comments` list



A key feature of good testing is to ensure that *boundaries* are checked, because these are often the points at which things go wrong. The boundaries associated with the **`SalesItem`** class are, for example, the empty comment list. Boundaries set for the **`Comment`** class include the restriction of the rating to the range 1 to 5. Ratings at the top and bottom of this range are boundary cases. It will be important to check not only ratings in the middle of this range, but also the maximum and minimum possible rating.

In order to conduct tests along these lines, create a **`SalesItem`** object on the object bench and try the following as initial tests of the comment functionality. If you perform these tests carefully, they should uncover two errors in our code.

Exercise 9.1 Add several comments to the sales item while keeping an eye on the inspector for the comments list. Make sure the list behaves as expected (that is, its size should increase). You may also like to inspect the *elementData* field of the ArrayList object.

Exercise 9.2 Check that the **`showInfo`** method correctly prints the item information, including the comments. Try this both for items with and without comments.

Exercise 9.3 Check that the **`getNumberOfComments`** method works as expected.

Exercise 9.4 Now check that duplicate authors are correctly handled—i.e., that further comments by the same author are rejected. When trying to add a comment with an author name for whom a comment already exists, the **`addComment`** method should return **`false`**. Check also that the comment was not added to the list.

Exercise 9.5 Perform boundary checking on the rating value. That is, create comments not only with medium ratings, but also with top and bottom ratings. Does this work correctly?

Exercise 9.6 Good boundary testing also involves testing values that lie just beyond the valid range of data. Test 0 and 6 as rating values. In both cases, the comment should be rejected (**`addComment`** should return **`false`**, and the comment should not be added to the comment list).

Exercise 9.7 Test the **`upvoteComment`** and **`downvoteComment`** methods. Make sure that the vote balance is correctly counted.

Exercise 9.8 Use the **`upvoteComment`** and **`downvoteComment`** methods to mark some comments as more or less helpful. Then test the **`findMostHelpfulComment`** method. This method should return the comment that was voted most helpful. You will notice that the method returns an object reference. You can use the *Inspect* function in the method result dialog to check whether the correct comment was returned. Of course, you will need to know which is the correct comment in order to check whether you get the right result.

Exercise 9.9 Do boundary testing of the **`findMostHelpfulComment`** method. That is, call this method when the comments list is empty (no comments have been added). Does this work as expected?

Exercise 9.10 The tests in the exercises above should have uncovered two bugs in our code. Fix them. After fixing these errors, is it safe to assume that all previous tests will still work as before? Section 9.4 will discuss some of the testing issues that arise when software is corrected or enhanced.

From these exercises, it is easy to see how valuable interactive method invocations and inspectors are in giving immediate feedback on the state of an object, often avoiding the need to add print statements to a class when testing or debugging it.

9.3.2 Positive versus negative testing

Concept

Positive testing is the testing of cases that are expected to succeed.

When deciding about what to test, we generally distinguish *positive* and *negative* test cases. Positive testing is the testing of functionality that we expect to work. For example, adding a comment by a new author with a valid rating is a positive test. When testing positive test cases, we have to convince ourselves that the code did indeed work as expected.

Negative testing is the test of cases that we expect to fail. Using an invalid rating, or attempting to store a second comment from the same author, are both negative tests. When testing negative cases, we expect the program to handle this error in some specified, controlled way.

Pitfall It is a very common error for inexperienced testers to conduct only positive tests. Negative tests—testing that what should go wrong indeed does go wrong, and does so in a well-defined manner—are crucial for a good test procedure.