Chapter 10

Test First—
Learning a New Way of Life

In this chapter, you’ll follow an example of writing test-first code to create a version of Conway’s *Game of Life*. This program is a classic and you may have a favorite way of coding it that is much simpler or more flexible than the one presented here. You could, of course, refactor this code to improve it but refactoring is a topic for another chapter. Here the goal is to take you through the cycle of writing a test, writing the code that makes the test code compile, and then run. As you’ll see in the section on testing GUIs, you don’t test everything; you just test what can break.

In this and the other chapters in Part III, each practice we focus on will be viewed somewhat in isolation. Ordinarily, user stories are broken into tasks. You would be working on a particular task when you decide to write your first test for it. In this chapter we will look at test-first development in a way that allows us to focus on both writing and running tests and then seeing the effect of this technique on your code.

OVERVIEW OF TESTING FIRST

Like most of what we do in XP, the testing cycles are short. You should be able to code a little and see the results quickly, although how quickly will depend on the task at hand. The key idea is that you cycle through tiny steps and end up coding more quickly and more robustly than if you try to take on a big task all at once. As you code, you will develop a suite of unit tests that you will run as often as possible. Save, compile, run the tests.

As an analogy, think of this suite of tests as if it were a spelling and grammar checker in a word processor. These tools make sure that the words you use are spelled correctly and that the sentences you put together obey basic rules of usage. The tools don’t tell you that what you are writing will interest your reader nor will the tools make the argument you are trying to make. In the same way, the unit tests you write will tell
you that local pieces of code behave the way they are intended to. Unit tests do not offer information about global behavior. For that, the client (perhaps with your help) will write acceptance tests. This is like having someone else read what you’ve written and tell you whether it convinces or moves them.

When using a spell checker you’ll come across words that the spell checker doesn’t recognize. (In this document the word “refactoring” was consistently flagged.) A good spell checker will allow you to add words to its dictionary. Similarly, as you program, you may find a section of code that passes all the tests but still has a bug in it. Add that test to your suite of tests and then fix the code. A spell checker won’t catch a mistake such as, “Can you here the music?” The word “here” is spelled correctly; it’s just the wrong word. The question should read, “Can you hear the music?” This mistake can be picked up only by a tool that understands the context in which “here” and “hear” appear. Perhaps a second pair of eyes will pick up this mistake, perhaps not. There are errors in your code that can’t be picked up by unit tests. Again, a full set of acceptance tests does address these problems.

Here’s a look at the testing cycle:

1. Think of the next small step that you want to accomplish. You’ll get a feel for the size of the step as you work through the example in this chapter. You may be tempted to take larger steps. But, by taking small steps that you can validate, you’ll end up moving very quickly.

2. Think of how you will test what you’ve accomplished this step. This is white box testing. You have complete access to the code you’re testing. Suppose you are testing that two ints are summed correctly. You might check that the sum of four and five is nine. This example is intentionally over simplified.

3. Write the code for one of your tests. Although you can choose another testing framework, JUnit is the one described in this book. Details of the JUnit framework appear later in this chapter. You will need to check out the JUnit javadocs to see the test methods that you can use. In the case where you’re summing two ints, make up a name for the method you’ll be testing. Let’s call it add(). You’ll need a class containing this method. Let’s call it Calculator. Then, if calculator is an instance of Calculator, calculator.add(4,5) should return 9. Once you save your test and try to compile it, you’ll discover that there’s a problem. Your code won’t compile. Your next step is now clear.

4. Write just enough code so that your test compiles. You need to make sure there is a Calculator class and that it contains an add() method that takes two ints and returns an int. What may be difficult at this point is that you aren’t trying to write the code that does everything it needs to do. At this point add() could just return the integer -3. This doesn’t happen to be the correct answer, but it will help your code compile. Once it compiles you move on.

5. Your test should be failing at this point. There are simple cases in which just adding a constructor will enable your test to pass. In most cases, there will be a middle ground where your code compiles but the test fails.

6. Now write just enough code to make your test pass. In this example with add(), it should just return the sum of the two ints passed in. You may be tempted to
take care of other issues, but you need to stay focused on your current goal, which is to pass the test.

7. Look for the next test you have to write. Go back to step 2 and consider whether there are more tests that have to be written to make sure that you have achieved your small step. You don’t need to test every case. You don’t need to add two positive ints, a positive and a negative, and so on.

8. Now return to step (1) and continue with the next small step. Now you see that you’ve been in a nested loop all along. You can refactor code at any point of the process if you need to. The best time to refactor code is after it passes all of the tests.

You’ll work through actual code examples in the rest of the chapter to see how this process comes together. The key is not to do more at any step then you need.

THE RULES OF LIFE

John Conway’s Game of Life takes place on an infinite grid with square cells. Each cell has as neighbors the eight cells that touch it. At time zero, you can set up the board in any state you wish and start the game. The state of the board at each discrete point in time is determined by the state of the board at the previous time by the following rules:

• A live cell stays alive only if it has two or three live neighbors, otherwise it dies.
• A dead cell comes to life only if it has three live neighbors, otherwise it remains dead.

These are also sometimes listed as three rules.

• A cell with two live neighbors remains in the same state.
• A cell with three live neighbors will be alive in the next period of time.
• A cell with fewer than two live neighbors dies of loneliness and with more than three dies of overcrowding.

There are many variations on this game. Some are played on a finite board with different rules for dealing with the edges. Some are played on a board with different shaped cells and different rules for life or death. In this example, assume that the board is a 30 by 40 cell rectangle and that you identify the left and right edges and the top and bottom edges. In other words if you are traveling across the third row from right to left and fall off the end of the board, you will reenter on the right side of the board still in the third row. (Some may recognize this as the torus identification.)

In each time period, you will visit each cell in the board twice. There are other ways to implement this game, but this will do for the example. The first time you visit a cell you will check to see if it will change in the next time period. The second time you visit a cell you will change it if it needs changing. You can implement this in a single pass but then you have to be careful that cells that change don’t change the information of neighboring cells before the neighboring cells decide whether or not they need to change. You can also implement this so that not all cells must be visited in each pass. Again, this nicety is beyond the needs of this example.
**SETTING UP JUNIT**

The tool you’ll use for running the unit tests that you’ll be writing throughout this course is JUnit. JUnit is an open source-testing framework for Java programmers originally created by Kent Beck and Erich Gamma. You can find out more about the project, download the latest version of the software, and investigate extensions and alternatives to JUnit at www.junit.org.

Using JUnit is as simple as downloading the zip file and unzipping it. There really is nothing to install. Commercial Integrated Development Environments (IDEs) such as Borland’s JBuilder and IntelliJ’s IDEA, as well as open-source IDEs such as Eclipse and Netbeans have integrated JUnit into their environment. When you use one of these tools, after setting application-specific project properties, you can usually run your test suite automatically. Even without such tools, JUnit has been designed to facilitate the automatic running of a suite of tests. Read the examples and articles that are included in the download of www.junit.org and you will be well on your way. The JavaDocs for JUnit are also included in the download.

As you begin a new project you need to provide some basic structure for running your suite of tests. The basic structure is generally the same on each project. Once it is in place, you can concentrate on adding tests and writing the code that passes them. In the following section you’ll look at the specifics of setting up the basic structure you’ll need.

**Setting Up the Infrastructure**

You will create a suite of tests that in turn call other suites of tests or classes that extend the JUnit class `junit.framework.TestCase`. The classes that extend `TestCase` will contain a number of tests that will be described in the section titled *Extending TestCase*.

For now, create a class named `AllTests` in the top level of your package hierarchy. In this example, you’ll create a package called `life` and inside of it a class called `AllTests`. For those who are not familiar with working with packages, this means that you will create a directory named `life`. Inside this directory, you will create a file named `AllTests.java` with the following code.

```java
package life;
import junit.framework.Test;
import junit.framework.TestSuite;
public class AllTests {
    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }
    public static Test suite () {
        TestSuite suite= new TestSuite("All JUnit Tests");
        return suite;
    }
}
```

The method `suite()` will be where you link to your other suites or test classes. For now, `suite()` doesn’t call anything. You just call the `TestSuite` constructor. Later
you will add TestCase classes or other suites to the TestSuite that you are calling suite. This does requires that you add the two import statements at this point to import the Test interface and TestSuite class that implements this interface.

The method main() can be used to run your test suite using the text-based TestRunner. In the next section, Running a Suite, you’ll see how you might run AllTests. First, you’ll need to compile AllTests.java. You’ll need to include the junit.jar file in the classpath when you compile and when you run AllTests. From just outside of the life directory compile AllTests using the following command.

```java
javac -classpath <junit.jar's classpath> life/AllTests.java
```

The code should compile at this point. If it doesn’t you may have to add :. to the end of the classpath to point to your current directory on a UNIX-based machine or ;. to the end of the classpath to point to your current directory on a machine using Windows.

**Running a Suite**

It may seem premature to run the suite. It doesn’t do anything yet. The key in test-first design is to compile and run the tests very frequently. The more often you run the tests, the easier it will be for you to track down mistakes. The difference between the code that passed all of the tests the last time you ran them and the code that is currently failing tests must be the code you’ve written since the last time you ran the tests. The more frequently you run the tests, the more certain you can be of where the problem lies.

You could easily say that there are no tests to run and you’d be right. However, if you wait you won’t know if it’s the initial setup that’s wrong or if it’s the test code you’ll be adding. Run the tests.

There is a variety of ways to run the tests. As AllTests contains a main() you can just run AllTests from the command line.

```java
java -classpath <junit.jar's classpath> life/AllTests
```

As was the case when you were compiling, if this doesn’t run you may have to add :. to the end of the classpath on a UNIX-based machine or ;. to the end of the classpath on a Windows-based machine. If it runs successfully you should see an indication of how much time it took and a message that reads something like O.K. (0 tests).

You can also run the tests using one of JUnit’s GUI test runners. From the command line, from just outside of the life directory, you would enter the following command.

```java
java -classpath <junit.jar's classpath> junit.swingui.TestRunner
dlfe/AllTests.java
```

This time you are running the application with main() method in the class junit.swingui.TestRunner and passing it the argument life/AllTests.java. If this is your preference, then you can eliminate the main() method from AllTests.java.

You’ve heard about the green bar that you get when your tests pass. You may feel a bit cheated at this point because there is no bar at all. Then again, there are no test cases to pass or fail either. As you add test cases, you will either get a green or red bar.
As mentioned before, many IDEs have integrated support for JUnit. Most that do allow you to choose whether you want to use the GUI or text-based test runners. Many IDEs also offer their own test runner, which presents the results of running JUnit differently from the standard offerings.

**CREATING YOUR FIRST TEST**

The following is our plan for setting up the first test. Begin by creating a class called `TestCell` that extends `junit.framework.TestCase`. You'll add a pointer from `AllTests` and your test suite fails. Things will seem to get worse before they get better. If you try to add a method that specifies an actual test, your code won't even compile. Your test refers to a class that doesn't exist yet. Even if the class existed, the methods your test is calling in the class haven't been written yet. Next, stub out the class and get your code to compile. Again, your test will fail. Write just enough code to get the test to pass. Once the test passes you can take another look at your code and decide if you need to do any refactoring.

**Extending TestCase**

Create a subdirectory of the life directory called model. Inside model, create a new file called `TestCell.java` containing the following code.

```java
package life.model;
import junit.framework.TestCase;
public class TestCell extends TestCase {
}
```

You will use this basic template for all of your classes that will contain the actual tests. They will all extend `junit.framework.TestCase`. If you are using a version of JUnit prior to the 3.8 release, your class will also require a constructor like this.

```java
public TestCell(String name) {
    super(name);
}
```

In any case, you will then create methods for running various tests. These methods will have the following signature:

```java
public void test <name of test here>();
```

JUnit uses reflection to run any method that begins with the name test. For each testxxx method JUnit calls the method setUp() followed by testxxx() followed by tearDown(). This allows you to put code in the body of setUp() that initializes variables and sets up the fixture the way you want. If you need different set-ups for your tests, create a new class that extends TestCase. The tearDown() method is similarly used to clean up changes you've made to the environment so that test results don't interfere with each other. You can explicitly call the test methods and not use reflection if that is a requirement.
Add the line highlighted below to the suite() method in AllTests.

```java
public static Test suite() {
    TestSuite suite = new TestSuite("All JUnit Tests");
    suite.addTest(new TestSuite(life.model.TestCell.class));
    return suite;
}
```

This addition adds the contents of the class TestCell to this TestSuite so that the tests you add to TestCell will be run as part of this suite. Save and compile AllTests.java and TestCell.java. Run AllTests whichever way appeals to you. Perhaps to your surprise, you get the message that there is one failure. If you’re using the GUI test runner you’ll encounter your first red bar. You are given the helpful message that the cause of the failure is that there are no tests in the class TestCell. This is a relief. You knew there weren’t any tests yet. You’ll actually come to appreciate seeing the test suite fail before it passes—this let’s you know that the mechanism is working. In this case, you know that running AllTests actually does cause TestCell to be called.

**Where Do Tests Come From?**

Test first is a great practice whether or not you are doing XP. If you are doing XP then you are writing tests to accomplish the next step in a task. A task is a developer-defined component of completing a user story. Perhaps you have a user story like this.

**Initialize Board**

The board will consist of a rectangular grid of square cells. Each cell has as its neighbors the eight cells that border it in any way. A cell is initially dead.

Then maybe one of your tasks is:

**Initialize Cell (Task for Initialize Board)**

Create a cell that is initially dead and has links to its eight neighbors.

Your first step may be to create a `Cell` object that is initially dead. You’ll write the test for this and then write the code in a minute. In order to concentrate on different issues in test-first programming, the sequence in this chapter does not follow a set of tasks to be completed. For each step, however, you can imagine the task that underlies the example being presented.

**Writing Your Test**

Start by writing a simple test. Create a new `Cell` object and then check to make sure that the newly created `Cell` isn’t alive. Give your test method a descriptive name because
if it fails the error message containing the method name will help you locate what’s going wrong. You could write the first test method in `TestCell` like this.

```java
public void testNewCellIsDead(){
    Cell cell = new Cell();
    assertTrue(!cell.isAlive());
}
```

When you try to compile this code you will get compiler errors that indicate that `Cell` is an unknown type and that `isAlive()` is an unknown method. Of course, they’re unknown; you haven’t created them yet. You have correctly written code that tries to create a new `Cell` object even though you know that the class used to construct such an object doesn’t exist yet. In writing this test you’ve made some decisions about what the `Cell` object looks like. You know that it has a no argument constructor and contains a method named `isAlive()` that returns a `boolean`.

The `assertTrue()` method that you used in `testNewCellIsDead()` is provided in the `junit.framework.Assert` class. Check the JUnit JavaDocs for the full list of assert-like methods that are available to you. Most are named `assertTrue()`, `assertEquals()`, `assertNull()`, `assertNotNull()`, or `assertSame()`. `Assert` is the parent class to `TestCase`.

### Getting It to Compile

What’s the least that you can do to get the code to compile? You need to create a class named `Cell`. Inside of the `model` directory create `Cell.java` containing the following code:

```java
package life.model;
public class Cell {
}
```

Save your work and recompile the project. You may notice that the test code still can’t compile. You’re right, but take small steps.

The compile fails because `Cell` doesn’t contain an `isAlive()` method yet. Add this code to `Cell` now.

```java
protected boolean isAlive(){
    return true;
}
```

The code compiles. As soon as code compiles, run your tests. This process has to become habitual. Save, compile, run the tests. The test fails. You get an `AssertionFailedError`. Your test expected `cell.isAlive()` to be false in this case. Your next task is to get the test to pass.

### Getting It to Pass

If the goal is to get the test to pass, the easiest way would be to just return `false` instead of `true` from `isAlive()`. Try that. Save, compile, and run the test suite. The test passes. This hardly feels satisfying.
Maybe what’s needed is another test that forces a better solution. Add this test to
the TestCell class.

```java
public void testLiveCellIsAlive(){
    Cell cell = new Cell();
    cell.setAlive(true);
    assertTrue(cell.isAlive());
}
```

Here you create a new Cell and set it to be alive and then check that it is alive.
Save and compile and you see that it won’t compile. Add the setAlive() method to the
Cell class. The simplest version does nothing for now.

```java
protected void setAlive(boolean alive){
}
```

The code compiles but the new test fails. The methods setAlive() and getAlive() are obviously accessors, so introduce the boolean alive and rewrite the body of the
methods. Cell.java now looks like this.

```java
package life.model;
public class Cell {
    private boolean alive;
    protected void setAlive(boolean alive){
        this.alive = alive;
    }
    protected boolean isAlive(){
        return alive;
    }
}
```

Save, compile, and run the tests. Both tests pass. Look around to see if any code
needs to be cleaned up.

**Look Around for Refactoring**

There isn’t much code so there isn’t much to clean up. The Cell class looks pretty good.
Refactoring can be as simple as changing the name of a method or variable. If you feel
the need, go ahead and change one of the names. If you change the variable alive, you
will also have to change references to it. Similarly, if you change the names of the ac-
cessor methods you will have to change the references to them.

You can, however, benefit from refactoring the TestCell class. In each of your
test methods you have declared and initialized a Cell object. Instead, create an in-
stance variable and initialize it in the setUp() method as follows:

```java
package life.model;
import junit.framework.TestCase;
public class TestCell extends TestCase{
```
138

Core Practices Part III

protected void setUp() {
    cell = new Cell();
}

public void testNewCellIsDead(){
    assertTrue(!cell.isAlive());
}

public void testLiveCellIsAlive(){
    cell.setAlive(true);
    assertTrue(cell.isAlive());
}
}

This removes duplicated code and makes sure that we’ve initialized the cell variable before running each test. Refactoring does not have to be grand and dramatic. You’ve made a small change that will make future tests easier to add. Your test code may end up being twice as long as your production code. You’ll need to keep your test code clean so that it can be easily understood.

Take a quick look back at what you’ve done. You started with a simple test: when a cell is created verify that it is dead. By writing this test you helped shape the code being tested. A second test kept you honest. Now that those two tests were passing you looked to see if you could clean up your code. This clean up applied to both the production code and to the test code. You worked in small steps and began to feel the rhythm of save, compile, and run the tests.

WRITING MORE TESTS

Be wary of times when you choose not to test first. You will think of a new really easy feature to implement and just want to write the code without the tests. You should resist this impulse for three reasons. First, your accuracy in judging what is easy may not be consistently reliable. Second, having the test code will allow you to refactor with confidence. Third, long after you’ve forgotten about what you were thinking when you wrote this code, someone will come along and break what you’ve done. If you have a test in place then, assuming that they run the tests before they check their code in, they will know immediately that they’ve broken your code and will know where to look. It is easier to write the tests as you go and you will write better code if you write the tests first.

Checking for Changes in Live Cells

So far, you’ve written code that allows Cell objects to be initialized and set to be alive or dead. Now you can write the algorithm for getting from this time period to the next. Start with a test. You know that a live cell with two or three live neighbors will stay alive in the next time period, otherwise it will die. Here is a sequence of tests you could write for this case.
public void testLiveCellWithTwoLiveNeighborsStaysAlive(){
    cell.setAlive(true);
    for (int i = 0; i < 2; i++){
        cell.incrementNumberOfLiveNeighbors();
    }
    assertTrue(!cell.needsToChange());
}

public void testLiveCellWithThreeLiveNeighborsStaysAlive(){
    cell.setAlive(true);
    for (int i = 0; i < 3; i++){
        cell.incrementNumberOfLiveNeighbors();
    }
    assertTrue(!cell.needsToChange());
}

public void testLiveCellWithFourLiveNeighborsDies(){
    cell.setAlive(true);
    for (int i = 0; i < 4; i++){
        cell.incrementNumberOfLiveNeighbors();
    }
    assertTrue(cell.needsToChange());
}

public void testLiveCellWithOneLiveNeighborsDies(){
    cell.setAlive(true);
    cell.incrementNumberOfLiveNeighbors();
    assertTrue(cell.needsToChange());
}

Of course, you would write these tests one at a time and get them to compile and pass. You could write more tests but this is a representative bunch that you hope will cover enough of your cases. Add these four tests to TestCell and you’ll find that it doesn’t compile. You need to define the methods `incrementNumberOfLiveNeighbors()` and `needsToChange()` in Cell. The intent of `incrementNumberOfLiveNeighbors()` is to increase this particular cell object’s count of its live neighbors. Introduce an instance variable of type `int` called `numberOfLiveNeighbors` and let the body of `incrementNumberOfLiveNeighbors()` increase the value of `numberOfLiveNeighbors` by one. The method `needsToChange()` needs to return false for a live cell with two or three neighbors and true otherwise.

Here’s the code for Cell including an accessor method for `numberOfLiveNeighbors`.

```java
package life.model;

public class Cell {
    private boolean alive;
    private int numberOfLiveNeighbors;

    protected void setAlive(boolean alive){
        this.alive = alive;
    }
```
protected boolean isAlive(){
    return alive;
}

protected void incrementNumberOfLiveNeighbors(){
    numberOfLiveNeighbors++;
}

protected int getNumberOfLiveNeighbors(){
    return numberOfLiveNeighbors;
}

protected boolean needsToChange(){
    if (isAlive() &&
        (getNumberOfLiveNeighbors()==3 ||
         getNumberOfLiveNeighbors()==2) ) return false;
    else return true;
}
}

Save, compile, and run the tests and you’ll see that all six of the tests pass. Look at the needsToChange() method. It only works for live cells. It can’t possibly work for dead cells because it always returns true for them. This is a recurring theme. Don’t write the logic for the more complicated case because you don’t yet have an official need for the harder case. When you have that need, you will have tests to help you keep on track.

Check for Changes in Dead Cells
You can come up with analogous test cases for dead cells. Add these four tests to TestCell.

    public void testDeadCellWithTwoLiveNeighborsStaysDead(){
        for (int i = 0; i < 2; i++){
            cell.incrementNumberOfLiveNeighbors();
        }
        assertTrue(!cell.needsToChange());
    }

    public void testDeadCellWithThreeLiveNeighborsLives(){
        for (int i = 0; i < 3; i++){
            cell.incrementNumberOfLiveNeighbors();
        }
        assertTrue(cell.needsToChange());
    }

    public void testDeadCellWithFourLiveNeighborsStaysDead(){
        for (int i = 0; i < 4; i++){
            cell.incrementNumberOfLiveNeighbors();
        }
        assertTrue(!cell.needsToChange());
    }

    public void testDeadCellWithOneLiveNeighborsStaysDead(){

The code compiles with these additions, but three of the four tests fail. Change Cell accordingly. Here’s the new version of needsToChange().

```java
protected boolean needsToChange(){
    if (getNumberOfLiveNeighbors()==2) return false;
    else if (isAlive()) {
        if (getNumberOfLiveNeighbors()==3) return false;
        else return true;
    } else if (getNumberOfLiveNeighbors()==3) return true;
    else return false;
}
```

Run the tests and you’ll see that all ten of them are now passing.

**What You Don’t Have**

You don’t have an exhaustive set of tests and you probably won’t need them. A comprehensive test suite would require eighteen tests to check the behavior of live and dead cells with zero to eight neighbors. What you have instead, is a representative collection of tests. You have tests that check what happens when live cells have one, two, three, or four neighbors. Testing both one and four may be excessive but it allows you to check that live cells can die from loneliness and from overcrowding even though the resulting needsToChange() method lumps these cases together. You may be tempted to check for zero, five, six, seven, and eight neighbors. At this point that would be overkill and wouldn’t add anything. If later you notice errant behavior you may come back to add more tests. You weren’t forced to alter the logic of needsToChange() until you also tested what happens to dead cells with one, two, three, or four neighbors. In checking for changes, you have a representative set of tests.

You also don’t have a running application. You can’t start up your Game of Life and see some shell running. This may seem contrary to your experience. Usually you get some part of your application working and then you run it to make sure that it is correct. In this case, what would you look at? You haven’t even thought about the GUI yet. Even so, you have a lot of confidence that this corner of your application works as needed. You’ve produced one class containing thirty-two lines of production code and two classes containing ninety-eight lines of test code. Although the division does not need to be this dramatic, you should be concerned if you have more classes than test methods.

**ALLOWING CELLS TO CHANGE**

At this point, you may be tempted to add a GUI. Try to delay creating your GUI until the model is a little more complete. Sure, you can go ahead, create your GUI, and later refactor it; but the fear is that you’ll put too much functionality in your GUI. Your goal is to make the user interface as thin as possible. The client prioritization of user stories
will drive the order in which you add functionality. For the purposes of this example, let’s next add the ability for a live cell to die and for a dead cell to come alive.

**The Test Cases**

You want to choose a representative set of test cases. You know that `needsToChange()` is working properly. You only need to check that a live cell that needs to change does and a live cell that doesn’t need to change doesn’t. Similarly, check for proper behavior in a dead cell that needs to change and one that doesn’t. You can add this check to your existing tests. Change `testLiveCellWithTwoLiveNeighborsStaysAlive()` as highlighted below.

```java
public void testLiveCellWithTwoLiveNeighborsStaysAlive(){
    cell.setAlive(true);
    for (int i = 0; i < 2; i++){
        cell.incrementNumberOfLiveNeighbors();
    }
    assertTrue(!cell.needsToChange());
    cell.change();
    assertTrue(cell.isAlive());
}
```

This won’t compile because `Cell` doesn’t contain a `change()` method. Add the following method to `Cell`.

```java
protected void change(){
}
```

The code now compiles and the tests all pass. So, you’ve handled the case where nothing is supposed to happen. Now change `testLiveCellWithOneLiveNeighborDies()` as highlighted below.

```java
public void testLiveCellWithOneLiveNeighborsDies(){
    cell.setAlive(true);
    cell.incrementNumberOfLiveNeighbors();
    assertTrue(cell.needsToChange());
    cell.change();
    assertTrue(!cell.isAlive());
}
```

The code compiles but the test fails. Change the `change()` method like this.

```java
protected void change(){
    if (needsToChange()) setAlive(!isAlive());
}
```

The tests pass. Change two tests for dead cells to confirm that the code behaves correctly. Here are the adjusted tests.
public void testDeadCellWithThreeLiveNeighborsLives(){
    for (int i = 0; i < 3; i++){
        cell.incrementNumberOfLiveNeighbors();
    }
    assertTrue(cell.needsToChange());
    cell.change();
    assertTrue(cell.isAlive());
}

public void testDeadCellWithOneLiveNeighborsStaysDead(){
    cell.incrementNumberOfLiveNeighbors();
    assertTrue(!cell.needsToChange());
    cell.change();
    assertTrue(!cell.isAlive());
}

They compile and all tests pass.

Refactoring

As an illustration of how you might refactor at this point, notice that the change() method calls needsToChange(). This means that when you poll all of the cells to see if they need to change and then you come back to tell the cells that need to change to go ahead and do so, at that point you’re asking them whether they need to change. Instead, refactor to store the information of whether or not a change is needed in a private boolean named changeNeeded. Alter the needsToChange() method like this.

private boolean changeNeeded;
protected boolean needsToChange(){
    changeNeeded = false;
    if (isAlive()){
        if (getNumberOfLiveNeighbors() !=2 &&
            getNumberOfLiveNeighbors() !=3){
            changeNeeded = true;
        }
    } else if (getNumberOfLiveNeighbors()==3)
        changeNeeded = true;
    return changeNeeded;
}

The code compiles and the tests pass. Now you can refactor the change() method to check the results of needsToChange() on the previous polling of cells.

protected void change(){
    if (changeNeeded) setAlive(!isAlive());
}

The code compiles and the tests run. At this point, you may want to change the return type of needsToChange() to void and introduce an accessor method, getChangeNeeded(). This would require refactoring the tests. It’s probably a bit cleaner
because `needsToChange()` would then do what its name suggests and `getChangeNeeded()` would be used to find the status of the result.

In addition, as one reviewer noted, this particular refactoring may be viewed as dangerous because we’ve introduced the instance variable `changeNeeded`, whose significance is temporary. More appropriate examples of refactorings are included in the Refactoring tutorial in Chapter 13. Rather than continue with this path, let’s look at the GUI for a Cell.

### TESTING GUIs

Before discussing how you might introduce tests for the GUI for the Game of Life, consider how you might have tested this application in the past. You might have coded enough to get the game running and tested it with different configurations to see if everything seems to be behaving properly. In a small application like this one you could probably get away with that strategy. In the Game of Life, there are certain patterns that have well-known behavior. For example, suppose your initial board configuration has three live cells: one cell somewhere in the middle of the board along with the cell immediately to its right and the cell immediately to its left. In the next time period, the middle cell will remain alive while the right and left cells will die. In addition, the cells immediately above and below the center cell will come to life. In other words, the horizontal row of three live cells will be replaced by a vertical row of three live cells with the same center. This so-called “Blinker” pattern will repeat. This is a nice visual test. Start with the horizontal row and you should see this oscillating pattern. The test first idea is that you don’t want to have to wait that long to see if your application is behaving properly. If you have to wait that long, you will then have to search for where the errors are. In test first, you have a pretty good idea where the problem lies and can begin working on the problem instead of spending a great deal of time locating it. You’ll need to keep your GUI thin and continue to write automated tests.

### A Second Extension to TestCase

Create a new subdirectory of `life` called `gui`. In it put `TestCellPanel.java` with the following code.

```java
package life.gui;
import junit.framework.TestCase;
public class TestCellPanel extends TestCase {
}
```

You’ll also need to add the following line to the `suite()` method in `AllTests` so that the tests in `TestCellPanel` are called.

```java
suite.addTest(new TestSuite(life.gui.TestCellPanel.class));
```

Save, compile, and run the tests. You’ll get a failure that lets you know that everything’s okay. The test runner is complaining that `TestCellPanel` doesn’t contain any
tests. It doesn’t, but the warning let’s you know that TestCellPanel was successfully added to the suite of tests being run. Next, you’ll add tests.

**Tests for Cell’s GUI**

What do you want to test for the component that provides the GUI for a Cell object? Suppose you use a JPanel for this component. What do you need to test? If you set the background color using setBackground() you don’t really need to check that it’s been set. You shouldn’t need to check that the properties and methods inherited from JPanel have been correctly implemented.

You also don’t need to see the GUI to test it. You want the testing process to be as automated as possible. You don’t want to have to dismiss windows or click buttons and check for the results every time you run the test suite. This means that you don’t have to create some sort of JFrame that holds a BoardPanel that in turn holds a grid full of CellPanel objects. You just need to begin by creating a single CellPanel and by testing its functionality. At some point you will need to see the GUI to make sure that it looks the way you want it to. This isn’t a process that you can automate with testing and you want to have the tests in place first so that you can be sure that your adjustments to the look of the GUI aren’t breaking any tests.

Start, as you did for Cell, by testing that a newly created CellPanel object is off. The test might look like this.

```java
public void testNewCellPanelIsOff(){
    CellPanel cellPanel = new CellPanel();
    assertTrue(!cellPanel.isOn());
}
```

This won’t compile. Create a CellPanel class that extends JPanel and provide it with an isOn() method that returns a boolean.

```java
package life.gui;
import javax.swing.JPanel;
public class CellPanel extends JPanel {
    protected boolean isOn() {
        return false;
    }
}
```

The code now compiles and the tests pass. For a next test, let’s create a CellPanel and then turn on the associated Cell and make sure the CellPanel reflects the change. Refactor your test code by introducing a setUp() method along with the new test.AnAliveCellPanelIsOn() method. The new TestCellPanel class looks like this.

```java
package life.gui;
import junit.framework.TestCase;
public class TestCellPanel extends TestCase {
    public void testNewCellPanelIsOff(){
        CellPanel cellPanel = new CellPanel();
        assertTrue(!cellPanel.isOn());
    }
}
```

This won’t compile. Create a CellPanel class that extends JPanel and provide it with an isOn() method that returns a boolean.
CellPanel cellPanel;

public TestCellPanel(String name)
{
    super(name);
}

protected void setUp()
{
    cellPanel = new CellPanel();
}

public void testNewCellPanelIsOff()
{
    assertTrue(!cellPanel.isOn());
}

public void testAnAliveCellPanelIsOn()
{
    cellPanel.getCell().setAlive(true);
    assertTrue(cellPanel.isOn());
}

This won’t compile. You have to write a getCell() method. To pass the tests you’ll have to fix the isOn() method. Here’s one possible solution. Note that you’ve had to add a constructor along with an instance variable to hold a handle to the Cell object. If you haven’t worked much with packages, it may not have occurred to you that you need to add an import statement to import life.model.Cell as it isn’t visible from life.gui.CellPanel. The changes to CellPanel.java are as follows.

```java
package life.gui;
import life.model.Cell;
import javax.swing.JPanel;
public class CellPanel extends JPanel {
    private final Cell cell;
    public CellPanel()
    {
        cell = new Cell();
    }
    protected boolean isOn()
    {
        if (cell.isAlive()) return true;
        else return false;
    }
    protected Cell getCell()
    {
        return cell;
    }
}

This still won’t compile. One solution is to change the access of the isAlive() and setAlive() methods in Cell to public because they are being called from outside of the life.model package.
Testing Communication from Cell to CellPanel

This still isn’t setting any visible properties in the CellPanel. You can amend the tests to require that the CellPanel changes colors.

```java
public void testNewCellPanelIsOff(){
    assertTrue(!cellPanel.isOn());
    assertEquals(cellPanel.getBackground(),cellPanel.DEAD_COLOR);
}

public void testAnAliveCellPanelIsOn(){
    cellPanel.getCell().setAlive(true);
    assertTrue(cellPanel.isOn());
    assertEquals(cellPanel.getBackground(),cellPanel.LIVE_COLOR);
}
```

As you may have come to expect, this code requires many changes in `CellPanel`.

```java
package life.gui;
import life.model.Cell;
import javax.swing.JPanel;
import java.awt.Color;
public class CellPanel extends JPanel {
    private final Cell cell;
    private boolean on;
    public static final Color DEAD_COLOR = Color.white;
    public static final Color LIVE_COLOR = Color.black;
    public CellPanel(){
        cell = new Cell();
    }
    protected boolean isOn() {
        setOn(cell.isAlive());
        return on;
    }
    protected Cell getCell(){
        return cell;
    }
    public void setOn(boolean on){
        this.on = on;
        if (on) setBackground(LIVE_COLOR);
        else setBackground(DEAD_COLOR);
    }
}
```

The tests all pass, but once again this solution doesn’t feel satisfying. The `isOn()` method shouldn’t have to poll cell. The cell object should notify the corresponding
CellPanel of any changes. You may recognize this as the Observer design pattern. eX-
treme Programming practitioners use design patterns, UML, or anything else that hap-
pens to solve the problem without introducing unnecessary overhead. When it is
appropriate to introduce a design pattern that simplifies your task, go ahead and use it.
You may also be coding along and just recognize that you’re in the middle of refactor-
ing toward a design pattern. Renaming classes and methods to communicate this fact
may help those reading your code. You can find more information on this topic of
refactoring toward design patterns on Joshua Kerievsky’s web site http://industriallog-
ic.com. Surprisingly few changes have to be made to the existing code to make this
work. In the CellPanel class change the body of the constructor to this.

```java
    cell = new Cell( this );
```

You are giving cell a way to communicate back with this CellPanel. Second re-
move the line `setOn (cell.isAlive());` from the `isOn()` method. There are also a few
changes that need to be made in Cell. You need to add two constructors, a CellPanel
variable, and change `setAlive()` to reflect the changes in the GUI. Here are the
changes to Cell.

```java
package life.model;
import life.gui.CellPanel;
public class Cell {
    private boolean alive;
    private CellPanel cellPanel;
    private int numberOfLiveNeighbors;
    private boolean changeNeeded;
    public Cell(){
        cellPanel = new CellPanel();
    }
    public Cell(CellPanel cellPanel){
        this.cellPanel = cellPanel;
    }
    public void setAlive(boolean alive){
        this.alive = alive;
        cellPanel.setOn(alive);
    }
    //rest of class omitted as nothing changes
}
```

The code compiles, the tests run.

**Testing Communication from CellPanel to Cell**

Suppose we have a story that allows the user to click on a cell in the GUI to toggle its
state. CellPanel is also used to send messages to Cell. You want to make sure that
when the user clicks on a CellPanel the change is reflected in the corresponding Cell.
This example is used to demonstrate how you might write a unit test for validating the
result of clicking on a particular CellPanel. Add the following tests to TestCellPanel.java:

```java
public void testClickOnDeadCellTurnsItAlive(){
    assertTrue(!cellPanel.isOn());
    cellPanel.dispatchEvent(
        new MouseEvent(cellPanel,MouseEvent.MOUSE_CLICKED,
                        0,MouseEvent.BUTTON1_MASK,0,0,1,false));
    assertTrue(cellPanel.isOn());
}

public void testClickOnLiveCellTurnsItDead(){
    cellPanel.getCell().setAlive(true);
    assertTrue(cellPanel.isOn());
    cellPanel.dispatchEvent(
        new MouseEvent(cellPanel,MouseEvent.MOUSE_CLICKED,
                        0,MouseEvent.BUTTON1_MASK,0,0,1,false));
    assertTrue(!cellPanel.isOn());
}
```

You’ll also need to add the following import statement.

```java
import java.awt.event.MouseEvent;
```

In the first test you verify that the CellPanel object is not on. You then send a mouse clicked event to the CellPanel object and check that the result is that the CellPanel object is now on. The mouse click has to result in the cell object being updated, which in turn results in the CellPanel object being updated. This is the sequence as described in the Gang of Four description of the Observer pattern. (Note that the Design Patterns book written by Gamma, Helm, Johnson, and Vlissides is commonly referred to as the “Gang of Four” book.) As you might expect, the code compiles but nothing happens. You need to add a MouseListener of some sort to CellPanel. Here are the changes to CellPanel.java.

```java
package life.gui;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
// unchanged code omitted ...
public CellPanel(){
    cell = new Cell(this);
    this.addMouseListener(new ClickHandler());
}
// more code omitted ...
public class ClickHandler extends MouseAdapter{
    public void mouseClicked(MouseEvent e){
        cell.setAlive(!cell.isAlive());
    }
}
```
If you were familiar with this pattern, you may have been tempted to dive right in and start writing code. Write the tests first, take small steps, and your code will be more robust.

A LOOK BACK

It is difficult to test first all of the time. Even devoted XPers will try to just knock some code out once in a while. For the most part, they end up regretting it. Ron Jeffries is kind enough to own up to this publicly in *Extreme Programming Installed*, but everyone does it.

What happens when you notice yourself coding without tests is important. It’s like quietly meditating while trying not to let your mind wander. At some point your mind wanders. The strategy is to notice your mind wandering and return to meditating. What often happens is that you will berate yourself for letting your mind wander and now you are on a path further from your goal. When you notice yourself not testing, return to testing. Programming as part of a vigilant pair will help keep you from wandering. A partner need not be a testing master to observe that you’ve started coding without tests and to suggest that you form your test before proceeding.

Do not test everything. Only test code that can break. In the running example you wrote code that ended up testing accessors. This was because the methods existed before they became accessors for a variable. If you had started with a variable xxx and methods getXxx() and setXxx() there would be little need to write tests to make certain the accessors were working properly.

The key to test-first programming is to take very small steps where nothing could go wrong. Fix what does go wrong. Move on. If you’ve followed the example in this chapter, you’ve already begun to feel good when the tests all pass. Maybe you’ve even come to see compiler messages and unit test messages as helping you fix your code.

EXERCISES

1. Just before moving on to test the GUI it was suggested that you change the return type of needsToChange() to void and introduce an accessor method getChangeNeeded(). This would require refactoring the tests. It’s probably a bit cleaner because needsToChange() would then do what its name suggests and getChangeNeeded() would be used to find the status of the result. Complete that refactoring.

2. You’ve seen tests that verify that live or dead cell that gain between one and four live neighbors behave properly. Add a test to show that a live cell with two live neighbors dies when one of its neighbors dies. Create a method called mourNewDepartedNeighbor().

3. Each cell should have eight neighbors that it can update when it changes state. Add this facility. Create the tests first.