Western

UNIVERSITY · CANADA

Topic 10

Software Testing

Partly based on old CS 2212 notes; partly based on Martin Fowler's excellent article, *Mocks Aren't Stubs*; partly based on this excellent presentation; partly based on the experience of your instructor. ☺

Computer Science 2212b
Introduction to Software Engineering
Winter 2014

Jeff Shantz
jeff@csd.uwo

Agenda

- Introduction to testing
- Unit testing
- Drivers
- Stubs
- Mockito / PowerMock
  - Mocks
  - Stubbing Multiple Calls
  - Custom Answers
  - Stubbing Static Methods

Introduction to Testing

What is the purpose of testing?

- **Historical View**
  - *Testing is done to show the system works*
  - Tend to go easy on the program
  - Programmers use same logic to test as they did to code
  - Some (many) bugs do not get caught
- **Modern View**
  - *Testing is done to uncover bugs*
  - We purposely take the attitude of trying to break the program
  - Result: more bugs caught, more reliable system

Terminology

Test Case: A set or sequence of inputs used to test a program, along with an expected output

```
@Test
public void testAddTwoNegativeNumbers() {
  Calculator calculator = new Calculator();
  int result = calculator.add(-4, -5);
  Assert.assertEquals(-9, result);
}
```

A JUnit test case is shown here, but test cases need not necessarily be written in code.

Terminology

Test Suite: A set of test cases

- In JUnit, a test class could be considered a test suite
- JUnit also provides a `TestSuite` class
  - Can be used to group test classes into suites
    - A suite consisting of all database-related tests
    - A suite consisting of all slow tests
    - etc.

LIVE DEMO

(https://github.com/jsuwo/junit-testsuites)

Good Test Cases and Test Suites

- **Good test case:** one we think is likely to uncover a bug.
- **Good test suite:** contains good test cases and tests the requirements thoroughly.
- Hence,
    - The better our test suite, and
    - The more of it our software passes
    - The more **confidence** we can have in our software

Failures, Faults, and Errors

Bug

- Informal term that can mean several different things
- Sometimes, it is more useful to use precise terminology

Failure

- Something the program does wrong (crashing, incorrect result)

Fault

- The incorrect code causing the failure (= instead of ==)

Error

- Mistake the programmer made leading to the fault (made a typo; didn't realize that == was needed)

When it is unambiguous, we still often use the term ***bug***.

Can We Find All Bugs?

Software errors tend to follow the Pareto Principle (80-20 rule)

- **80% of the failures are caused by 20% of the faults**
  - Easier to find – failures occur frequently
- **20% of the failures are caused by 80% of the faults**
  - Less frequent and therefore harder to find

Some failures may be very hard to find

- Timing issues (race conditions)
- Complex interactions with external systems

In a large system, it is likely we will never find all the bugs

- Avoid, find, eliminate as many bugs as possible
- Build failsafe checks to alleviate effects of faults

Testing vs. Debugging

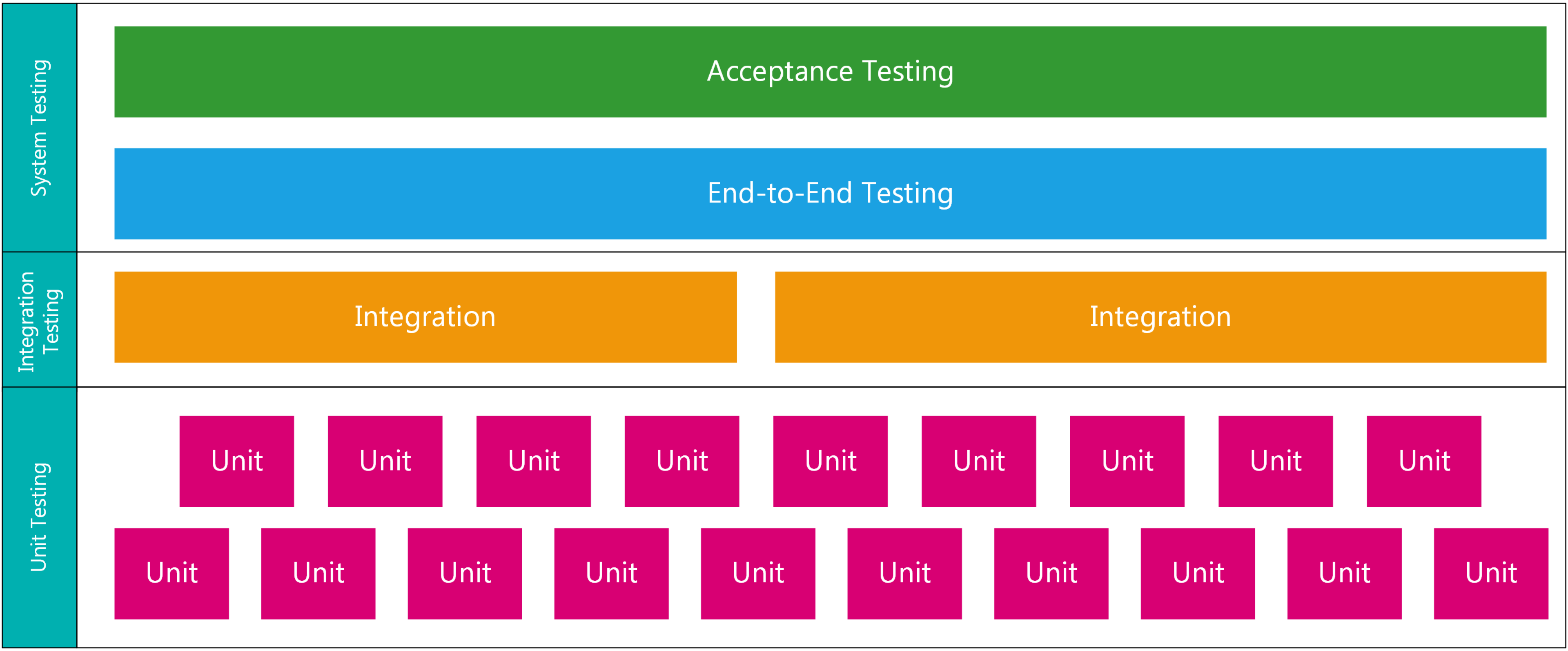
Testing: running test cases, finding *failures*

- Can often be done without looking at the code
- Can be automated or partially automated

Debugging: finding and correcting *faults*

- Need to work with the code
- Use a debugger
- Cannot generally be automated
  - Static analysis tools exist for finding certain faults (findbugs)

Levels of Testing



Unit testing: testing individual methods and classes

Integration testing: ensuring that modules compile and interoperate correctly

System Testing: testing the whole system

- **End-to-end testing**: test workflows end-to-end
- **Acceptance testing:** workflows tested by the client/user or a tester acting as such

Methods of Software Testing

White-Box Testing (Structural Testing)

- Look at finished code
- Try to write tests to execute paths through it
- See what code has been missed (e.g. using code coverage reports)
- Create more tests to test missed code

Black-Box Testing (Functional Testing)

- Look at requirements
- Construct cases which correspond directly to these
- Write automated tests to test these cases
- See which requirements have been missed
- Create more tests to test missed requirements

Methods of Software Testing

White-Box Testing (Structural Testing)

- Examples of tools used: JUnit, TestNG
- Can be applied at all levels
- Typically dominates the unit level

Black-Box Testing (Functional Testing)

- Examples of tools used: Cucumber, Selenium, AutoIt, FEST
- Can be applied at all levels
- Typically dominates the system level (end-to-end, acceptance)

We need to use both strategies to develop comprehensive test suites.

Drivers

How do we test a module which is low in the hierarchy without the main program?

- We use a *driver:* a simple main program which exists only to test a low-level function or module

Testing with a driver

- Write the driver
- Compile the driver together with the module
- Run the driver

In practice, this should be used only for the *simplest* of programs. In reality, we use a test framework like JUnit, which acts as the driver of our tests.

Stubs

How do we test a function which relies upon other functions, but we don’t have those functions finished yet?

- We use *stubs:* a simple fake function which *stands in* for another function, usually returning a known value.

Unit tests are intended to test methods in *isolation*

- Any other methods we call from a method should generally be *stubbed out*
- Tools exist to help with this – more later

Reasons for Using Stubs

We’ll usually use a stub if a method we need to test

- uses other methods/classes that aren’t finished yet
  - Stub out the unfinished methods/classes
- uses methods/classes that work with external sources (files, database, network)
  - Unit tests need to be **fast**
  - Unit tests need to be **isolated** (code + database = integration test)
  - Stub out these methods/classes
- uses methods that return different values based on date/time
- uses *stochastic* (non-deterministic) methods

Manually Stubbing Methods

- We’ll see later how tools like Mockito can help here.
- If we just need to stub out one or two methods, though, sometimes it’s easier just to do it manually.

LIVE DEMO

(https://github.com/jsuwo/junit-stubs)

Stubbing Stochastic Methods

- Stubs are easy to create for *deterministic* methods.
- What about non-deterministic methods that exhibit random behaviour?
  - Always return the same value
  - Always return the same sequence of values
    - Use a static variable to track how many times the method has been called
    - On the first invocation, we'll return 5
    - On the second invocation, we'll return 99
    - …

xUnit

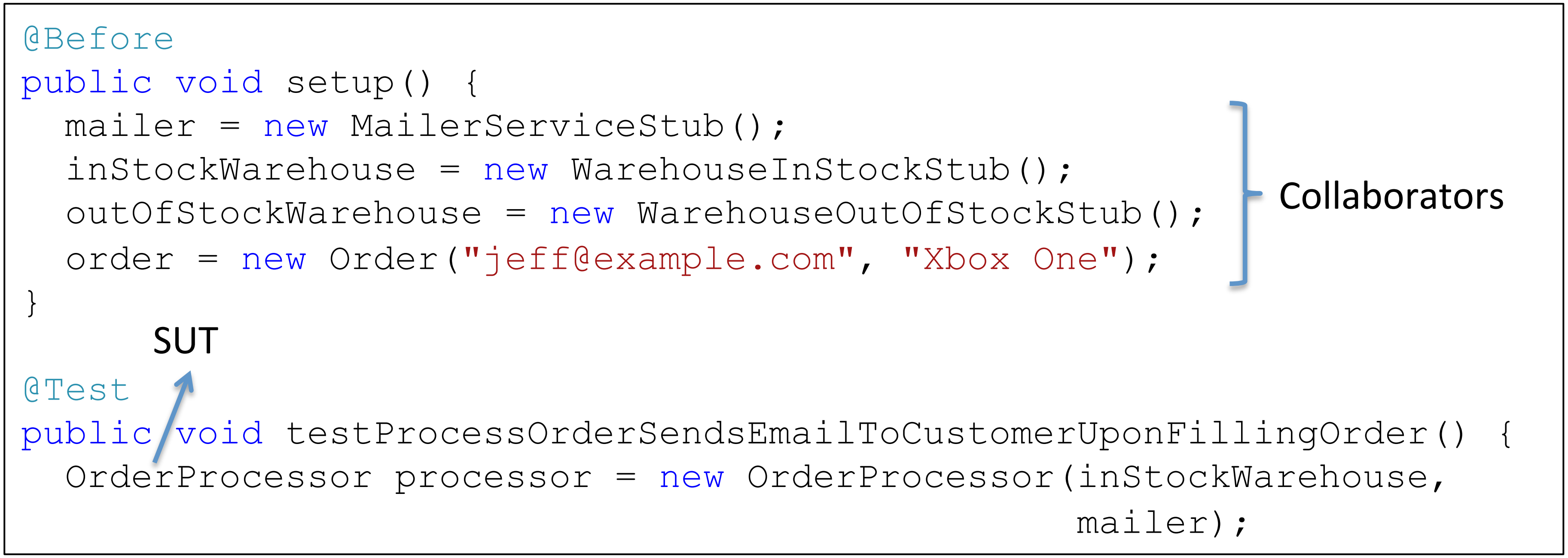
JUnit adopts a style of unit testing that has come to be known as *xUnit*:

- JUnit (Java)
- NUnit (.NET)
- CppUnit (C++)
- …

Tests in xUnit frameworks follow a 4-phase sequence:

1. Setup
2. Exercise
3. Verify
4. Teardown

xUnit Phase 1: Setup



```
@Before
public void setup() {
  mailer = new MailerServiceStub();
  inStockWarehouse = new WarehouseInStockStub();
  outOfStockWarehouse = new WarehouseOutOfStockStub();
  order = new Order("jeff@example.com", "Xbox One");
}

@Test
public void testProcessOrderSendsEmailToCustomerUponFillingOrder() {
  OrderProcessor processor = new OrderProcessor(inStockWarehouse,
                                                mailer);
```

- **Two types of objects created in the Setup phase**
  - The *system-under-test* (SUT)
  - All *collaborators* needed
- **Often implemented across two places:**
  - Partly in `@Before` methods (objects common to multiple tests)
  - Partly in our test methods (objects specific to one test)

xUnit Phase 2: Exercise

```
@Test
public void testProcessOrderSendsEmailToCustomerUponFillingOrder() {
  OrderProcessor processor = new OrderProcessor(inStockWarehouse,
                                                mailer);
  processor.processOrder(order);
  assertEquals(1, mailer.numberSent());
}
```

- **In the Exercise phase, we *exercise* the behaviour we want to test**
  - i.e. we call (exercise) the method being tested

xUnit Phase 3: Verify

```
@Test
public void testProcessOrderSendsEmailToCustomerUponFillingOrder() {
  OrderProcessor processor = new OrderProcessor(inStockWarehouse,
                                                mailer);
  processor.processOrder(order);
  assertEquals(1, mailer.numberSent());
}
```

- **`assert` statements form the Verify phase**
    - We check to see if the exercised method carried out its task correctly

xUnit Phase 4: Teardown

```java
@After
public void teardown() {
  databaseConnection.close();
}
```

- **The Teardown phase cleans up**
  - If we have open database connections or files, the teardown phase is where we would close them
  - Typically implemented in `@After` methods

State Verification

So far, the style of testing we've adopted uses *state verification*

- We determine if the exercised method worked by examining:
  - The state of the SUT
  - The state of its collaborators

```
@Test
public void testProcessOrderSendsEmailToCustomerUponFillingOrder() {
  OrderProcessor processor = new OrderProcessor(inStockWarehouse,
                                                mailer);
  processor.processOrder(order);
  assertTrue(order.isProcessed());
  assertEquals(1, mailer.numberSent());
}
```

Mocks

Mocks (mock objects): objects that mimic the behaviour of real objects

- Used in *behaviour verification* (more later)
- Can simulate the behaviour of complex objects
- Implemented using a *mocking library; e.g. in Java,*
  - EasyMock
  - Jmock
  - JMockIt
  - Mockito
- Mocking libraries available in most languages

Mocks

- Can often be confused with *stubs*
  - Mocks allow us to stub methods
  - Also allow us to:
    - Verify that specific methods were called
    - Verify that specific arguments were passed
  - Thus, we can record and verify the *interactions* between the SUT and its collaborators

LIVE DEMO

(https://github.com/jsuwo/junit-mockito)

Behaviour Verification

- **Mocks use *behaviour verification***
  - We test to ensure the correct calls were made

```
@Test
public void testProcessOrderSendsEmailToCustomerUponFillingOrder() {

  when(warehouse.fill(order)).thenReturn(true);

  OrderProcessor processor = new OrderProcessor(warehouse, mailer);
  processor.processOrder(order);

  verify(warehouse).fill(order);
  verify(mailer).send(isA(Message.class));
}
```

- Here, we verify that
  - The `send` method was called on our `mailer` mock
  - The `send` was passed one argument of type `Message`

Behaviour Verification

- **We follow a *given-when-then* pattern when testing using behaviour verification**

Given
When
Then

```
@Before
public void setup() {
  mailer = mock(MailerService.class);
  warehouse = mock(Warehouse.class);
  order = mock(Order.class);

  when(order.getCustomerEmail()).thenReturn("jeff@example.com");
}

@Test
public void testProcessOrderSendsEmailToCustomerUponFillingOrder() {

  when(warehouse.fill(order)).thenReturn(true);

  OrderProcessor processor = new OrderProcessor(warehouse, mailer);
  processor.processOrder(order);

  verify(warehouse).fill(order);
  verify(mailer).send(isA(Message.class));
}
```

Mockito Argument Matchers

In our `when` and `verify` calls, we often want to check the types and values of arguments passed.

Specific object

```
verify(warehouse).fill(order);
```

Object of a specific type

```
verify(mailer).send(isA(Message.class));
```

Any `String` (can also use `anyBoolean`, `anyChar`, `anyDouble`, `anyList`, `anyObject`, etc.

```
when(warehouse.checkStock(anyString()).thenReturn(5);
```

`String` that starts with a value (can also use `endsWith`)

```
when(warehouse.checkStock(startsWith("Xbox").thenReturn(5);
```

`String` that matches a regular expression

```
when(warehouse.checkStock(matches("X.*x").thenReturn(true);
```

Many more Matchers available: http://docs.mockito.googlecode.com/hg/org/mockito/Matchers.html

Stubbing Multiple Calls

The SUT may call a stubbed method multiple times

- We may want to provide different return values each time

```
when(iterator.hasNext()).thenReturn(true, true, true, false);
```

OR

```
when(iterator.hasNext()).thenReturn(true)
                        .thenReturn(true)
                        .thenReturn(true)
                        .thenReturn(false);
```

- We may want to verify that a method has been called a specific number of times (can also use `atMost`, `atLeastOnce`, `atLeast`, `never`, etc.)

```
verify(iterator, times(4)).hasNext();
```

Stubbing Multiple Calls

LIVE DEMO

(https://github.com/jsuwo/junit-mockito-stubmultiple)

Custom Answers

Sometimes, we want to verify state on objects passed to a stubbed method.

- Can do this in a custom `Answer`
- Suppose we are testing a `registerStudent` method in a `Registrar` class:

```
public class Registrar {

  public void registerStudent(String sNumber, Course course) {
    Student student = new Student(sNumber);
    course.addStudent(student);
  }

}
```

Custom Answers

We mock the `Course` object in our test…

```
@Test
public void testRegisterStudentRegistersTheCorrectStudent() {
  Course course = mock(Course.class);
  when(course.addStudent(anyObject())).thenReturn(true);

  Registrar registrar = new Registrar();
  registrar.registerStudent(“250000000”, course);

  verify(course).addStudent(isA(Student.class));
}
```

- We verify that the `addStudent` method was called and passed a `Student` object
- What if we want to verify that the `Student` object passed has the correct student number?

Custom Answers

We can stub the `addStudent` method with a custom `Answer`, and add our assertions to the `answer` method:

```
@Test
public void testRegisterStudentRegistersTheCorrectStudent() {
  Course course = mock(Course.class);

  doAnswer(new Answer() {
    public Object answer(InvocationOnMock invocation) {
      Object[] args = invocation.getArguments();

      Student s = (Student)args[0];
      assertEquals(s.getStudentNumber(), “250000000”);
      return true;
    }
  }).when(course).addStudent(isA(Student.class));

  Registrar registrar = new Registrar();
  registrar.registerStudent(“250000000”, course);
}
```

Stubbing Static Methods

How do we stub a static method?

- We have no instance to *mock*
- Mockito is not capable of stubbing static methods
- PowerMock adds extensions to Mockito to allow this

Step 1: Add Annotations to the Test Class

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(Static.class)

public class StaticTest { … }
```

Stubbing Static Methods

Step 2: Tell PowerMockito to mock the class in question

```
@Test
public void testMethod() {
  PowerMockito.mockStatic(Static.class);
```

Step 3: Stub the method as usual

```
  when(Static.method()).thenReturn(42);
```

Step 4: Verify that the method was called, if desired

```
  …
  PowerMockito.verifyStatic();
  Static.method();
```

Test-Driven Mailer Class Demo

LIVE DEMO

(no repo for this one – come to class)

Testing Serialization Demo

LIVE DEMO

(no repo for this one – come to class)

Unit vs. Integration Tests

Code

+

Database

INTEGRATION TEST

Unit vs. Integration Tests

My Code

+

Your Code

INTEGRATION TEST

Unit vs. Integration Tests

First, unit test your modules and mock out other modules

Unit vs. Integration Tests

Code

+

Mock of Database

UNIT TEST

Unit vs. Integration Tests

My Code

+

Mock of Your Code

UNIT TEST

Unit vs. Integration Tests

Then, test them together.

With only integration tests, can’t definitively say
- *The problem is in your code*
- *The problem is in the database*

Hence, we **<u>waste time</u>** finding the bug.

Unit + integration tests means
- *My code works*
- *My code works with the database*
- *My code works with your code*

System Tests

System tests ensure that everything works together

My Code
+
Your Code
+
Database
+
Server
+
Client
+
Kitchen Sink

Western

UNIVERSITY · CANADA