

12

Unit Testing

Self-review Questions

12.1 *Why is testing not just necessary but fundamental to good programming practice?*

Code must work correctly in order to deliver the right result. Even a single error can cause a program to fail, either by crashing immediately or, worse, by appearing to work but producing invalid results that go unnoticed. When run, a typical program will move through many millions of states, each of which needs to be valid. The sheer number of states a program can enter, combined with the range of potential errors that can be made in the program code, makes the task of checking every state effectively impossible.

A number of tools and techniques help the programmer produce correct code, such as the type checking done by the compiler and the programming language specification, but testing is the technique of primary importance in producing correct code. Mathematical and proof techniques can be of use in checking code but there is no practical mathematical approach to proving that an arbitrary program is correct (indeed, the Halting Problem shows us that there is no general purpose way of even proving that a program will terminate, see http://en.wikipedia.org/wiki/Halting_problem).

The aim of testing is to find errors by running sections of code and observing the results. While some tests are used to simply confirm that the right result is computed, the majority of tests are focussed on running the code in ways most likely to find errors. Testing is the one general purpose, and practical, way that a program can be shown to work correctly.

12.2 *What is a unit in unit testing?*

Unit testing is based on calling methods and checking the result. Hence, a unit is a collection of related methods that can be tested within the same test class using the same fixtures declared in the test class.

A unit can be a complete class from the application being developed, or it can be some subset of the methods declared within a class that deal with a related piece of functionality. It is also possible to form a unit from methods from several classes, providing those methods are dealing with the same unit of behaviour and all use the same fixtures. An interface, being a collection of related methods, is also a potential unit.

Typically when a class is first being developed, a test class is created to test the methods as they are written. When the test methods no longer all use the same fixtures, or an interface is introduced, the test class can be split up into several test classes so that the tests in each new test class do not use the same fixtures. This has the effect of changing the unit from being a class to a subset of the methods in a class. It may also be a good idea to subdivide a test class if it simply

gets too long. As the development of the application proceeds, the units will be progressively refined to match the current design.

Beginners often assume that you must have one test class for each application class, and that this is the only organisation of test class. Use refactoring to move away from this position as a program is developed.

12.3 *What is a Red test?*

A Red test is one that has found an error when an assertion has failed or an unexpected exception gets thrown.

This can be described as a test that has succeeded, as the test has served its purpose of locating an error. Often, however, programmers talk about a red test as a failed test, meaning failed in the sense that an assertion has failed.

A Red test should be addressed immediately in order to fix the problem found. Ideally, there should never be more than one Red test at any one time.

12.4 *What is a Green test?*

A Green test is one that has completed without any assertions failing or any unexpected exceptions being thrown.

12.5 *What is an assertion?*

An assertion asserts, or checks, that the value of a boolean expression is true. If the value is false an error is raised. The boolean expression represents some property of the code being tested. For example, `assertEquals` will check that the value of two expressions is the same, so that if a method call is meant to return the value 10 `assertEquals` will compare the expected value with the actual value returned.

12.6 *Why must tests be automated?*

Testing manually (that is by having a person run each test step-by-step), is tedious, slow, error prone and very, very boring. Tests are likely to be chosen arbitrarily, will not be re-run and only a few tests will actually be run.

In contrast, automated testing allows large numbers of tests to be run repeatedly at high speed. The computer does all the hard work of setting up the tests, performing the tests and checking the results, which is what computers are very good at. This makes it possible to create a comprehensive test suite and use it effectively.

12.7 *Why must tests be repeatable?*

A unit test should always be repeatable, meaning that it can be run whenever required under exactly the same conditions. The test can then be relied on to detect when the code being tested no longer meets the specification set by the test, throughout the lifetime of that code. This avoids arbitrary decisions being made about how to perform the test or under what conditions.

Unit tests should be run every time a change is made to code and it is recompiled. This immediately identifies any errors that were made in the last change with respect to the test specification and also gives confidence that all the code still works correctly. If an error in the code is found directly after it has been introduced then the programmer (i.e., you) is in a much better position to fix the error as all the details will be fresh. In addition, by keeping each change small (literally one line of code) the location of the error can be found quickly without having to resort to a tool like a debugger. Of course, this depends on a test being in place that will detect any new error that has been introduced. If in doubt, new tests should be added.

Repeated running of a test suite is sometimes referred to as regression testing, where regression means that an error has been introduced into previously working code. If the code "regresses" then the test suite can be used to locate the cause of the error via the test or tests that detect errors.

12.8 *What is a dummy object and why is it needed?*

A dummy object provides a fake implementation that can be substituted, or injected, into code when running a test. The dummy object allows the code to run without having to create real objects, which may be difficult to create and initialise when testing.

12.9 *What is TestNG?*

TestNG is a testing framework consisting of a set of annotations used to create test classes, infrastructure and a test runner to run tests and a reporting mechanism to record the results of running tests. TestNG is written in Java, requiring Java 5. Full details can be found at the TestNG website <http://testng.org>.

12.10 *How is a TestNG test run?*

By creating a `testng.xml` configuration file that describes the test suite and then running the framework and supplying it with the configuration file. From the command line this looks like:

```
java org.testng.TestNG testng.xml
```

(this also requires that your classpath is set correctly).

In practice it is more convenient to run TestNG either via an Ant build file or from within an IDE like Eclipse or IntelliJ IDEA. The TestNG download includes an Ant task that provides a comprehensive set of attributes for configuring the TestNG task and Ant will also manage issues like the setting up the classpath.

12.11 *What does a configuration method do?*

A configuration method can be called either before or after running tests. When called before a test it is used to initialise the test fixtures and any other data needed to run the test. When called after a test, a configuration method can be used to tidy-up or release any resources held by fixtures such as open files or network connections.

The `@Configuration` annotation is used to mark an instance method as a configuration method. There are a number of attributes that can be added to the annotation declaration to specify exactly when a configuration method is called. For example, before or after each test method, before and after a test class and so on. A full list of annotations is given in the TestNG documentation (see <http://testng.org/doc/documentation-main.html#annotations>).

12.12 *How is a test method annotated?*

A test method is annotated with the `@Test` annotation. The annotation has various attributes that are used to control when and how the annotated test method is run. Further information can be found at: <http://testng.org/doc/documentation-main.html#annotations>

12.13 *List the steps of the test-driven development cycle.*

1. Think. Identify the next small piece of functionality to work on, moving the design forward one small step.
2. Write a test that specifies what the next small piece of code to be written should do.
 - The piece of code should be a small method, or statement(s) within a method.
3. Write enough code to let the new test compile and nothing else.
4. Run the test and see the new test detect an error. This gives the RED state.
5. Write just enough code to make the new test complete without error. Use the simplest code you can. This gives the GREEN state.
 - Keep the code really simple, even if it is messy code or temporarily creates duplication.
6. Refactor as necessary (not needed on every iteration).

- All tests must run unchanged before and after refactoring.
 - Clean up and restructure code.
 - Remove compromises and duplication made to get test passing initially.
 - An important aspect of TDD is to maintain design and code quality.
 - If a test must be changed, go to step 1 and treat the change as the next iteration.
7. Repeat for next small step.

12.14 Explain refactoring and give an example.

Refactoring is the process of altering the internal structure of code without changing its external behaviour, with the goal of cleaning-up and improving the detailed design of the code. The external behaviour means that the same method calls to invoke the code being refactored must work after the refactoring. An individual refactoring is typically straightforward, such as renaming a variable or method, although it may require a number of changes throughout sections of the code. A sequence of refactorings can make substantial changes to code, while preserving the external interface in terms of the public classes and methods. Refactoring is frequently used to remove code duplication.

Refactoring is an integral part of the Test Driven Development process. After each iteration potential refactorings should be considered and applied if they will result in better quality code. An absolutely essential requirement of refactoring is that all tests must run without failure before any refactoring is performed *and* must still run without error after the refactoring is done.

Most integrated development environments, including Eclipse, NetBeans and IDEA, include extensive support for refactoring. Automated tools are the best, quickest and most reliable way of performing refactoring and are a good argument for using one of the development environments.

Each refactoring is given a simple name allowing it to be referred to easily. As a result a common vocabulary or language of refactorings has developed and is in wide use amongst developers. The definitive work on refactoring is: "Refactoring: Improving the Design of Existing Code", by Martin Fowler with Kent Beck, John Brant, William Opdyke, and Don Roberts (June 1999), Addison-Wesley, ISBN 0201485672. This book forms a refactoring catalog, and the names of many of the refactorings are now in common use.

Example refactorings include:

- Extract Method — remove one or more lines of code from a method into a new method, replacing the code in the original method with a method call, passing parameters as necessary. This is a standard technique for reducing the size and complexity of methods.

- Extract Class — simplify a class by removing one or more instance variables and methods into a new class.
- Pull Up Method — move a duplicated method from two or more subclasses into a superclass, so only one version of the method exists.
- Rename Method (or Variable or Class or Interface or Package) — rename a method so the name better describes the purpose of the method. Choosing a good name for a method can be hard and it can take time to identify what the best name is. It is always better to rename a method than to stick with the original but inappropriate name. The automated refactoring tools give the major advantage that they will automatically rename a method throughout the source code of a program, which can save a lot of time and avoid mistakes in finding all the places the name is used.

A full catalog of common refactorings can be found at Martin Fowler's Refactoring website: <http://www.refactoring.com/>.

Also see the Wikipedia article on refactoring at <http://en.wikipedia.org/wiki/Refactoring> for more information.

12.15 *Why is duplication bad?*

Duplication is a frequent source of errors and a symptom of poor design. Both design and code can be duplicated, leading to the same structures being declared multiple times within a program.

Duplicated code is often created via a copy-and-paste approach, where existing code is simply copied somewhere else in a program as a quick coding solution. When an error is discovered in one copy of the code, the programmer has to find and edit all the other copies. In practice this is error prone, as it is easy to miss one or more copies, and the programmer may not know or have forgotten that the other copies exist.

Duplication does not require identical sections of code or parts of a design. Similar elements can often be abstracted into a single more general version. One of the core goals of refactoring and the Test Driven Development process is to identify and remove duplication.

Programming Exercises

12.1 *Use the test-driven approach to develop a method for calculating the n th Fibonacci number. Remember to take very small steps, one test at a time, and always write the simplest code that will pass the tests.*

The Fibonacci numbers are a sequence of positive numbers where each number is the sum of the two preceding numbers. The first two numbers are zero and one, giving the sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, and so on.

The sequence can be computed recursively using:

$$fibonacci(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & \text{if } n > 1 \end{cases}$$

Developing a Fibonacci method starts by writing the first test:

```
@Test
public void fibonacci_zero () {
    assertEquals ( fibonacci( 0 ), 0 );
}
```

This simply asserts that `fibonacci(0)` is 0 as defined in the formula above. The decision has been made to name the method `fibonacci` and make it a static method. TestNG has, of course, been used to write the test code. In order to make this compile, the simplest version of `fibonacci` is written:

```
public static int fibonacci ( final int n ) {
    return 0 ;
}
```

The test now runs and no errors are found, so we have Green. As the first step is very straightforward, we have skipped the need to see the first test fail (i.e., Red). Next a test for `fibonacci(1)` is added, which does result in Red:

```
@Test
public void fibonacci_one () {
    assertEquals ( fibonacci( 1 ), 1 );
}
```

The simplest way to get the new test to pass is to add an if statement to the `fibonacci` method:

```
public static int fibonacci ( final int n ) {
    if ( n == 0 ) { return 0 ; }
    else { return 1 ; }
}
```

The second test passes and so does a third test for `fibonacci(2)` as that happens to return 1 as well. Writing the fourth test for `fibonacci(3)` is more interesting as it requires the `fibonacci` method to be modified. Keeping things simple, as always, gives:

```
public static int fibonacci ( final int n ) {
    if ( n == 0 ) { return 0 ; }
    else {
        if ( n == 1 || n == 2 ) { return 1 ; }
        else { return 2 ; }
    }
}
```

Noting that the definition of `fibonacci` defines $fibonacci(3) = fibonacci(2) + fibonacci(1)$, the method can be refactored to:

```
public static int fibonacci ( final int n ) {
    if ( n == 0 ) { return 0 ; }
    else {
        if ( n == 1 || n == 2 ) { return 1 ; }
        else { return fibonacci ( 2 ) + fibonacci ( 1 ) ; }
    }
}
```

This has introduced recursion into the method. A further refactoring can eliminate the literal values passed as parameters in the recursive calls, replacing them with values computed using the parameter n . This makes use of the fact that $fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)$.

```
public static int fibonacci ( final int n ) {
    if ( n == 0 ) { return 0 ; }
    else {
        if ( n == 1 || n == 2 ) { return 1 ; }
        else { return fibonacci ( n - 1 ) + fibonacci ( n - 2 ) ; }
    }
}
```

With this in place tests for `fibonacci(4)` and `fibonacci(5)` can be added, resulting in Green and building confidence that the method works:

```
@Test
public void fibonacci_four () {
    assertEquals ( fibonacci( 4 ) , 0 ) ;
}

@Test
public void fibonacci_five () {
    assertEquals ( fibonacci( 5 ) , 0 ) ;
}
```

At this point it is becoming very obvious that there is some serious duplication occurring in the test methods. They differ only in the value passed to the `fibonacci` method and the expected result. We could do all the testing via a single parameterised test method, where the value passed to the `fibonacci` call and the expected result are the parameters.

It turns out that not only can a TestNG test method be parameterised but TestNG also supports a mechanism called a `DataProvider` that allows a parameterised test method to be repeatedly called using different parameters each time (see <http://testng.org/doc/documentation-main.html> for detailed information). This allows all the tests so far, and additional ones, to be run using the following code:

```
@DataProvider ( name = "sequence" )
public Object[][] createData () {
```

```

return new Object[][] {
    { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 },
    { 5, 5 }, { 6, 8 }, { 7, 13 }, { 8, 21 }, { 9, 34 },
    { 10, 55 }, { 11, 89 }, { 18, 2584 }, { 21, 10946 }
};
}

@Test( dataProvider = "sequence" )
public void fibonacci_n ( final int n, final int expected ) {
    assertEquals ( fibonacci ( n ), expected );
}

```

The method `createData` is annotated with `@DataProvider (name = "sequence")` to denote that it returns a data structure containing test data. The annotation parameter `name` allows the `DataProvider` to be named so it can be referred to by test methods.

The test method `fibonacci_n` is annotated with `@Test` as normal but a `dataProvider` parameter is used to name the `DataProvider` that should be used when the test method is run. Given that the test method takes two parameters, the `DataProvider` provides two parameter values for each call of the test method. This is why the `DataProvider` method returns an array of pairs (each pair actually being an array with 2 items). When the tests are run the test method will be automatically be called for each pair of values declared by the `DataProvider`.

The tests are now Green. With the `DataProvider` and `fibonacci_n` methods in place, all the other test methods can be deleted as they are now redundant. They have, however, served their purpose in getting development to this point.

Next it is time to consider any other refactorings that might take place. The test code does not need any further work now but the `fibonacci` method can be improved. The first step is to note that `fibonacci(2)` does not need to be treated as a special case as it can be computed recursively. This gives:

```

public static int fibonacci ( final int n ) {
    if ( n == 0 ) { return 0 ; }
    else {
        if ( n == 1 ) { return 1 ; }
        else { return fibonacci ( n - 1 ) + fibonacci ( n - 2 ) ; }
    }
}

```

A little better but we still have a nested `if` statement, which looks clumsy. Noting that if `n < 2` the method can just return `n`, `fibonacci` can be further simplified to:

```

public static int fibonacci ( final int n ) {
    if ( n < 2 ) { return n ; }
    else { return fibonacci ( n - 1 ) + fibonacci ( n - 2 ) ; }
}

```

The end result of all this is the following code:

```

import static org.testng.Assert.assertEquals ;
import org.testng.annotations.DataProvider ;
import org.testng.annotations.Test ;

public class Fibonacci {
    public static int fibonacci ( final int n ) {
        if ( n < 2 ) { return n ; }
        else { return fibonacci ( n - 1 ) + fibonacci ( n - 2 ) ; }
    }
    @DataProvider ( name = "sequence" )
    public Object[][] createData () {
        return new Object[][] {
            { 0 , 0 } , { 1 , 1 } , { 2 , 1 } , { 3 , 2 } , { 4 , 3 } ,
            { 5 , 5 } , { 6 , 8 } , { 7 , 13 } , { 8 , 21 } , { 9 , 34 } ,
            { 10 , 55 } , { 11 , 89 } , { 18 , 2584 } , { 21 , 10946 }
        } ;
    }
    @Test ( dataProvider = "sequence" )
    public void fibonacci_n ( final int n , final int expected ) {
        assertEquals ( fibonacci ( n ) , expected ) ;
    }
}

```

For convenience the `fibonacci` method has been declared in the test class while being developed.

In summary, this answer has shown a step-by-step test-driven approach to developing a `fibonacci` method. It might be argued that in a realistic development setting many of the steps could have left out but that misses the point of deliberately taking small steps, all the time writing tests and thinking carefully about design and refactoring. We are left with a method that has been tested and that we have good confidence in.

Or do we? Doing some more research on Fibonacci numbers reveals some further issues. First there is the question of whether `0` is actually a Fibonacci number at all. Then there is the problem that Fibonacci numbers quickly get too large to be represented by values of type `int` or `long`. Also, our `fibonacci` method can be called with negative values, always returning zero. Should it throw an exception instead? To address these issues we need to go back and specify the method rather more carefully. We leave that as a further exercise for you.

12.2 *Could the method or variable names in the extended class `Person` be improved to make the code even easier to understand? Suggest some new names and see if they work better.*

12.3 *Make the `calculateAge` method in the extended class `Person` public and provide a set of tests for it in class `PersonTest`. Are your new tests any different from those for the method `getAgeOn`? Was it worth making `calculateAge` public?*

12.4 *Follow the test-driven approach to write a basic `DateValidator` class for use with the `Person` class.*

- 12.5** *Modify the Person class `getAgeOn` method to perform date validation. Extend your validator class and interface as necessary, using the test-driven approach obviously!*
- 12.6** *Use test-driven development to add a `getEmailAddress` method to class Person. Email address must be validated to check they have the correct structure.*
- 12.7** *Evolve the classes so as to remove all the String parameters from Person.*

Challenges

- 12.1** *Use test-driven development to write a program using class Person that allows the user to:*
- *Enter a list of famous historical people (name and birth date).*
 - *elect a person from the list and print their age on a selected date.*