
Unit Testing with
JUnit

Unit Testing

- A unit test is a piece of code that performs a test on another piece of code in isolation.
- Each unit test is independent from each other.
- A unit test can be executed automatically, without user intervention

Why write unit tests?

- It ensures the code works as expected
  - at the time the code is written
  - after subsequent changes
- Helps refactoring
- Works as documentation of tests

Unit Testing Myths

- Writing Unit tests takes a lot of time.
  - With a little of practice, take the same time as testing manually
  - In the long run, saves you a lot of time, helping you detect bugs introduced later in your code

JUnit

- Is the most popular Unit Testing Framework for Java
- 2 Versions available
  - 3.x
  - **4.x** (Requires Java 5) ← use this version

Writing Test Cases

- Import JUnit packages

```
import org.junit.*;
import static org.junit.Assert.*;
```

- Each test method is annotated with @Test

```
@Test
public void testAddBook()

    // Write test code here...

}
```

Class to be tested

```
public interface BookLibrary {

    public boolean addBook(Book book);

    public boolean removeBook(Book book);

    public int getBookCount();

    public Book searchBook(String isbn);

    public List<Book> getSorted();

}
```

Test Method

```
@Test
public void testAddBook() {
    Book book = new Book("1-930110-99-5", "JUnit in Action", "Vincent Massol");

    /* Initially, the library is empty */
    assertEquals(0, library.getBookCount());

    /* Add the book */
    boolean res = library.addBook(book);

    /* The book was added */
    assertTrue(res);

    /* The library now contain one book */
    assertEquals(1, library.getBookCount());

    /* The library contains the book added */
    assertEquals(book, library.searchBook("1-930110-99-5"));
}
```

Other methods

- A method with @Before is called before each test.

```
/**
 * This method is executed before each test.
 */
@Before
public void setUp() throws Exception {
    library = new BookLibraryImpl();
}
```

- Use them to create/destroy objects shared among methods

Other methods

- A method with @After is called after each test.
- Methods with @BeforeClass and @AfterClass run once before and after all test cases.
- A method with @Ignore is not run.

Handling Exceptions

- The “expected” parameter is used when a test expects an exception

```
@Test(expected = ArithmeticException.class)
public void divisionWithException() {
    // divide by zero
    simpleMath.divide(1, 0);
}
```

- The test fails if no exception is thrown

AssertMethods

- Used to test conditions inside test methods
  - **assertTrue(...)**: Tests if the parameter is true
  - **assertFalse(...)**: Tests if the parameter is false
  - **assertEquals(...)**: Tests if the two parameters are equal (equals method)
  - **assertSame(...)**: Tests if the two parameters are the same (==)
  - **assertNotSame(...)**: Tests if the two parameters are not the same.
  - **assertNotNull(...)**: Test if the parameter is not null.
  - **assertNull(...)**: Tests if the parameter is null.
  - **fail()**: always fails.

Complete Test Case

```
import org.junit.*;
import static org.junit.Assert.*;

public class BookLibraryTest {

    private BookLibrary library;

    @Before
    public void setUp() throws Exception {
        library = new BookLibraryImpl();
    }

    @Test
    public void testAddBook() {
        Book book = new Book("1-930110-99-5", "JUnit in Action", "Vincent Massol");

        assertEquals(0, library.getBookCount()); //Initially, the library is empty

        boolean res = library.addBook(book); // Add the book
        assertTrue(res); // Test that the book was added
        assertEquals(1, library.getBookCount()); // The library now contain one book
        assertEquals(book, library.searchBook("1-930110-99-5")); //The library contains the book added
    }

}
```

Complete Test Case

```java
import org.junit.*;
import static org.junit.Assert.*;

public class BookLibraryTest {

    private BookLibrary library;

    @Before
    public void setUp() throws Exception {
        library = new BookLibraryImpl();
    }

    @Test
    public void testAddBook() {
        Book book = new Book("1-930110-99-5", "JUnit in Action", "Vincent Massol");

        assertEquals(0, library.getBookCount()); //Initially, the library is empty

        boolean res = library.addBook(book); // Add the book
        assertTrue(res); // Test that the book was added
        assertEquals(1, library.getBookCount()); // The library now contain one book
        assertEquals(book, library.searchBook("1-930110-99-5")); //The library contains the book added
    }

}
```

Is this test case enough to test the method?

Guidelines for writing tests

- Test for the main flow (the happy path)
- Tests for the main alternative flows
- Test for boundary conditions (such as null arguments, negative numbers, etc)

Some pitfalls

- Classes that call static methods are difficult to test.
  - Be careful with your implementation of the Singleton pattern!!
  - Use a factory.
- Entangled designs may be very difficult to test.