

EECS 4313

Software Engineering Testing



Topic 11:

Test Code Patterns

- How to design your test code

Zhen Ming (Jack) Jiang

Testing and Inheritance

- Should you retest inherited methods?
- Can you reuse superclass tests for inherited and overridden methods?
- To what extent should you exercise interaction among methods of all superclasses and of the subclass under test?

Inheritance

- In the early years people thought that inheritance will reduce the need for testing
 - Claim 1: “If we have a well-tested superclass, we can reuse its code (in subclasses, through inheritance) with confidence and without retesting inherited code”
 - Claim 2: “A good-quality test suite used for a superclass will also be sufficient for a subclass”
- Both claims are wrong.

Inheritance-related bugs

■ Missing Override

- A subclass omits to provide a specialized version of a superclass method
- Subclass objects will have to use the superclass version, which might not be appropriate
- E.g., method `equals` in `Object` tests for reference equality. In a given class, it might be right to override this behaviour

Inheritance-related bugs

- Direct access to superclass fields from the subclass code
 - Changes to the superclass implementation can create subclass bugs
 - Subclass bugs or side effects can cause failure in superclass methods
 - If a superclass is changed, all subclasses need to be tested
 - If a subclass is changed, superclass features used in the subclass must be retested

Testing of Inheritance

- Principle: inherited methods should be retested in the context of a subclass
- Example 1: if we change some method **m** in a superclass, we need to retest **m** inside all subclasses that inherit it

Example 2

```
class A {  
    int x; // invariant: x > 100  
    void m() { // correctness depends on  
                // the invariant } }
```

```
class B extends A {  
    void m2() { x = 1; } }
```

- If we add a new method **m2** that has a bug and breaks the invariant, method **m** is incorrect in the context of **B** even though it is correct in **A**
 - Therefore, **m** should be tested in **B**

Example 3

```
class A {  
void m() { ...; m2(); ... }  
void m2() { ... } }
```

```
class B extends A {  
    void m2() { ... } }
```

- If inside **B** we override a method from **A**, this indirectly affects other methods inherited from **A**
 - e.g., method **m** calls **B.m2**, not **A.m2**: so, we cannot be sure that **m** is correct anymore and we need to retest it inside **B**
- Test cases developed for a method **m** defined in class **A** are not necessarily sufficient for retesting **m** in subclasses of **A**
 - e.g., if **m** calls **m2** in **A** and then some subclass overrides **m2** we have a completely new interaction that may not be covered well by the old test cases for **m**
- Still it is essential to run all superclass tests on a subclass
 - Goal: check behavioral conformance of the subclass w.r.t. the superclass (LSP)

Inheritance-related bugs

- Square Peg in a Round Hole
 - Design Problem
 - A subclass is incorrectly located in a hierarchy
 - **Liskov Substitution Principle (LSP)**:
Functions that use references to base classes must be able to use objects of derived classes without knowing it.

An example

- Consider class Rectangle below

```
class Rectangle{
    public void setWidth(double w) {itsWidth=w;}
    public void setHeight(double h) {itsHeight=h;}
    public double getHeight() {return itsHeight;}
    public double getWidth() {return itsWidth;}

    private double itsWidth;
    private double itsHeight;
};
```

- Assume that the system containing **Rectangle** needs to deal with squares as well
- Since a **square** is a rectangle, it seems to make sense to have a new class Square that extends Rectangle
- That very “reasonable” design can cause some significant problems

Problems with this design

- Do not need both itsHeight and itsWidth
- setWidth and setHeight can bring a Square object to a corrupt state (when height is not equal to width)

One
solution

```
class Square{
    setWidth(double w){
        super.setWidth(w);
        super.setHeight(w);
    }
    // Similar for setHeight
}
```

Not really a solution

- Consider this client code

```
Rectangle r;  
...  
r.setWidth(5);  
r.setHeight(4);  
assert(r.getWidth() * r.getHeight() == 20);
```

- The problem is definitely not with the client code

What went wrong?

- The Liskov Substitution Principle was violated
 - If you are expecting a rectangle, you can not accept a square
- The overridden versions of `setWidth` and `setHeight` broke the post-conditions of their superclass versions
- Isn't a square a rectangle?
 - Yes, but not when it pertains to its behaviour

Effect of Inheritance on Testing?

- Does not reduce the volume of test cases
- Rather, number of interactions to be verified goes up at each level of the hierarchy

Polymorphic Server Test

- Consider all test cases that exercise polymorphic methods
- According to LSP, these should apply at every level of the inheritance hierarchy
- Expand each test case into a set of test cases, one for each polymorphic variation

An example

```
class TestAccount {
    Account a;

    @Before
    public void setUp(){
        a = new Account();
    }

    @Test
    public final void testDeposit(){
        a.deposit(100);
        assertTrue(a.getBalance()==100);
    }
}
```


An example

```
class TestSavingsAccount extends TestAccount{
    SavingsAccount sa;

    @Before
    public void setUp(){
        a = new SavingsAccount();
        sa = new SavingsAccount();}

    @Test
    public void testInterest(){
        sa.deposit(100);
        sa.applyInterest(0.01);
        assertEquals("The balance does not match", 101.0,
                    sa.getBalance(), 0);
    }
}
```

Testing abstract classes

- Abstract classes cannot be instantiated
- However, they define an interface and behaviour (contracts) that implementing classes will have to adhere to
- We would like to test abstract classes for **functional compliance**
 - Functional Compliance is a module's compliance with some documented or published functional specification

Functional vs. syntactic compliance

- The compiler can easily test that a class is syntactically compliant to an interface
 - All methods in the interface have to be implemented with the correct signature
- Tougher to test functional compliance
 - A class implementing the interface `java.util.List` may be implementing `get(int index)` or `isEmpty()` incorrectly
- Think LSP...

Abstract Test Pattern

- This pattern provides the following
 - A way to build a test suite that can be reused across descendants
 - A test suite that can be reused for future as-yet-unidentified descendants
 - Especially useful for writers of APIs.

An example

- Consider a statistics application that uses the Strategy design pattern

```
public interface StatPak
{
    public void reset();
    public void addValue(double x);
    public double getN();
    public double getMean();
    public double getStdDev();
}
```

Abstract Test Rule 1

- Write an abstract test class for every interface and abstract class
- An abstract test should have test cases that cannot be overridden
- It should also have an abstract Factory Method for creating instances of the class to be tested.

Example abstract test class

```
public abstract TestStatPak {
    private StatPak statPak;

    @Before
    public final setUp() throws Exception {
        statPak = createStatPak();
        assertNotNull(statPak);
    }

    // Factory Method. Every test class of a
    // concrete subclass K must override this
    // to return an instance of K
    public abstract StatPak createStatPak();
    //Continued in next slide..
}
```

Example abstract test class (continued)

```
@Test
public final void testMean() {
    statPak.addValue(2.0);
    statPak.addValue(3.0);
    statPak.addValue(4.0);
    statPak.addValue(2.0);
    statPak.addValue(4.0);
    assertEquals("Mean value of test data should be 3.0",
                 3.0, statPak.getMean());
}
```

```
@Test
public final void testStdDev() { ... }
```


Abstract Test Rule 2

- Write a concrete test class for every implementation of the interface (or abstract class)
- The concrete test class should extend the abstract test class and implement the factory method

Example concrete test class

```
public class TestSuperSlowStatPak
    extends TestStatPak {

    public StatPak createStatPak()
    {
        return new SuperSlowStatPak();
    }
}
```

Only a few lines of code and all the test cases for the interface have been reused

Guideline

- Tests defining the functionality of the interface belong in the abstract test class
- Tests specific to an implementation belong in a concrete test class
 - We can add more test cases to **TestSuperSlowStatPak** that are specific to its implementation

Crash Test Dummy

- Most software systems contain a large amount of error handling code
- Sometimes, it is quite hard to create the situation that will cause the error
 - Example: Error creating a file because the file system is full
- Solution: Fake it!

```
import java.io.File;
import java.io.IOException;

class FullFile extends File {

    public FullFile(String path) {
        super(path);
    }

    public boolean createNewFile() throws IOException {
        throw new IOException();
    }
}
```

```
public void testFileSystemFull() {  
    File f = new FullFile("foo");  
    try {  
        saveAs(f);  
        fail();  
    }  
    catch (IOException e)  
    {}  
}
```

How do we ensure the file system is full,
so that it will throw the IOException?

```
public void testFileSystemFull() {
    File f = new FullFile("foo") {
        public boolean createNewFile() throws IOException {
            throw new IOException();
        }
    };
    try {
        saveAs(f);
        fail();
    }
    catch (IOException e)
    {}
}
```

It is much better to use the Mocking framework

Method Stubbing using Mockito

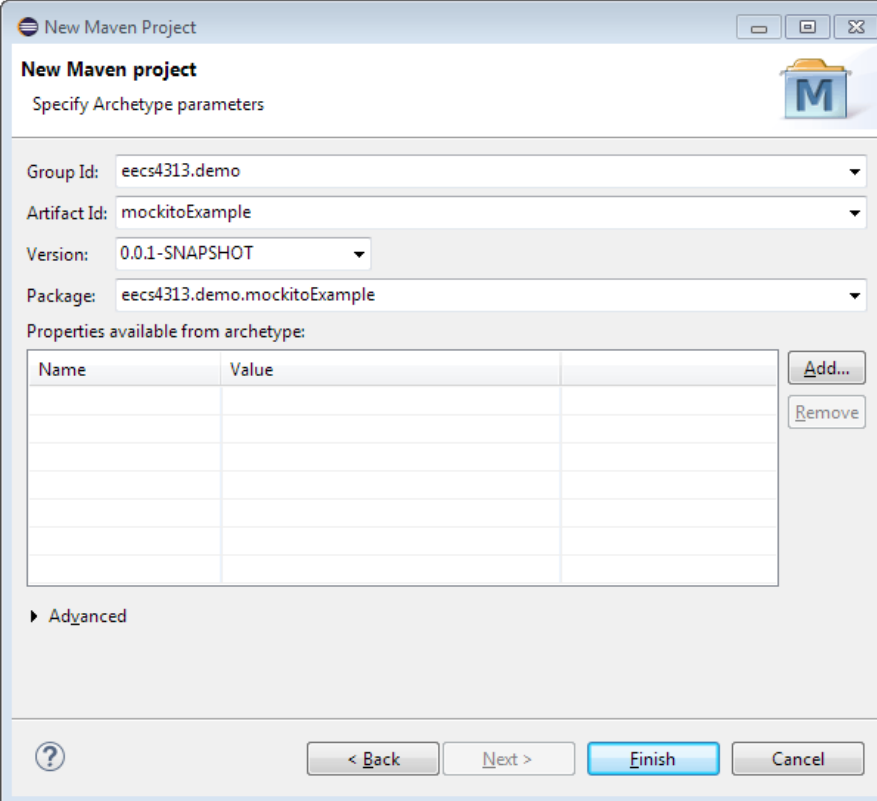
The Mockito materials are adapted from:

<https://dzone.com/articles/getting-started-mocking-java>

<http://examples.javacodegeeks.com/core-java/mockito/junit-mockito-example/>

Create Maven Project

- Create a new Maven project:
 - File -> new project -> Maven project and enter information similar as below



New Maven Project

New Maven project

Specify Archetype parameters

Group Id: eecs4313.demo

Artifact Id: mockitoExample

Version: 0.0.1-SNAPSHOT

Package: eecs4313.demo.mockitoExample

Properties available from archetype:

Name	Value

Advanced

Buttons: ? < Back Next > Finish Cancel

Create a Book class under the src folder

```
package eecs4313.demo.mockitoExample;
```

```
import java.util.List;
```

```
/** Model class for the book details.*/
```

```
public class Book {
```

```
    private String isbn;
```

```
    private String title;
```

```
    private List<String> authors;
```

```
    private String publication;
```

```
    private Integer yearOfPublication;
```

```
    private Integer numberOfPages;
```

```
    private String image;
```

```
    public Book(String isbn,
```

```
                String title,
```

```
                List<String> authors,
```

```
                String publication,
```

```
                Integer yearOfPublication,
```

```
                Integer numberOfPages,
```

```
                String image) {
```

```
        this.isbn = isbn;
```

```
        this.title = title;
```

```
        this.authors = authors;
```

```
        this.publication = publication;
```

```
        this.yearOfPublication = yearOfPublication;
```

```
        this.numberOfPages = numberOfPages;
```

```
        this.image = image;
```

```
    }
```

```
    public String getIsbn() {
```

```
        return isbn;
```

```
    }
```

```
    public String getTitle() {
```

```
        return title;
```

```
    }
```

```
    public List<String> getAuthors() {
```

```
        return authors;
```

```
    }
```

```
    public String getPublication() {
```

```
        return publication;
```

```
    }
```

```
    public Integer getYearOfPublication() {
```

```
        return yearOfPublication;
```

```
    }
```

```
    public Integer getNumberOfPages() {
```

```
        return numberOfPages;
```

```
    }
```

```
    public String getImage() {
```

```
        return image;
```

```
    }
```

```
}
```

Create a BookDAL class under the src folder

```
package eecs4313.demo.mockitoExample;

import eecs4313.demo.mockitoExample.Book;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

/**
 * API layer for persisting and retrieving the Book objects.
 */
public class BookDAL {

    private static BookDAL bookDAL = new BookDAL();

    public List<Book> getAllBooks(){
        return Collections.EMPTY_LIST;
    }

    public Book getBook(String isbn){
        return null;
    }

    public String addBook(Book book){
        return book.getIsbn();
    }
}
```

```
public String updateBook(Book book){
    return book.getIsbn();
}

public static BookDAL getInstance(){
    return bookDAL;
}
}
```

The DAL layer has no functional and we want to unit test the piece of code.

In addition, DAL might later communicate with an ORM Mapper or Database API, which we are not concerned with yet.

We want to be able to test the DAL class without actually configuring the data source by using the Mocks

Editing the POM file



The screenshot shows an IDE window with several tabs: TestAccount..., MockServer.java, TestM.java, M.java, Server.java, and Bo. The main editor displays the content of a pom.xml file. The file structure is as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>eecs4313.demo</groupId>
  <artifactId>mockitoExample</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>mockitoExample</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <!-- Dependency for Mockito -->
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <version>1.9.5</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The dependency for JUnit is highlighted with a blue background. A red dashed box is drawn around the entire <dependencies> section. The IDE interface at the bottom shows a navigation bar with tabs: Overview, Dependencies, Dependency Hierarchy, Effective POM, and pom.xml.

Create a JUnit test class for BookDAL

New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

setUpBeforeClass() tearDownAfterClass()
 setUp() tearDown()
 constructor

Do you want to add comments? (Configure templates and default value [here](#))

Generate comments

Class under test:

New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- BookDAL
 - getAllBooks()
 - getBook(String)
 - addBook(Book)
 - updateBook(Book)
 - getInstance()
- Object
 - Object()
 - getClass()
 - hashCode()
 - equals(Object)
 - clone()
 - toString()
 - notify()
 - notifyAll()
 - wait(long)

4 methods selected.

Create final method stubs
 Create tasks for generated test methods

Inject the mock BookDAL and mock data in the setup

```
package eecs4313.demo.mockitoExample;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import org.mockito.*;

public class BookDALTest extends TestCase {
    private static BookDAL mockedBookDAL;
    private static Book book1;
    private static Book book2;

    @Before
    public void setUp(){
        //Create mock object of BookDAL
        mockedBookDAL = Mockito.mock(BookDAL.class); // create a mock object of BookDAL

        //Create few instances of Book class.
        book1 = new Book("8131721019", "Compilers Principles",
            Arrays.asList("D. Jeffrey Ulman", "Ravi Sethi", "Alfred V. Aho", "Monica S. Lam"),
            "Pearson Education Singapore Pte Ltd", 2008, 1009, "BOOK_IMAGE");

        book2 = new Book("9788183331630", "Let Us C 13th Edition",
            Arrays.asList("Yashavant Kanetkar"), "BPB PUBLICATIONS", 2012, 675, "BOOK_IMAGE");

        //Stubbing the methods of mocked BookDAL with mocked data, such that whenever the API is invoked, the mocked data is returned
        Mockito.when(mockedBookDAL.getAllBooks()).thenReturn(Arrays.asList(book1, book2));
        Mockito.when(mockedBookDAL.getBook("8131721019")).thenReturn(book1);
        Mockito.when(mockedBookDAL.addBook(book1)).thenReturn(book1.getIsbn());
        Mockito.when(mockedBookDAL.updateBook(book1)).thenReturn(book1.getIsbn());
    }
}
```

Populating the rest of the methods

@Test

```
public void testGetAllBooks() throws Exception {  
  
    List<Book> allBooks = mockedBookDAL.getAllBooks();  
    assertEquals(2, allBooks.size());  
    Book myBook = allBooks.get(0);  
    assertEquals("8131721019", myBook.getIsbn());  
    assertEquals("Compilers Principles", myBook.getTitle());  
    assertEquals(4, myBook.getAuthors().size());  
    assertEquals((Integer)2008, myBook.getYearOfPublication());  
    assertEquals((Integer) 1009, myBook.getNumberOfPages());  
    assertEquals("Pearson Education Singapore Pte Ltd", myBook.getPublication());  
    assertEquals("BOOK_IMAGE", myBook.getImage());  
}
```

@Test

```
public void testGetBook(){  
  
    String isbn = "8131721019";  
  
    Book myBook = mockedBookDAL.getBook(isbn);  
  
    assertNotNull(myBook);  
    assertEquals(isbn, myBook.getIsbn());  
    assertEquals("Compilers Principles", myBook.getTitle());  
    assertEquals(4, myBook.getAuthors().size());  
    assertEquals("Pearson Education Singapore Pte Ltd", myBook.getPublication());  
    assertEquals((Integer)2008, myBook.getYearOfPublication());  
    assertEquals((Integer)1009, myBook.getNumberOfPages());  
}
```

@Test

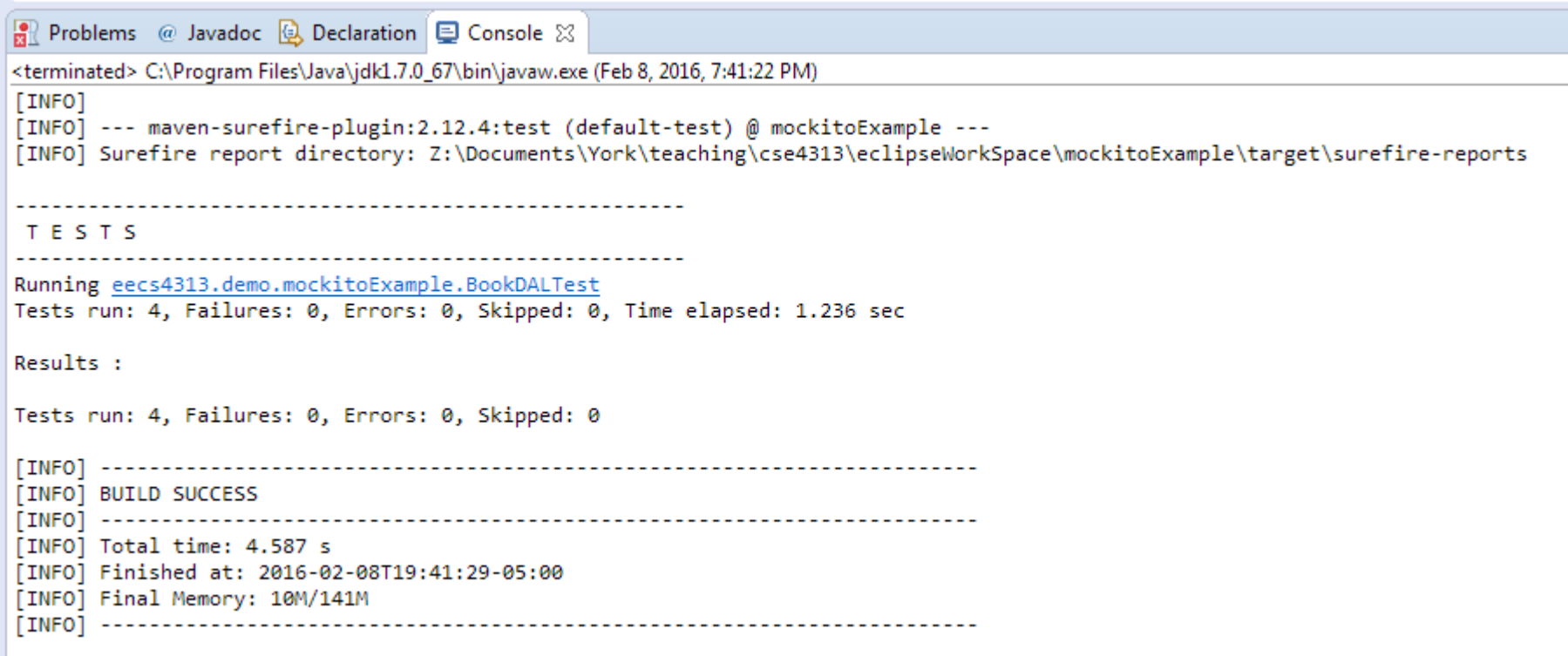
```
public void testAddBook(){  
    String isbn = mockedBookDAL.addBook(book1);  
    assertNotNull(isbn);  
    assertEquals(book1.getIsbn(), isbn);  
}
```

@Test

```
public void testUpdateBook(){  
  
    String isbn = mockedBookDAL.updateBook(book1);  
    assertNotNull(isbn);  
    assertEquals(book1.getIsbn(), isbn);  
}  
}
```

Run the test code

- Right click “mockitoExample” -> Run as -> Maven test



```
Problems @ Javadoc Declaration Console
<terminated> C:\Program Files\Java\jdk1.7.0_67\bin\javaw.exe (Feb 8, 2016, 7:41:22 PM)
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ mockitoExample ---
[INFO] Surefire report directory: Z:\Documents\York\teaching\cse4313\eclipseWorkspace\mockitoExample\target\surefire-reports

-----
T E S T S
-----

Running eecs4313.demo.mockitoExample.BookDALTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.236 sec

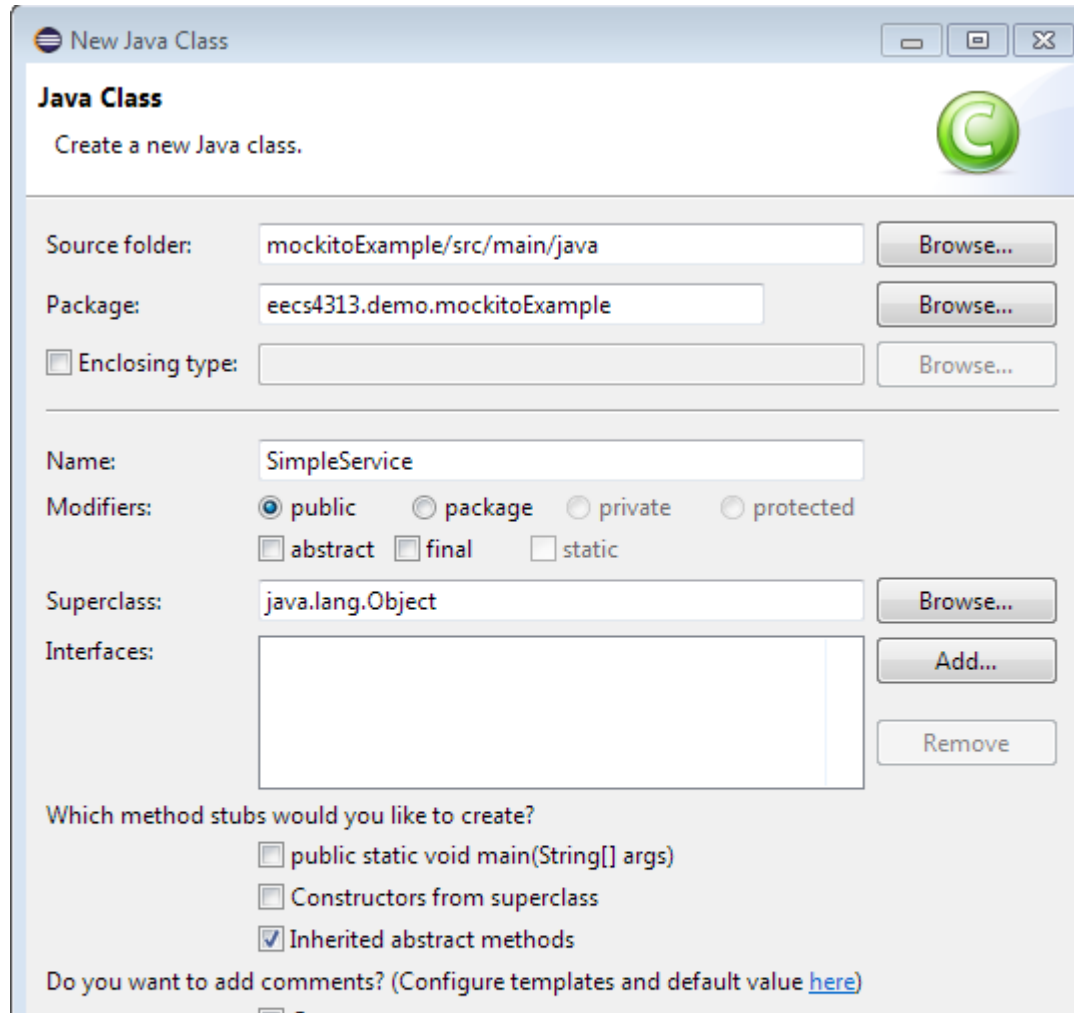
Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.587 s
[INFO] Finished at: 2016-02-08T19:41:29-05:00
[INFO] Final Memory: 10M/141M
[INFO] -----
```


Another Example

- Within this project, create a SimpleService Java class in the src



New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

Enclosing type:

Name:

Modifiers: public package private protected
 abstract final static

Superclass:

Interfaces:

Which method stubs would you like to create?

public static void main(String[] args)
 Constructors from superclass
 Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

SimpleService.java

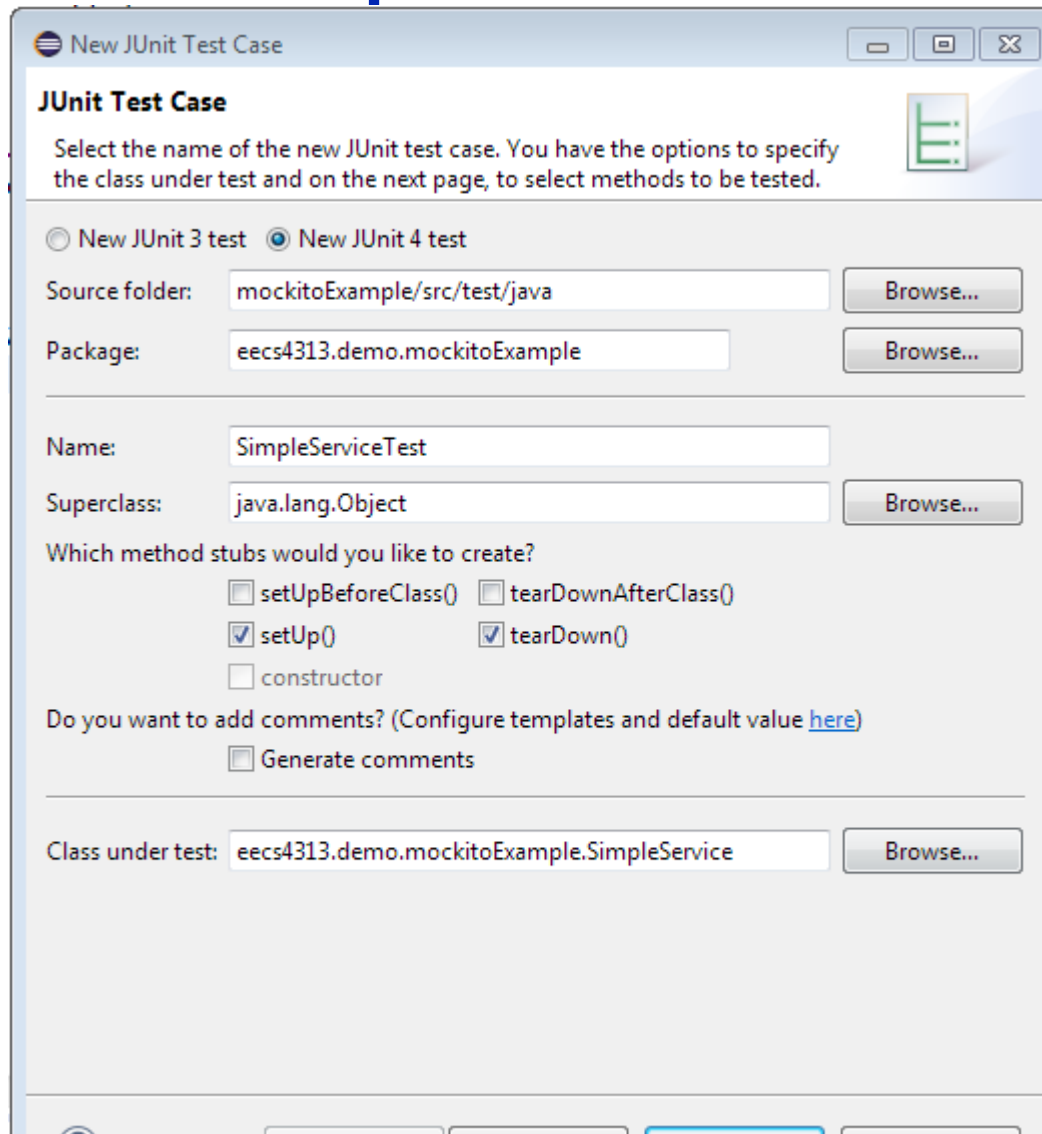
```
package eecs4313.demo.mockitoExample;

public class SimpleService {
    public int getUniqueId() {
        return 43;
    }

    public int callMe(int num) {
        invokeSomeMethod("");
        return num;
    }

    public void invokeSomeMethod(String someMethod) {
        callMe(1);
    }
}
```

Create a JUnit test case for SimpleService



New JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

setUpBeforeClass() tearDownAfterClass()
 setUp() tearDown()
 constructor

Do you want to add comments? (Configure templates and default value [here](#))

Generate comments

Class under test:

```
package eecs4313.demo.mockitoExample;
```

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

```
import org.junit.After;
```

```
import org.junit.Before;
```

```
import org.junit.Ignore;
```

```
import org.junit.Test;
```

```
import java.util.*;
```

```
import org.mockito.*;
```

SimpleServiceTest.java

```
public class SimpleServiceTest {
```

```
    // Test 1
```

```
    @Test
```

```
    public void testGetUniqueId() {
```

```
        // create mock
```

```
        SimpleService test = Mockito.mock(SimpleService.class);
```

```
        // define return value for method getUniqueId()
```

```
        Mockito.when(test.getUniqueId()).thenReturn(43);
```

```
        // use mock in test....
```

```
        assertEquals(test.getUniqueId(), 43);
```

```
    }
```

```
    // Test more than one return value.
```

```
    // Demonstrates the return of multiple values
```

```
    @Test
```

```
    public void testMoreThanOneReturnValue() {
```

```
        Iterator i = Mockito.mock(Iterator.class);
```

```
        Mockito.when(i.next()).thenReturn("Mockito").thenReturn("is neat!!");
```

```
        String result = i.next() + " " + i.next();
```

```
        assertEquals("Mockito is neat!!", result);
```

```
    }
```

```
// Test return value dependent on method parameter.
```

```
@Test
```

```
public void testReturnValueDependentOnMethodParameter() {
```

```
    Comparable c = Mockito.mock(Comparable.class);
```

```
    Mockito.when(c.compareTo("Mockito")).thenReturn(1);
```

```
    Mockito.when(c.compareTo("Eclipse")).thenReturn(2);
```

```
    // assert
```

```
    assertEquals(1, c.compareTo("Mockito"));
```

```
}
```

```
/**
```

```
 * Test return value in dependent on method parameter.
```

```
 */
```

```
@Test
```

```
public void testReturnValueInDependentOnMethodParameter() {
```

```
    Comparable c = Mockito.mock(Comparable.class);
```

```
    Mockito.when(c.compareTo(Mockito.anyInt())).thenReturn(-1);
```

```
    assertEquals(-1, c.compareTo(9));
```

```
}
```

```
@Test
```

```
public void testVerify() {
```

```
    // create and configure mock
```

```
    SimpleService test = Mockito.mock(SimpleService.class);
```

```
    Mockito.when(test.getUniqueId()).thenReturn(43);
```

```
    // call method testing on the mock with parameter 12
```

```
    test.callMe(12);
```

```
    test.getUniqueId();
```

```
    test.getUniqueId();
```

```
    test.invokeSomeMethod("Hello World");
```

```
    test.invokeSomeMethod("called at least once");
```

```
    test.invokeSomeMethod("called at least twice");
```

```
    test.invokeSomeMethod("called five times");
```

```
    test.invokeSomeMethod("called at most 3 times");
```

SimpleServiceTest.java (continued)

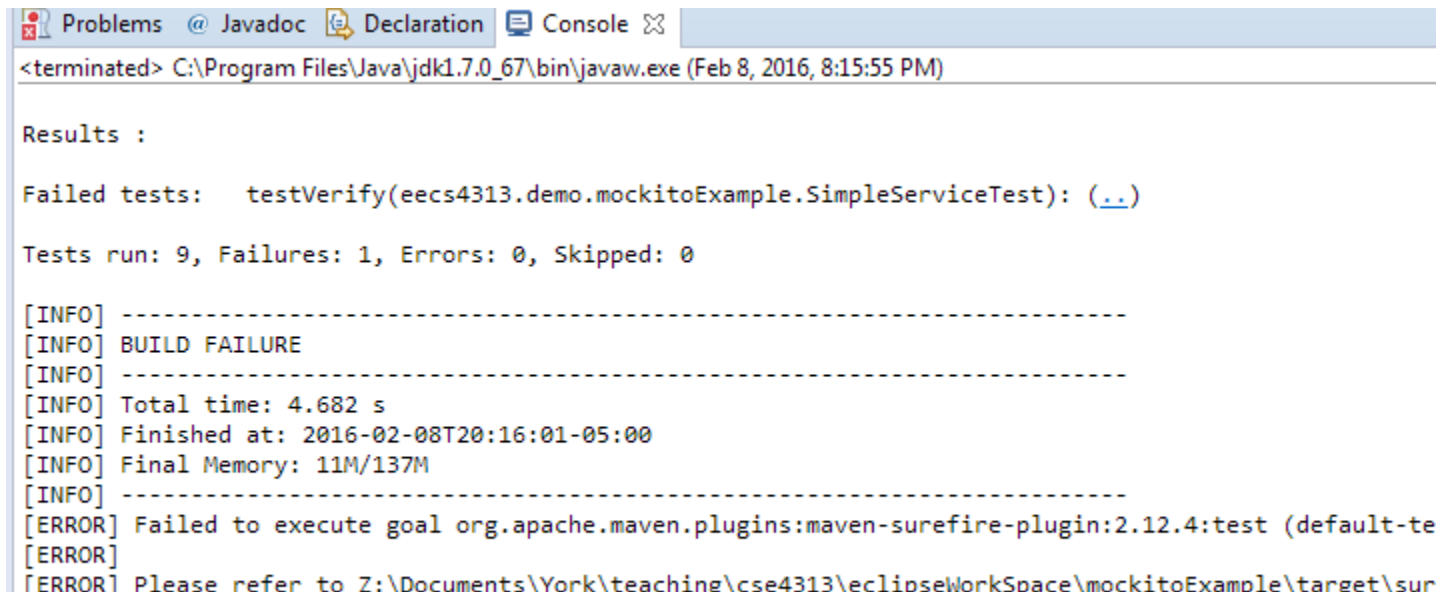
```
// now check if method testing was called with the parameter 12
Mockito.verify(test).callMe(Matchers.eq(12));

// was the method called twice?
Mockito.verify(test, Mockito.times(2)).getUniqueId();

// other alternatives for verifying the number of method calls for a method
Mockito.verify(test, Mockito.never()).invokeSomeMethod("never called");
Mockito.verify(test, Mockito.atLeastOnce()).invokeSomeMethod("called at least once");

// Will all fail because we didn't met the conditions.
Mockito.verify(test, Mockito.atLeast(2)).invokeSomeMethod("called at least twice");
Mockito.verify(test, Mockito.times(5)).invokeSomeMethod("called five times");
Mockito.verify(test, Mockito.atMost(3)).invokeSomeMethod("called at most 3 times");
}
}
```

Once done, right click the project folder, Run as -> Maven test



The screenshot shows an IDE console window with the following content:

```
Problems @ Javadoc Declaration Console
<terminated> C:\Program Files\Java\jdk1.7.0_67\bin\javaw.exe (Feb 8, 2016, 8:15:55 PM)

Results :

Failed tests:   testVerify(eecs4313.demo.mockitoExample.SimpleServiceTest): (..)

Tests run: 9, Failures: 1, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 4.682 s
[INFO] Finished at: 2016-02-08T20:16:01-05:00
[INFO] Final Memory: 11M/137M
[INFO] -----
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test (default-te
[ERROR]
[ERROR] Please refer to Z:\Documents\York\teaching\cse4313\eclipseWorkspace\mockitoExample\target\sur
```

More Mockito

- Mockito API

- <https://mockito.googlecode.com/hg-history/1.5/javadoc/org/mockito/Mockito.html>

- Unit tests with Mockito

- <http://www.vogella.com/tutorials/Mockito/article.html>

Log String

- Often one needs to test that the sequence in which methods are called is correct
- Solution: Have each method append to a log string when it is called
 - Then, assert that the log string is the correct one
 - Requires changes to the implementation

Accessing private fields

- Object-oriented design guidelines often designate that certain fields should be private / protected
- This can be a problem for testing since a tester may need to assert certain conditions about private fields
- Making these fields public defeats the purpose

A solution

- Using reflection, one can actually call private methods and access private attributes!
- An example

```
class A {  
    private String sayHello(String name) {  
        return "Hello, " + name;  
    }  
}
```

```
import java.lang.reflect.Method;

public void testPrivateMethod {
    A test = new A();
    Method[] methods = test.getClass().getDeclaredMethods();

    for (int i = 0; i < methods.length; ++i) {
        if (methods[i].getName().equals("sayHello")) {
            Object params[] = {"Ross"};
            methods[i].setAccessible(true);
            Object ret = methods[i].invoke(test, params);
            System.out.println(ret);
        }
    }
}
```

Testing Code Smells

Relevant Readings

- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. Refactoring – Improving the Design of Existing Code.
- Steve McConnell. Code Complete: : A Practical Handbook of Software Construction. (Chapter 24)
- Refactoring test code. Van Deursen et al. Tech Report. 2001.
- xUnit Patterns.
 - <http://xunitpatterns.com/Test%20Smells.html>

Problem: "Bit rot"

- After several months and new versions, many codebases reach one of the following states:
 - *rewritten* : Nothing remains from the original code.
 - *abandoned* : Original code is thrown out, rewritten from scratch.
- Why?
 - Systems evolve to meet new needs and add new features
 - If the structure of the code does not also evolve, it will "rot"
 - This can happen even if the code was initially reviewed and well-designed at the time of check-in

Software maintenance

- **Software maintenance:** Modification or repair of a software product after it has been delivered.
- Purposes:
 - fix bugs
 - improve performance
 - improve design
 - add features,
 - etc.
- Studies have shown that ~80% of maintenance is for non-bug-fix-related activities such as adding functionality (Pigosky 1997).

Maintenance is hard

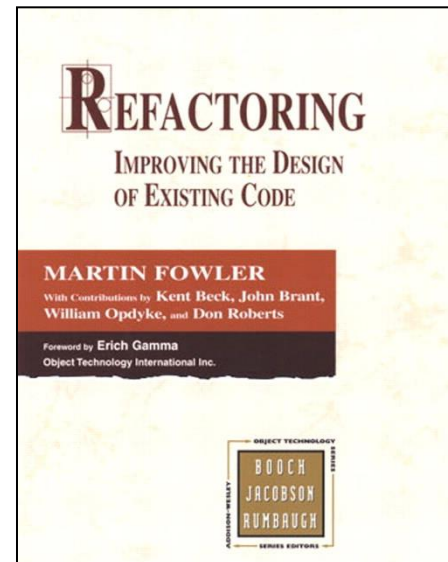
- It's harder to maintain code than write your own new code.
 - must understand code written by another developer, or code you wrote at a different time with a different mindset
 - most developers dislike software maintenance
- Maintenance is how developers spend much of their time.
- It pays to design software well and plan ahead so that later maintenance will be less painful.
 - Capacity for future change must be anticipated

Refactoring

- **Software refactoring** is the systematic practice of improving application code's structure without altering its behavior.
 - Incurs a short-term time/work cost to reap long-term benefits
 - A long-term investment in the overall quality of your system.
- Refactoring is not the same thing as:
 - adding features
 - debugging code
 - rewriting code

A Brief History of Code Refactoring

- Invented by two computer science graduate students in the late 1980s:
 - Bill Opdyke at University of Illinois at Urbana-Champaign
 - Bill Griswold at University of Washington
- Canonical reference



Why refactor?

- Why fix a part of your system that isn't broken?
 - Each part of your system's code has the following three purposes. If the code does not do one or more of these, it is "broken."
 1. to execute its functionality,
 2. to allow change,
 3. to communicate well to developers who read it.
- Refactoring:
 - changes internal structure of the program and improves software's design
 - makes it easier to understand and cheaper to modify
 - do not change its observable behaviour

When to refactor?

- When is it best for a team to refactor their code?
 - best done **continuously** (like testing) as part of the process
 - hard to do well late in a project (like testing)
- Refactor when you identify an area of your system that:
 - isn't well designed
 - isn't thoroughly tested, but seems to work so far
 - now needs new features to be added

Bad smells



If it stinks,
change it

*Kent Beck grandma
discussing child-rearing philosophy*

Signs you should refactor

- “Code Smells”

- code is **duplicated**
- a routine is **too long**
- a loop is too long or **deeply nested**
- a class has poor **cohesion**
- a class uses too much **coupling**
- inconsistent level of **abstraction**
- too many **parameters**
- to **compartmentalize** changes (change one place → must change others)
- to modify an **inheritance hierarchy** in parallel
- to **group related data** into a class
- a “**middle man**” object doesn't do much
- **poor encapsulation** of data that should be private
- a **weak subclass** doesn't use its inherited functionality
- a class contains **unused code**

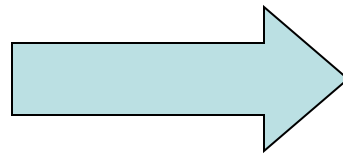
Code Smells

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
 - (change one place → must change others)
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
 - (subclass doesn't use inherited members much)
- Comments

What smells in here?

```
void funcA()  
{  
    int x, y = 2;  
    x = y * y;  
    printf("%d", x);  
}
```

```
void funcB()  
{  
    int x, y = 4;  
    x = y * y;  
    funcC(x);  
}
```



Duplicate Code

```
void funcA()  
{  
    ...  
    x = sqr(y);  
    ...  
}  
void funcB()  
{  
    ...  
    x = sqr(y);  
    ...  
}  
int sqr(int x)  
{  
    return x * x;  
}
```

Extract Method

Duplicated Code

- Code repeated in multiple places
- Refactoring
 - Extract Method
 - Extract Class
 - Pull Up Method
 - Form Template Method

What smells in here?

```
void funcA(  
    int param1,  
    int param2,  
    char* param3,  
    float param4,  
    float param5,  
    void* param6)  
{  
    int temp1, temp2,  
        temp3;  
  
    // Do stuff with all  
    // this data  
}
```



Long Parameter
List

```
class newObj  
{  
public:  
    int getParam1();  
    int getParam2();  
    char* getParam3();  
    float getParam4();  
    float getParam5();  
    void* getParam6();  
}  
void funcA(newObj obj)  
{  
    int temp1, temp2, temp3;  
    // Do stuff with all  
    // this data  
}
```

*Introduce Parameter
Object*

Test Smells

- Mystery Guest
- Resource Optimism
- Test Run War
- General Fixture
- Eager Test
- Lazy Test
- Assertion Roulette
- Indirect Testing
- For Testers Only
- Sensitive Equality
- Test Code Duplication

Test Smells

- Mystery Guest
- Resource Optimism
- Test Run War
- General Fixture
- **Eager Test**
- Lazy Test
- Assertion Roulette
- Indirect Testing
- For Testers Only
- Sensitive Equality
- Test Code Duplication

Eager Test

- When a test method checks several methods of the object to be tested, it is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.
- The solution is simple: separate the test code into test methods that test only one method using Fowler's Extract Method (F:110), using a meaningful name highlighting the purpose of the test. Note that splitting into smaller methods can slow down the tests due to increased setup/teardown overhead.

Test Smells

- Mystery Guest
- Resource Optimism
- Test Run War
- General Fixture
- Eager Test
- **Lazy Test**
- Assertion Roulette
- Indirect Testing
- For Testers Only
- Sensitive Equality
- Test Code Duplication

Lazy Tests

- This occurs when several test methods check the same method using the same fixture (but for example check the values of different instance variables). Such tests often only have meaning when considering them together so they are easier to use when joined using Inline Method (F:117)