Module 3 (Part II)

Writing Unit Tests

Index

- Writing tests with Assertions
- Defining and using Custom Assertions
- Single condition tests
- Testing for expected errors
- Abstract test

❖ Introduction

- The previous sections present a simple unit test framework and the fundamentals of xUnit
- The unit test framework's architecture is important to understand
- Most of your time should be spent writing unit tests, implementing production code to make the tests pass, or refactoring
- This section includes examples of common patterns used when writing unit tests, as well as related tips on unit test development

❖ Writing tests with Assertions

- The code examples shown so far use ***plain asserts***
- These are the most generic type of test assertion, which take a Boolean condition that must evaluate to TRUE for the test to succeed
- A plain assert, the unit test for the Library method removeBook(), is shown in Example 4-1

Example 4-1. Test method testRemoveBook() using a plain assert

LibraryTest.java

```
public void testRemoveBook( ) {
    library.removeBook( "Dune" );
    Book book = library.getBook( "Dune" );
    assertTrue( book == null );
}
```

- Although the line of code where the failure occurred is shown, the output does not describe the specific cause of the failure

```
There was 1 failure:
1) testRemoveBook(LibraryTest)junit.framework.AssertionFailedError
```

- It often is helpful to add an informative message to the assertion
- The xUnits generally have two versions of every assert method, one of which takes a message parameter describing the assert
- Example 4-2 shows the test method using an assert with a message

Example 4-2. Test method using an assert with a message

LibraryTest.java

```
public void testRemoveBook( ) {
   library.removeBook( "Dune" );
   Book book = library.getBook( "Dune" );
   assertTrue( "book is not removed", book == null );
}
```

With the additional message, the rest results provide better information about the cause of the test failure:

```
1) testRemoveBook(LibraryTest)junit.framework.AssertionFailedError: book is not
removed
```

- Although all assert conditions ultimately must evaluate to a Boolean result of TRUE or FALSE, it can be tedious to constantly reduce every expression to this form
- The xUnits offer a variety of assert functions to help
- Examples of several of the assert methods from JUnit are as follows:

```
assertFalse( book == null );
assertFalse( "book is null", book == null );
assertNull( book );
assertNull( "book is not null", book );
assertNotNull( book );
assertNotNull( "book is null", book );
assertEquals( "Solaris", book.title );
assertEquals( "unexpected book title", "Solaris", book.title );
```

- These assert methods all have variants that take a message parameter to describe the failure, as shown above
- The assertEquals() method has variants that take different data types as arguments

❖ Defining and using Custom Assertions

- The basic assert methods cover only a few common cases
- It's often useful to extend them to cover additional test conditions and data types
- Custom assert methods save test coding effort and make the test code more readable
- So far, the Library tests check a Book's title attribute to verify the expected Book object, as shown in Example 4-3 in the test method testGetBooks()

Example 4-3. Test comparing two Books using their title attributes

LibraryTest.java

```
public void testGetBooks( ) {
    Book book = library.getBook( "Dune" );
    assertTrue( book.getTitle( ).equals( "Dune" ) );
    book = library.getBook( "Solaris" );
    assertTrue( book.getTitle( ).equals( "Solaris" ) );
}
```

- It's clearly useful to have an assert method that compares an expected Book to the actual Book, checking all of the attributes
- This new assert method is easy to implement by building on the generic assertTrue() method, as shown in Example 4-4

Example 4-4. Custom assert method to compare Books

BookTest.java

```
public class BookTest extends TestCase {

   public static void assertEquals( Book expected, Book actual ) {
      assertTrue(expected.getTitle().equals( actual.getTitle( ) )
            && expected.getAuthor().equals( actual.getAuthor( ) ));
   }
}
```

- The assert method assertEquals() takes expected and actual Book objects to compare
- It succeeds if the title and author attributes of the two Books are equal
- Example 4-5 shows how it is used

Example 4-5. Using the custom assert method

LibraryTest.java

```
public class LibraryTest extends TestCase {

    private Library library;
    private Book book1, book2;

    public void setUp( ) {
        library = new Library( );
        book1 = new Book("Dune", "Frank Herbert");
        book2 = new Book("Solaris", "Stanislaw Lem");
        library.addBook( book1 );
        library.addBook( book2 );
    }

    public void testGetBooks( ) {
        Book book = library.getBook( "Dune" );
        BookTest.assertEquals( book1, book );
        book = library.getBook( "Solaris" );
        BookTest.assertEquals( book2, book );
    }
}
```

❖ Single condition tests

- The idea is that a test method should only test one behavior
- If there is more than one assert condition, multiple things are being tested
- When there is more than one condition to test, then a **test fixture** should be set up, and each condition placed in a separate test method

Example 4-6. Poorly written unit test that tests multiple behaviors

LibraryTest.java

```
public void testLookupBooksByAuthor( ) {
    // Add two books by same author
    Book book3 = new Book( "Cosmos", "Carl Sagan" );
    Book book4 = new Book( "Contact", "Carl Sagan" );
    library.addBook( book3 );
    library.addBook( book4 );
    // Look up books by title and author
    Book book = library.getBook( "Cosmos", "Carl Sagan" );
    BookTest.assertEquals( book3, book );
    book = library.getBook( "Contact", "Carl Sagan" );
    BookTest.assertEquals( book4, book );
    // Look up both books by author
    Vector books = library.getBooks( "Carl Sagan" );
    assertEquals( "two books not found", 2, books.size( ) );
    book = (Book)books.elementAt(0);
    BookTest.assertEquals( book3, book );
    book = (Book)books.elementAt(1);
    BookTest.assertEquals( book4, book );
}
```

- It tests two separate behaviors: getting a Book by author and title and getting multiple Books by the same author
- Looking up two books by two different methods means there are several results to test; thus, there are many asserts - five in all
- The complexity of the changes increases the chance that a coding mistake will be made
- If the Book lookup by title and author fails, it has to be fixed before the test that gets multiple Books is run
- In other words, the tests are coupled so that failure of one may affect the success of the others

- When the number of asserts in a test method is excessive, change it into a **test fixture** with multiple test methods, each testing one behavior
- In Example 4-7, refactoring the test method makes it apparent that the two lookup methods are distinct behaviors and should be tested separately
- Example 4-7 shows LibraryTest with the two separate test methods, one for each behavior
- The code to add the two test Books is placed in the setUp() method
- The tests are isolated and the code is simplified

Example 4-7. The previous test method refactored into separate test methods

LibraryTest.java

```
public void setUp( ) {
    book3 = new Book( "Cosmos", "Carl Sagan" );
    book4 = new Book( "Contact", "Carl Sagan" );
    library.addBook( book3 );
    library.addBook( book4 );
}

public void testGetBookByTitleAndAuthor( ) {
    Book book = library.getBook( "Cosmos", "Carl Sagan" );
    BookTest.assertEquals( book3, book );
}
public void testGetBooksByAuthor( ) {
    Vector books = library.getBooks( "Carl Sagan" );
    assertEquals( "two books not found", 2, books.size( ) );
    Book book = (Book)books.elementAt(0);
    BookTest.assertEquals( book3, book );
    book = (Book)books.elementAt(1);
    BookTest.assertEquals( book4, book );
}
```

❖ Testing for expected errors

- It is important to test the error-handling behavior of production code in addition to its normal behavior
- Such tests generate an error and assert that the error is handled as expected
- In other words, an expected error produces a unit test success
- The canonical example of a unit test that checks expected error handling is one that tests whether an expected exception is thrown, as shown in Example 4-8

Example 4-8. Unit test for expected exception

LibraryTest.java

```
public void testRemoveNonexistentBook( ) {
    try {
        library.removeBook( "Nonexistent" );
        fail( "Expected exception not thrown" );
    } catch (Exception e) {}
}
```

- The expected error behavior is that an exception is thrown when the removeBook() method is called for a nonexistent Book
- If the exception is thrown, the unit test succeeds
- If it is not thrown, fail() is called
- The fail() method is another useful variation on the basic assert method
- It is equivalent to assertTrue(false), but it reads better

❖ Abstract test

- Just like regular classes, abstract classes and interfaces should have their own unit tests
- An AbstractTest contains an abstract factory method, which produces an instance of the object to test
- It also contains the test methods for the abstract class
- They resemble ordinary unit test methods, but test instances of the abstract class created by the factory method

- To test a concrete class that is descended from the abstract class, the unit test is subclassed from the AbstractTest
- Its factory method returns an instance of the concrete class
- When the concrete unit test is run, the AbstractTest is run as well
- So, the AbstractTest tests every concrete implementation of the abstract class
- Let’s create an AbstractTest for the interface DBConnection
- We’ll add the method isOpen() to it, as shown in Example 4-18

Example 4-18. The interface DBConnection

DBConnection.java

```
public interface DBConnection {
   void connect( );
   void close( );
   boolean isOpen( );
   Book selectBook( String title, String author );
}
```

The AbstractTest should test the behavior of the interface to make sure that any concrete implementation of it is correct. Tests of the `isOpen( )` method should verify that it returns `TRUE` after `connect( )` is called, and `FALSE` after `close( )` is called. The AbstractTest class `AbstractDBConnectionTestCase`, shown in Example 4-19, provides these tests.

Example 4-19. The AbstractTest class AbstractDBConnectionTestCase

AbstractDBConnectionTestCase.java

```
import junit.framework.*;

public abstract class AbstractDBConnectionTestCase extends TestCase {

   public abstract DBConnection getConnection( );

   public void testIsOpen( ) {
      DBConnection connection = getConnection( );
      connection.connect( );
      assertTrue( connection.isOpen( ) );
   }

   public void testClose( ) {
      DBConnection connection = getConnection( );
      connection.connect( );
      connection.close( );
      assertTrue( !connection.isOpen( ) );
   }
}
```

- The AbstractTest specifies a factory method, getConnection()
- Concrete tests that descend from it will implement the factory method, allowing the test methods testIsOpen() and testClose() to test an instance of the concrete class
- Notice how these methods use getConnection() to get the DBConnection to test

- To see the AbstractTest run, we need to define a concrete class descended from DBConnection and a corresponding concrete unit test descended from AbstractDBConnectionTestCase
- The concrete class JDBCConnection is shown in Example 4-20

Example 4-20. The concrete class JDBCConnection

JDBCConnection.java

```
public class JDBCConnection implements DBConnection {

    private String connectString;
    private boolean open;

    public JDBCConnection( String connect ) {
        connectString = connect;
        open = false;
    }

    public void connect( ) { open = true; }
    public void close( ) { open = false; }
    public boolean isOpen( ) { return open; }
    public String getConnectString( ) { return connectString; }
    public Book selectBook( String title, String author ) {
        return null;
    }
}
```

- JDBCConnection is an initial version of an interface to a JDBC database engine
- It differs from the base DBConnection by its member connectString, which contains the URL of a JDBC database connection
- The unit test JDBCConnectionTest tests JDBCConnection
- It is derived from the AbstractTest
- It is shown in Example 4-21

Example 4-21. The concrete test JDBCConnectionTest

JDBCConnectionTest.java

```
public class JDBCConnectionTest extends AbstractDBConnectionTestCase {

   public DBConnection getConnection( ) {
      return new JDBCConnection( "jdbc:odbc:testdb" );
   }

   public void testConnectString( ) {
      JDBCConnection connection = (JDBCConnection)getConnection( );
      String connStr = connection.getConnectString( );
      assertTrue( connStr.equals("jdbc:odbc:testdb") );
   }
}
```

- JDBCConnectionTest implements the factory method getConnection() and one test method, testConnectString()
- When the test is instantiated and run, the two test methods in the parent AbstractTest also will be run to test instances of JDBCConnection
- This way, the AbstractTest verifies that the concrete subclass passes the tests of the parent interface