Unit Testing and Scaffolding

Course of Software Engineering II
A.A. 2010/2011

Valerio Maggio, PhD Student
Prof. Sergio Di Martino
Testing Preliminaries

► You're a programmer
  ○ a coder, a developer, or maybe a hacker!

► As such, it's almost impossible that you haven't had to sit down with a program that you were sure was ready for use
  ○ or worse yet, a program you knew was not ready

► So, you put together a bunch of test to prove the correctness of your code
  ○ Let's think about what correctness means
Testing Preliminaries

► Correctness in Testing?
  ○ Which is the point of view?

► Different points of view means different types of tests (testing)

► Automated tools support
Outline

- Testing Taxonomy
  - Brief Introduction

- Unit testing with JUnit 4.x
  - Main differences with JUnit 3.x
  - Backward compatibilities
  - JUnit Examples in practice

- Test Scaffolding and Mocking
1. Types of testing
Example Scenario

► (... not properly related to Computer Science :)

► Please, imagine that you have to test a building
  o Test if it has been constructed properly
  o Test if it is able to resist to earthquake
  o ....

► What types of “testing” will you do?
  o Make an educated guess
Five Types of Testing

- Acceptance testing
- Stress/load testing
- Functional testing
- Integration testing
- Unit testing
Unit Testing

► Testing of the smallest pieces of a program
  ○ Individual functions or methods

► Keyword: Unit
  ○ (def) Something is a unit if it there's no meaningful way to divide it up further

► Buzz Word:
  ○ Testing in isolation
Unit Testing (cont.)

- Unit test are used to test a single unit in isolation
  - Verifying that it works as expected
  - No matter the rest of the program would do

- Possible advantages?
  - (Possibly) No inheritance of bugs of mistakes from made elsewhere
  - Narrow down on the actual problem
Unit Testing (cont.)

► Is it enough?
  ○ No, by itself, but...

► ... it is the foundation upon which everything is based!

► (Back to the example)
  ○ You can't build a house without solid materials.
  ○ You can't build a program without units that works as expected.
Integration Testing

- Aim: Push further back boundaries of isolation
- Tests encompass interactions between related units

(Important)

**Every test should still run in isolation**
- To avoid inheriting problems from outside

- Tests check whether the tested unit behave correctly as a group
 Functional Testing

- a.k.a. System Testing
- *Extends boundaries of isolation even further*
  - *To the point they don't even exist.*
- Testing application *Use Cases*

- System tests are very useful, but not so useful without Unit and Integration tests
  - You have to be sure of the pieces before you can be sure of the whole.
Stress and Acceptance Testing

► Stress/Load Testing
  ○ Test of loadings and performances
    ● (Non functional requirements)
  ○ Test if the building is able to resist to the earthquake

► Acceptance Testing
  ○ Last but not least....
  ○ ... does the customer get what he or she expected?
Before going on...

Let's take a look at this code, please

class Testable(object):
    
def method1(self, number):
        number += 4
        number **= 0.5
        number *= 7
        return number
    
def method2(self, number):
        return ((number * 2) ** 1.27) * 0.3
    
def method3(self, number):
        return self.method1(number) + self.method2(number)
    
def method4(self):
        return 1.713 * self.method3(id(self))
2. JUnit Testing Framework
JUnit 3.x Design

Design that is compliant with *xUnit framework* guidelines
JUnit Assertions

- `assertNotNull`
  - Test passes if Object is not null.

- `assertNull`
  - Test passes if Object is null.

- `assertEquals`
  - Asserts equality of two values

- `assertTrue`
  - Test passes if condition is True

- `assertFalse`
  - Test passes if condition is False

- `assertSame`
  - Test passes if the two Objects are not the same Object
JUnit 4.x Design

- Main features inspired from other Java Unit Testing Frameworks
  - TestNG
- Test Method **Annotations**
  - Requires Java5+ instead of Java 1.2+
- Main Method Annotations
  - @Before, @After
  - @Test, @Ignore
  - @SuiteClasses, @RunWith
Java5 Annotations at glance

- **Meta Data Tagging**
  - `java.lang.annotation`
  - `java.lang.annotation.ElementType`
    - `FIELD`
    - `METHOD`
    - `CLASS`
    - `...`

- **Target**
  - Specify to which `ElementType` is applied

- **Retention**
  - Specify how long annotation should be available
JUnit Test Annotation

```java
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Test {

/**
 * Default empty exception
 */
    static class None extends Throwable {
        private static final long serialVersionUID = 1L;
        private None() {
        }
    }

/**
 * Optionally specify <code>expected</code>, a Throwable, to cause a test method to succeed iff
 * an exception of the specified class is thrown by the method.
 */
    Class<? extends Throwable> expected() default None.class;

/**
 * Optionally specify <code>timeout</code> in milliseconds to cause a test method to fail if it
 * takes longer than that number of milliseconds.*/
    long timeout() default 0L;
}
Testing exception handling

► *Test anything that could possibly fail*

```java
public class TestDefaultController extends TestCase {
    ...
    public void testGetHandlerNotDefined() {
        try {
            SampleRequest request = new SampleRequest("testNotDefined");
            // The following line is supposed to throw a RuntimeException
            controller.getHandler(request);
            fail;
        }
        catch (RunTimeException e) {
            assert true;
        }
    }
    ...
}
```
New way of Testing exception handling

► Test anything that could possibly fail

```java
public class TestDefaultController {
    ...
    @Test(expected=RuntimeException.class)
    public void testGetHandlerNotDefined() {
        SampleRequest request = new SampleRequest("testNotDefined");
        // The following line is supposed to throw a RuntimeException
        controller.getHandler(request);
    }
    ...
}
```
JUnit 4.x backward compatibility

JUnit provides a façade class which operates with any of the test runners.
- org.junit.runner.JUnitCore
JUnit Matchers: Hamcrest

- JUnit 4.4+ introduces **matchers**
  - Imported from [Hamcrest project](http://code.google.com/p/hamcrest/)

- Matchers improve testing code refactoring
  - Writing more and more tests assertion became hard to read
  - **Remember:**
    - Documentation purposes

- Let's do an example …
Matchers Example

```java
public class HamcrestTest {
  private List<String> values;
  @Before
  public void setUpList() {
    values = new ArrayList<String>();
    values.add("x");
    values.add("y");
    values.add("z");
  }

  @Test
  public void withoutHamcrest() {
    assertTrue(values.contains("one")
               || values.contains("two")
               || values.contains("three"));
  }

  @Test
  public void withHamcrest() {
    assertThat(values, hasItem(anyOf(equalTo("one"),
                                     equalTo("two"),
                                     equalTo("three"))));
  }
}```
JUnit 4.x Extensions
3. Testing Scaffolding

*Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.*

Cit. Rich Cook
public class TestDB extends TestCase {
private Connection dbConn;
protected void setUp() {
    dbConn = new Connection("oracle", 1521, "fred", "foobar");
    dbConn.connect();
}
protected void tearDown() {
    dbConn.disconnect();
    dbConn = null;
}
public void testAccountAccess() {
    // Uses dbConn
    xxx xxx xxxxxx xxx xxxxxxxxx;
}
}
Integration testing problem

Integrate multiple components implies to decide in which order classes and subsystems should be integrated and tested

**CITO Problem**
- *Class Integration Testing Order* Problem

**Solution:**
- Topological sort of dependency graph
Integration testing example
Testing in isolation offers strong benefits
- Test code that have not been written
- Test only a single method (behavior) without side effects from other objects

*Focused* Integration Testing (!!!)

Solutions?
- Stubs
-Mocks
-...
Testing in Isolation: example
```java
public class UserDAOStub implements UserDAO {
    public boolean saveUser(String name) {
        return true;
    }
}

public class MailerStub implements Mailer {
    private List<String> mails = new ArrayList<String>();

    public boolean sendMail(String to, String subject, String body) {
        mails.add(to);
        return true;
    }

    public List<String> getMails() {
        return mails;
    }
}

[Test]
@Test
public void verifyCreateUser() {
    UserManager manager = new UserManagerImpl();
    MailerStub mailer = new MailerStub();
    manager.setMailer(mailer);
    manager.setDAO(new UserDAOStub());
    manager.createUser("tester");
    assert mailer.getMails().size() == 1;
}
@Test
public void createUser() {
    // create the instance we'd like to test
    UserManager manager = new UserManagerImpl();
    // create the dependencies we'd like mocked
    Mock mailer = mock(Mailer.class);
    Mock dao = mock(UserDAO.class);
    // wire them up to our primary component, the user manager
    manager.setMailer((Mailer)mailer.proxy());
    manager.setDAO((UserDAO)dao.proxy());
    // specify expectations
    dao.saveUser() must return true;
    expect invocation dao.saveUser() with parameter "tester";
    dao.sendMail must return true;
    expect invocation dao.sendMail with parameter "tester"
    // invoke our method
    manager.createUser("tester");
    // verify that expectations have been met
    verifyExpectations();
}
Key Ideas

(Ignoring the specifics of codes)

- Mocks do not provide our own implementation of the components we'd like to swap in

- Main Difference:
  - Mocks test behavior and interactions between components
  - Stubs replace heavyweight process that are not relevant to a particular test with simple implementations
Naming Problems and Buzz Worlds

► Unfortunately, while two components are quite distinct, they're used interchangeably.
  ○ Example: spring-mock package

► If we want to be stricter in terms of naming, stub objects defined previously are
  ○ test doubles

► Test Doubles, Stubs, Mocks, Fake Objects... how can we work it out?
Q: How can we verify logic independently when code it depends on is unusable?

Q1: How we can avoid slow tests?

A: We replace a component on which the SUT depends with a “test-specific equivalent.”
Q: How can we verify logic independently when it depends on indirect inputs from other software components?

A: We replace a real objects with a test-specific object that feeds the desired inputs into the SUT
Q: How can we implement Behavior Verification for indirect outputs of the SUT?

A: We replace an object on which the SUT depends on with a test-specific object that verifies it is being used correctly by the SUT.
Mock Objects Observations

- Powerful way to implement Behavior Verification
  - while avoiding Test Code Duplication between similar tests.

- It works by delegating the job of verifying the indirect outputs of the SUT entirely to a Test Double.

- Important Note: Design for Mockability
  - Dependency Injection Pattern
Design for Mockability

Dependency Injection

```java
public void doWork1() {
    B b = B.getInstance();
    b.doSomething();
}

private B b;

public void setB(B b) {
    this.b = b;
}
```
Q: How much are the directions in which we could slice functionalities of the system under test?

A:
- Vertical Slicing
- Horizontal Slicing
Mock Libraries

► Two main design philosophy:
  ○ DSL Libraries
  ○ Record/Replay Models Libraries

► Record Replay Frameworks
  ○ First train mocks and then verify expectations

► DSL Frameworks
  ○ Domain Specific Languages
  ○ Specifications embedded in “Java” Code
Mocking with EasyMock

```java
import static org.easymock.EasyMock.*;

public class EasyMockUserManagerTest {
    @Test
    public void createUser() {
        // create the instance we'd like to test
        UserManager manager = new UserManagerImpl();
        UserDAO dao = createMock(UserDAO.class);
        Mailer mailer = createMock(Mailer.class);
        manager.setDAO(dao);
        manager.setMailer(mailer);
        // record expectations
        expect(dao.saveUser("tester")).andReturn(true);
        expect(mailer.sendMail(eq("tester"), (String)notNull(), (String)notNull())).andReturn(true);
        replay(dao, mailer);
        // invoke our method
        manager.createUser("tester");
        // verify that expectations have been met
        verify(mailer, dao);
    }
}
```
EasyMock Test

- Create Mock objects
  - Java Reflections API

- Record Expectation
  - expect methods

- Invoke Primary Test
  - replay method

- Verify Expectation
  - verify method
JMock Example

```java
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.jmock.Expectations;

@RunWith( JMock.class )
public class TestAccountServiceJMock
{
    private Mockery context = new JUnit4Mockery();
    private AccountManager mockAccountManager;
    @Before
    public void setUp()
    {
        UserDAO dao = context.mock(UserDAO.class);
        Mailer mailer = context.mock(Mailer.class);
    }
    @Test
    public void createUser()
    {
        UserManager manager = new UserManagerImpl();
        // Set Mocks
        UserDAO dao = createMock(UserDAO.class);
        Mailer mailer = createMock(Mailer.class);
        manager.setDAO(dao);
        manager.setMailer(mailer);
        // invoke our method
        manager.createUser("tester");
        context.checking( new Expectations() {
            { oneOf(dao).saveUser("tester");
                will(returnValue(true));
            oneOf(mailer).sendMail("tester", (String)nonnull(), (String)nonnull());
                will( returnvalue(true) );
        } } )
```
JMock features

- JMock syntax relies heavily on chained method calls
  - Sometimes difficult to decipher and debugger

- Common Pattern:
  
  ```java
  invocation-count (mockobject).method(arguments);
  inSequence(sequence-name);
  when(state-machine.is(state-name));
  will(action);
  then(state-machine.is(new-state name));
  ```
JMock Working Example
JMock features (intro)

- JMock previous versions required subclassing
  - Not so smart in testing
  - Now directly integrated with JUnit4
  - JMock tests requires more typing

- JMock API is extensible
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;

@RunWith(JMock.class)
public class TurtleDriverTest {
    private final Mockery context = new JUnit4Mockery();
    private final Turtle turtle = context.mock(Turtle.class);

    @Test public void goesAMinimumDistance() {
        final Turtle turtle2 = context.mock(Turtle.class, "turtle2");
        final TurtleDriver driver = new TurtleDriver(turtle1, turtle2); // set up context. checking(new Expectations() {{
            // expectations
            ignoring (turtle2);
            allowing (turtle). flashLEDs();
            oneOf (turtle). turn(45);
            oneOf (turtle). forward(with(greaterThan(20)));
            atLeast(1).of (turtle). stop();
        }});
        driver. goNext(45); // call the code
        assertTrue("driver has moved", driver. hasMoved()); // further assertions
    }
}
1. Test Fixture

Mockery represents the context

- Neighboring objects it will communicate with
- By convention the mockery is stored in an instance variable named context

@RunWith(JMock.class) annotation

JUnit4Mockery reports expectation failures as JUnit4 test failures
2. Create Mock Objects

```java
private final Turtle turtle = context.mock(Turtle.class);
final Turtle turtle2 = context.mock(Turtle.class, "turtle2");
```

- The tests has two mock turtles
  - The first is a field in the test class
  - The second is local to the test
- References (fields and Vars) have to be `final`
  - Accessible from Anonymous Expectations
- The second mock has a specified name
  - JMock enforces usage of names except for the first (default)
  - This makes failures reporting more clear
3. Tests with Expectations

A test sets up its expectations in one or more *expectation blocks*

- An expectation block can contain any number of expectations
- Expectation blocks can be interleaved with calls to the code under test.
3. Tests with Expectations

Expectations have the following structure:

- `invocation-count`
- `(mockobject).method(arguments);`
- `inSequence(sequence-name);`
- `when(state-machine.is(state-name));`
- `will(action);`
- `then(state-machine.is(new-state name));`

```java
context. checking(new Expectations() {{
  // expectations
  ignoring (turtle2);
  allowing (turtle). flashLEDs();
  oneOf (turtle). turn(45);
  oneOf (turtle). forward(with(greaterThan(20)));
  atLeast(1).of (turtle). stop();
}});
```
What's with the double braces?

```java
context.checking(new Expectations(){{
    oneOf(turtle).turn(45);
}});
```

- Anonymous subclass of `Expectations`
- Baroque structure to provide a scope for building up expectations
  - Collection of expectation components
  - Is an example of **Builder Pattern**
  - Improves code completion
What's with the double braces?

```java
context.checking(new Expectations(){
    oneOf(turtle).turn(45);
});
```
context.checking(new Expectations(){{
  ignoring (turtle2);
  allowing (turtle).flashLEDs();
  oneOf(turtle).turn(45);
}});

- **Expectations** describe the interactions that are **essential** to the protocol we're testing.
- **Allowances** support the interaction we're testing.
  - `ignoring()` clause says that we don't care about messages sent to `turtle2`.
  - `allowing()` clause matches any call to `flashLEDs` of `turtle`. 
context.checking(new Expectations(){
    ignoring (turtle2);
    allowing (turtle).flashLEDs();
    oneOf(turtle).turn(45);
});

- Distintion between *allowances* and *expectations* is not rigid

- **Rule of Thumb:**
  - *Allow queries; Expect Commands*

- **Why?**
  - Commands could have side effects;
  - Queries don't change the world.
Dependency injection issues?

► Too Many Dependencies

○ Ideas??

```java
public class RacingCar {
  private final Track track;
  private Tyres tyres;
  private Suspension suspension;
  private Wing frontWing;
  private Wing backWing;
  private double fuelLoad;
  private CarListener listener;
  private DrivingStrategy driver;

  public RacingCar(Track track, DrivingStrategy driver, Tyres tyres,
                   Suspension suspension, Wing frontWing, Wing backWing,
                   double fuelLoad, CarListener listener)
  {
    this.track = track;
    this.driver = driver;
    this.tyres = tyres;
    this.suspension = suspension;
    this.frontWing = frontWing;
    this.backWing = backWing;
    this.fuelLoad = fuelLoad;
    this.listener = listener;
  }
}
```
Dependency injection issues?

► Dependency injection for mockability

```java
public class RacingCar {
    private final Track track;
    private DrivingStrategy driver = DriverTypes.borderlineAggressiveDriving();
    private Tyres tyres = TyreTypes.mediumSlicks();
    private Suspension suspension = SuspensionTypes.mediumStiffness();
    private Wing frontWing = WingTypes.mediumDownforce();
    private Wing backWing = WingTypes.mediumDownforce();
    private double fuelLoad = 0.5;
    private CarListener listener = CarListener.NONE;

    public RacingCar(Track track) {
        this.track = track;
    }

    public void setSuspension(Suspension suspension) { [...] }
    public void setTyres(Tyres tyres) { [...] }
    public void setEngine(Engine engine) { [...] }
    public void setListener(CarListener listener) { [...] }
}
```
Expectations or ... ?

► Too Many Expectations

○ Ideas??

//Production code
public void adjudicateIfReady(ThirdParty thirdParty, Issue issue) {
    if (firstParty.isReady()) {
        Adjudicator adjudicator = organization.getAdjudicator(); //getter
        Case acase = adjudicator.findCase(firstParty, issue); // Lookup
        thirdParty.proceedWith(acase);
    } else
        thirdParty.adjourn();
}

//Test Code
@Test public void decidesCasesWhenFirstPartyIsReady() {
    context.checking(new Expectations(){{
        one(firstPart).isReady(); will(returnValue(true));
        one(organizer).getAdjudicator(); will(returnValue(adjudicator));
        one(adjudicator).findCase(firstParty, issue); will(returnValue(acase));
        one(thirdParty).proceedWith(acase);
    }});
    claimsProcessor.adjudicateIfReady(thirdParty, issue);
}
Too Many Expectations

- Ideas??

```java
//Production code
public void adjudicateIfReady(ThirdParty thirdParty, Issue issue) {
    if (firstParty.isReady()) {
        Adjudicator adjudicator = organization.getAdjudicator(); // getter
        Case acase = adjudicator.findCase(firstParty, issue); // Lookup
        thirdParty.proceedWith(acase);
    }
    else
        thirdParty.adjourn();
}
```

```java
//Refactored Test Code
@Test public void decidesCasesWhenFirstPartyIsReady() {
    context.checking(new Expectations(){{
        allowing(firstPart).isReady(); will(returnValue(true));
        allowing(organizer).getAdjudicator(); will(returnValue(adjudicator));
        allowing(adjudicator).findCase(firstParty, issue); will(returnValue(acase));
        one(thirdParty).proceedWith(acase);
    }});
    claimsProcessor.adjudicateIfReady(thirdParty, issue);
}
```
Expectations or ...?

► Too Many Expectations
  o Ideas??

//Refactored Production Code
public void adjudicateIfReady(ThirdParty thirdParty, Issue issue) {
  if (firstParty.isReady())
    thirdParty.startAdjudication(organization, firstParty, issue);
  else
    thirdParty.adjourn();
}

//Refactored Test Code
@Test public void decidesCasesWhenFirstPartyIsReady() {
  context.checking(new Expectations(){{
    allowing(firstPart).isReady(); will(returnValue(true));
    allowing(organizer).getAdjudicator(); will(returnValue(adjudicator));
    allowing(adjudicator).findCase(firstParty, issue); will(returnValue(acase));
    one(thirdParty).proceedWith(acase);
  }});
  claimsProcessor.adjudicateIfReady(thirdParty, issue);
}
Let's think about the development process of this example:

Q: Does make sense to write tests before writing production code?

A: Two Keywords

- TDD: Test Driven Development
- Test-first Programming
Mocks and Stubs Pitfall

► False sense of security

► Maintenance Overhead
  ○ Keep mocks up2date with test and producton code

  ○ Example:
    ● UserManager won't send mail no more

  ○ Maintenance headaches in making test code to run
References

- Professional Java JDK 5 Edition
  - Richardson et. al., Wrox Publications 2006

- xUnit Test Patterns
  - G. Meszaros, Addison Wesley 2006

- Next Generation Java Testing
  - Beust, Suleiman, Addison Wesley 2007

- JUnit in Action, 2nd Ed.
  - Massol et al., Manning Pubs 2009

- Python Testing
  - Arbuckle Daniel, Packt Publishing 2010