Chapter 15

Software Testing

We know that a clean compile does not imply that a program will work correctly. We can detect errors in our code as we interact with the executing program. The process of exercising code to reveal errors or demonstrate the lack thereof is called *testing*. The informal testing that we have done up to this point has been adequate, but serious software development demands a more formal approach. We will see that good testing requires the same skills and creativity as programming itself.

Until recently testing was often an afterthought. Testing was not seen to be as glamorous as designing and coding. Poor testing led to buggy programs that frustrated users. Also, tests were written largely after the program's design and coding were complete. The problem with this approach is major design flaws may not be revealed until late in the development cycle. Changes late in the development process are invariably more expensive and difficult to deal with than changes earlier in the process.

Weaknesses in the standard approach to testing led to a new strategy: *test-driven development*. In test-driven development the testing is automated, and the design and implementation of good tests is just as important as the design and development of the actual program. Tests are developed before any application code is written, and any application code produced is immediately subjected to testing.

Object-oriented development readily lends itself to such a style of testing. When the nature of a class—its interface and functionality—has been specified, special client code can be written to create and exercise objects of that class. This client code cannot be executed until the class under test has been written, but it will be ready to go when then the class becomes available.

15.1 The `assert` Statement

The `assert` statement can be used to make a claim (an assertion) about some property within an executing program. Either the property is true, or it is false. If during the program's execution the property is true, the `assert` statement is as if it were not there; however, if the property is false, the JRE generates a runtime error. For example, consider the following interactive session:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 3;
> assert x == 3;
> assert x == 2;
```



```
AssertionError:
```

In DrJava's Interactions pane the user can continue; in an executing program the error terminates the program. Notice how the statement

```
assert x == 3;
```

has no effect. Quiet assertions are silent witnesses to the programmer's correct beliefs about the way the program is working. On the other hand, the assertion

```
assert x == 2;
```

fails, revealing the truth that even though we may believe `x` should have the value of 2 here, in fact it does not.

15.2 Unit Testing

Checks based on the `assert` statement (§ 15.1) are primitive devices for verifying the correctness of programs. *Unit testing* provides a more comprehensive framework for testing. Unit testing involves testing a component (that is, a unit) of a program. The class is the fundamental building block of a Java program. Clients can use a correctly written class to create objects that function correctly. If an object fails to behave correctly in a given situation, the error resides within the code of that object's class. The idea behind unit testing is that when all the components work correctly individually they can be combined into a complete software system that overall behaves correctly. Unit testing can be performed by hand or programmatically. Test-driven development (TDD) dictates that unit tests be automated and be available before the class to test is complete.

Hand testing is time consuming and error prone. Consider the following interactive session that tests a `TrafficLightModel` (10.1) object:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> TrafficLightModel t = new > TrafficLightModel(TrafficLightModel.CAUTION);
> assert t.getState()  == TrafficLightModel.CAUTION
```

A hand tester must remember to do this test after any change to the `TrafficLightModel` class and be sure that the state used on the first line of the interaction matches the state on the succeeding line. Couple this with the fact that perhaps hundreds of tests should be performed when any changes are made, and you can see why such hand testing is impractical in general.

A better approach would be to use a custom Java class to perform the testing as in `TrafficLightCustomTester` (15.1):

```
public class TrafficLightCustomTester {
    public void testColorChange() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        assert TrafficLightModel.GO == t.getState();
        t.setState(TrafficLightModel.CAUTION);
        assert TrafficLightModel.CAUTION == t.getState();
        t.setState(182);  //  Bogus value
```



```
        assert TrafficLightModel.STOP == t.getState();
    }
    public void testShow() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        TextTrafficLight lt = new TextTrafficLight(t);
        assert lt.show().equals("[ ) ( ) (G)]");
        lt.setState(TrafficLightModel.CAUTION);
        assert lt.show().equals("[ ) (Y) ( )]");
        lt.setState(TrafficLightModel.STOP);
        assert lt.show().equals("[(R) ( ) ( )]");
    }
    public static void main(String[] args) {
        TrafficLightCustomTester tester = new TrafficLightCustomTester();
        tester.testColorChange();
        tester.testShow();
    }
}
```

Listing 15.1: `TrafficLightCustomTester`—custom class for testing traffic light objects

If this program runs silently, all tests are passed. A runtime error indicates a failed test, as one of the assertions failed. The problem with the standalone custom testing class approach is there is no automated error reporting, and each tester is responsible for devising his own reporting scheme. Fortunately standardized testing frameworks are available that address these concerns.

15.3 JUnit Testing

JUnit is a testing framework developed by Erich Gamma and Kent Beck [1]. It is used to create unit tests for Java. DrJava provides built in support for JUnit. To see how it works, we will create a JUnit test case that subjects our traffic light classes to the same series of tests as performed in `TrafficLightCustomTester` (15.1). First, from the **File** menu choose the **New JUnit Test Case ...** item. Next, enter the name `TrafficLightTester` in the test case name dialog. The following skeleton code should appear in the editor:

```
import junit.framework.TestCase;

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when
 * running
 * the test with JUnit.
 */
public class TrafficLightTester extends TestCase {

    /**
     * A test method.
     * (Replace "X" with a name describing the test.  You may write as
```



```
     * many "testSomething" methods in this class as you wish, and
     * each
     * one will be called when running JUnit over this class.)
     */
    public void testX() {
    }

}
```

The comments explain how you should edit the file to produce the desired test cases. We could change the name of `testX()` to `testColorChange()` and add code to test the state changing operation. The test methods can contain any Java code, but their purpose should be conducting tests on application objects.

Our `TrafficLightTester` class subclasses `junit.framework.TestCase` from which it inherits a great deal of functionality. The big benefit is the ability for the JUnit framework to create an instance of our `TrafficLightTester` class, run all of its test methods, and report the results in a standardized attractive way. All of this work performed by the framework is done automatically.

In `TrafficLightTester` (15.2), we write two test methods—`testColorChange()` and `testShow()`:

```
import junit.framework.TestCase;

public class TrafficLightTester extends TestCase {
    public void testColorChange() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        assertEquals(TrafficLightModel.GO, t.getState());
        t.setState(TrafficLightModel.CAUTION);
        assertEquals(TrafficLightModel.CAUTION, t.getState());
        t.setState(182);  //  Bogus value
        assertEquals(TrafficLightModel.STOP, t.getState()); // Should be red
    }
    public void testShow() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[ ) ( ) (G)]");
        t.setState(TrafficLightModel.CAUTION);
        assertEquals(lt.show(), "[ ) (Y) ( )]");
        t.setState(TrafficLightModel.STOP);
        assertEquals(lt.show(), "[(R) ( ) ( )]");
    }

}
```

Listing 15.2: `TrafficLightTester`—testing traffic light objects

After compiling the above code, select the Test option from the toolbar. You may also select the Test All Documents item from the Tools menu or press Ctrl T (Cmd T on a Mac). A program that is part of the JUnit framework is executed that creates an instance of your test class. This program executes each method that begins with the prefix `test`. We may add other methods that do not begin with `test`, but these can only serve as helper methods for the

test methods; the JUnit framework will not directly call these non-`test` methods. The results of the above test are shown in Figure 15.1: As the tests are run a green bar labeled Test Progress indicates the tests' progress. The green

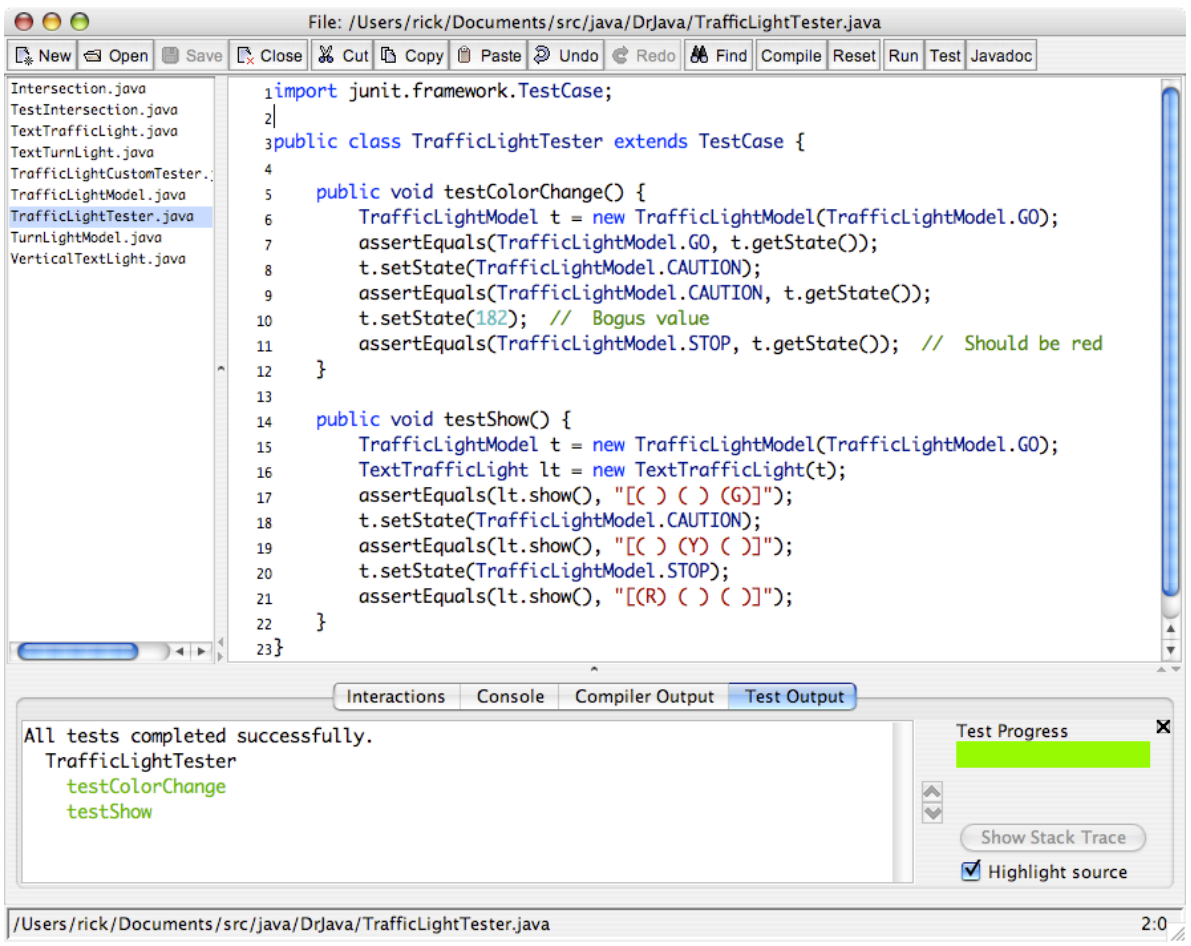

Figure 15.1: Results of a passed JUnit test

test method names shows the tests that passed. If you change Line 10 in the test to check for black instead of red, you will get the results shown in Figure 15.2: The bar turns red if a test fails. The offending test method is highlighted in red, while passing tests are green.

The `assertEquals()` method, not to be confused with the `assert` statement (see § 15.1), is the primary method that we will use in our tests. `assertEquals()` is inherited from `TestCase` and is overloaded to allow comparisons of all primitive and reference types. Its first parameter is the value expected by the tester; its second parameter is the actual value. The actual value is usually an expression to be evaluated (like `t.getState()`). Primitive types are compared as with `==`. Since all reference types have an `equals()` method (see § 14.2), the `assertEquals()` method compares reference types with `equals()`.

JUnit thus works like our original plain Java test program, but it provides an attractive report of the results of the tests. JUnit relieves developers from hand testing application classes.

While `TrafficLightTester` (15.2) is a valid test class, its design can be improved. It contains only two test methods, but it is checking six different things with `assertEquals()` calls. Each test method should be focused on checking one aspect of the object's functionality. The test method's name should be descriptive, reflecting the nature of the test being performed. Given a simple, focused test and a descriptive test name, when an `assertEquals()` fails, JUnit's report allows a tester to zoom in on the problem immediately. Also, more complex tests are difficult to maintain as the system evolves.

`BetterTrafficLightTester` (15.3) is a better test class, since it uses simpler, more focused test methods.

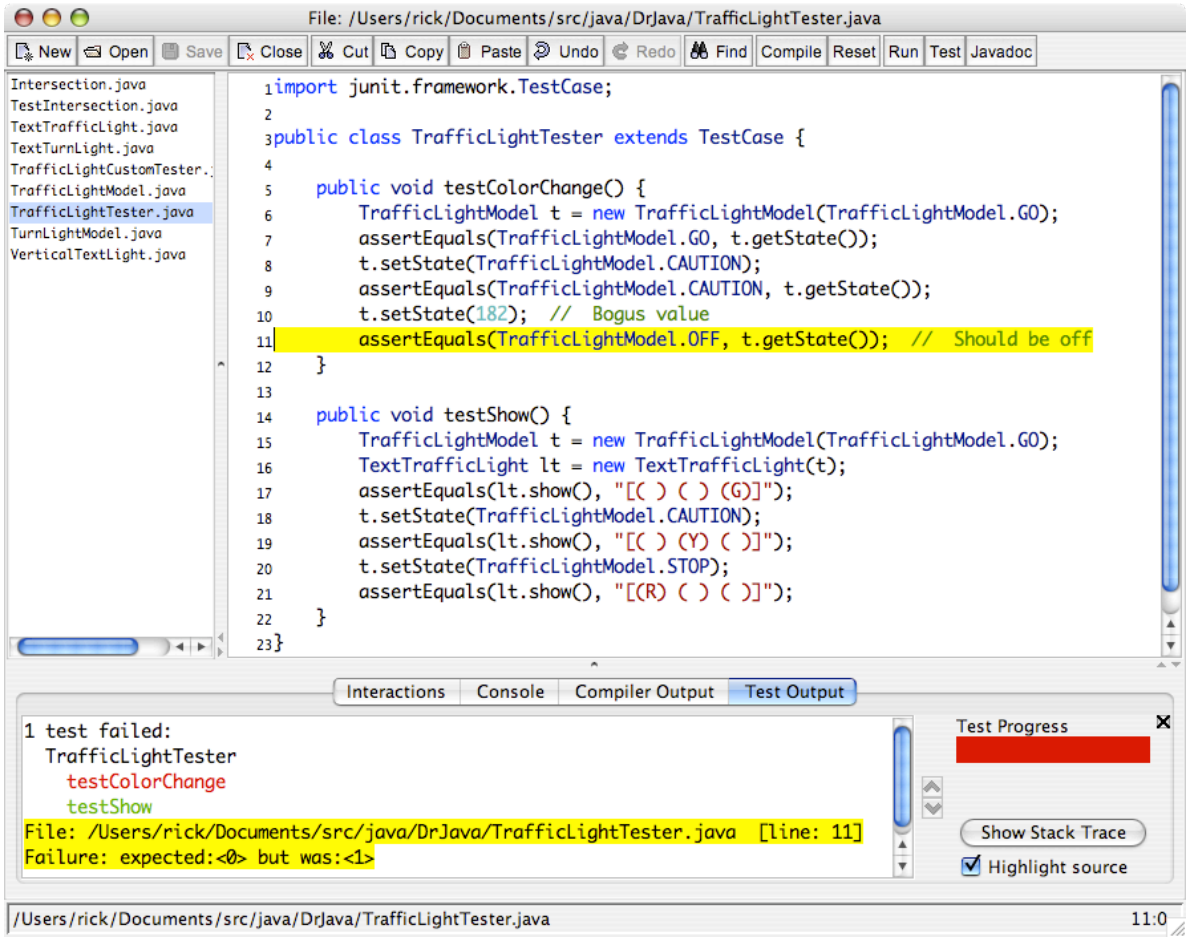

Figure 15.2: Results of a failed JUnit test

```
import junit.framework.TestCase;

public class BetterTrafficLightTester extends TestCase {
    public void testMakeGoLight() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        assertEquals(TrafficLightModel.GO, t.getState());
    }

    public void testMakeCautionLight() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.CAUTION);
        assertEquals(TrafficLightModel.CAUTION, t.getState());
    }

    public void testMakeStopLight() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.STOP);
        assertEquals(TrafficLightModel.STOP, t.getState());
    }

    public void testColorChangeGoToCaution() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        t.change();
        t.setState(TrafficLightModel.CAUTION);
```



```
    }

    public void testColorChangeStopToGo() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.STOP);
        t.change();
        t.setState(TrafficLightModel.GO);
    }

    public void testColorChangeCautionToStop() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.CAUTION);
        t.change();
        t.setState(TrafficLightModel.STOP);
    }

    public void testStopString() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.STOP);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[(R) ( ) ( )]");
    }

    public void testGoString() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[( ) ( ) (G)]");
    }

    public void testCautionString() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.CAUTION);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[( ) (Y) ( )]");
    }

    public void testOffString() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.OFF);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[( ) ( ) ( )]");
    }

}
```

Listing 15.3: `BetterTrafficLightTester`—more focused unit tests

15.4 Regression Testing

Programs, especially large commercial systems, are built incrementally. This means that a program is written with parts missing and with incomplete parts. Such a system, if constructed properly, can still be compiled and run; for example, some methods may have empty bodies because their code has yet to be written. Such a work in

progress necessarily will have limited functionality. Testers can develop appropriate test cases that test this limited functionality. As parts are added or completed, the system's functionality increases. Testers then develop additional test cases to test these new features. I

In TDD, where tests are written before the functional code they are meant to test, code is written only as a result of a failed test. This philosophy is very different from our approach thus far. Want to add a new feature to an existing program? Write the test(s) for that feature and then test the existing program. Of course the test(s) will fail, because the feature has yet to be added! You must now write the code for that feature and work on that code until all the tests are passed. In TDD the tests are crucial, since they certify that a feature behaves acceptably and is considered "correct" with respect to all the tests. Tests can contain errors, since the act of writing the tests is itself a form of programming. Fortunately TDD recommends that each test be simple and focused, so simple in fact that the potential for errors is greatly reduced. Also, focused tests are more helpful in pinpointing the location of bugs. A test that tries to do too much is known as an *eager test*. An eager test might test more than one method of an object. Should that eager test fail, the guilty method would not be immediately known; on the other hand, a focused test, one testing only one method, upon failure would implicate directly the guilty method.

Unfortunately, since software systems can be so complex, it is not uncommon for the addition of a new feature to break existing features that worked before the addition of the new feature. The new feature is likely at fault, but if the new feature has been implemented correctly it may mean the pre-existing features have some lurking errors. If indeed some existing features have errors, it means the existing test cases for those features are incorrect or incomplete. New test cases must be added for those existing features to demonstrate their problems, and then the existing features can be corrected.

Since test-driven development mandates that tests are written early, features are added to a system only when tests are available to assess the correctness of those features. The tests for existing features are retained to ensure that the new features do not break existing features. Such a strategy is called *regression testing*. Test cases that apply to a given feature are removed from the test suite only if that feature is removed from the system. The number of test cases naturally continue to grow as functionality increases. As you can see, an automated testing framework such as JUnit is essential for managing regression testing.

15.5 Summary

- The `assert` statement checks at runtime the value of a Boolean expression.
- An assertion that evaluates to true does not affect a program's runtime behavior.
- An failed assertion results in a runtime error.
- Unit tests check the correctness of components in isolation from the rest of the system.
- The JUnit testing framework provides testers an automated way of performing unit tests, and it reports the results of the tests in a standard way.
- DrJava can generate a skeleton JUnit test case.
- JUnit's test methods begin with the prefix `test...`.
- The `assertEquals()` methods of JUnit's `TestCase` class compares expected values to expressions. Successes and failures are tracked by the JUnit framework.

15.6 Exercises

1. What is the purpose of Java's `assert` statement?
2. How do you use Java's `assert` statement?
3. What are two courses of action that program execution can take when an `assert` statement is encountered?
4. What does TDD stand for?
5. What is one tenet of TDD that assists in writing correct tests?
6. What is unit testing?
7. Provide the JUnit `TestCase` method call that would check if integer variable `x` has the value 100.
8. What prefix must begin all JUnit test methods?
9. Does the `assertEquals()` method of the `TestCase` class have anything to do with Java's `assert` statement?

