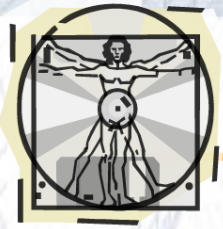

# SENG 637
# Dependability and Reliability of Software Systems

## Chapter 3: Criteria-Based Test Design
## Unit Testing

**Department of Electrical & Software Engineering, University of Calgary**

B.H. Far (far@ucalgary.ca)

http://people.ucalgary.ca/~far



# Contents

- Introduction to criteria-based test design
- Unit testing
- Automated test execution
  - JUnit
- Dependencies
  - Stubbing – mocking

# Types of Testing

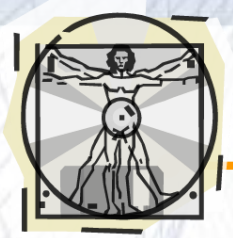
- Functional Testing: testing functional requirements
  - **Functional testing is checking correct functionality of a system**

Our focus

- Non-functional Testing: testing non-functional requirement,

  e.g. performance, reliability, security

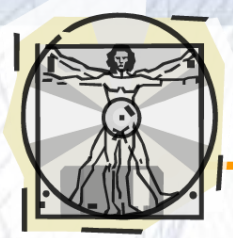
Later



# Criteria Based Testing

- What is a criterion anyway?
- A testing related concept you build your test cases around it systematically and efficiently

**Partitioning**
**Coverage (i.e. statement, decision, condition, path)**
**Mutation**
**Acceptance**
**etc.**



# Testing Approaches

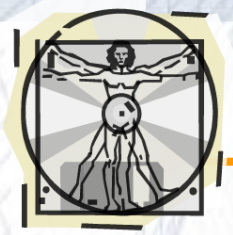
- Exhaustive testing
  - Pros: guarantee
  - Cons: infeasible in most real world applications
- Random testing (ad-hoc, exploratory)
  - Pros: cheap
  - Cons: inefficient in many situations
- Partitioning
  - Pros: effective in theory
  - Cons: practical solutions?

Done Already


UNIVERSITY OF CALGARY

# Section 2


## Unit Testing

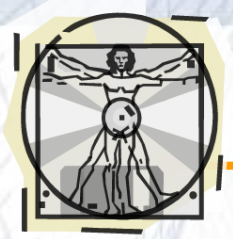


# Unit Testing Scenario

- Grab the code you want to test (SUT),
  - e.g. **`add(x,y)`**
- Write a program that calls SUT and executes it with the setup value of its arguments and spits out the execution result,
  - e.g., **`x=2`** and **`y=3`**, **`result=5`**
- Check if the result meets the expected result or not

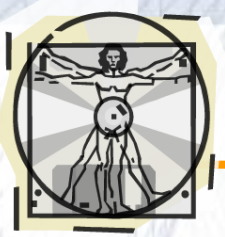
**How realistic is this scenario?**
**What difficulties it may cause?**

**How to automate this scenario?**



# Unit Testing

- **Scope:** Ensure that each unit (i.e. **class**, subsystem) has been implemented correctly
  - Looking for faults in a subsystem in isolation
  - Generally a “subsystem” means a particular class or object
- **Method:**
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run
  - Each method looks for particular results and passes / fails
- Often (not necessarily) based on **white-box** testing

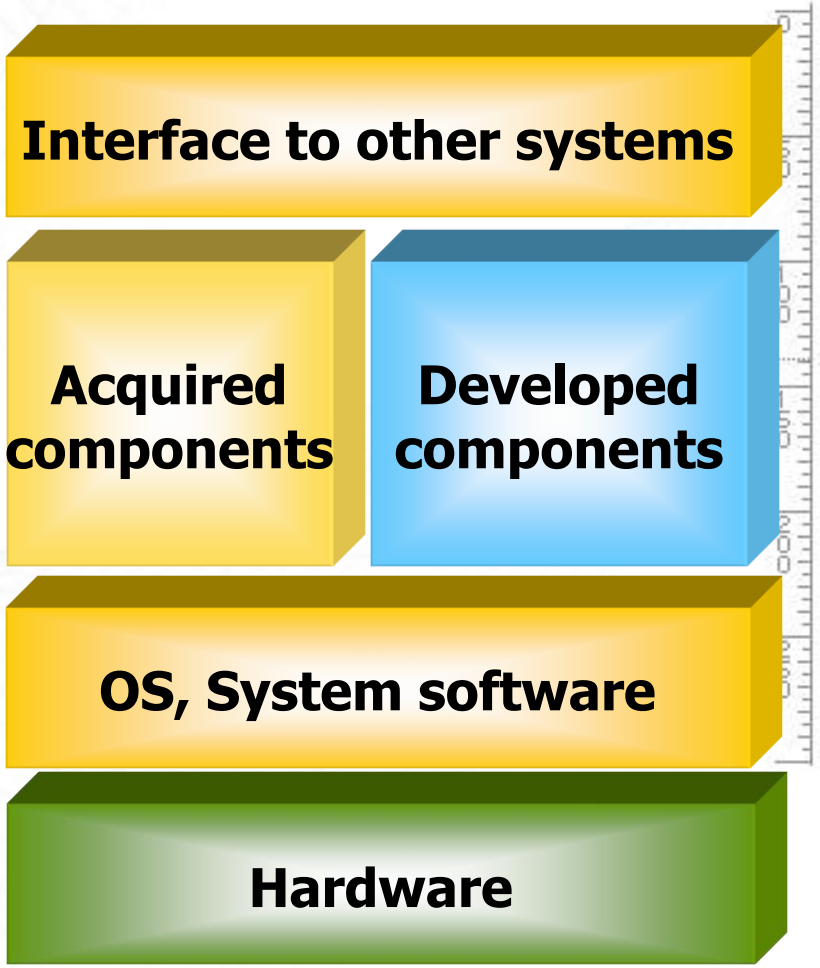


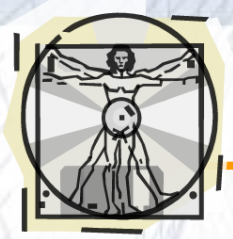
# Unit Testing: Characteristics

- A unit test must only test **one specific unit of functionality** (e.g. **class** or **method**)
- It is **fast**
  - Less than 10-minute build
- It does not access a **database** or **the file system**
- It does not communicate via a **network**
- It does not require any **special setup** to the system environment such as modifying a configuration file
- It leaves the system and the system environment in **the same state** that it had **prior to the test**
- Focus on **developed components** only

# Unit Testing: What to Test?

- Focus on developed components and surroundings
- One sample from each equivalent class of input data
  - Works for both white box and black box tests
- Invalid data
  - Null, missing, empty, ...
- Boundaries
  - Valid Input data
  - Boundary analysis
    - One sample from each class





# Unit Testing: When to Test?

- Waterfall
  - Typically written **as (or after)** the system has been developed
- TDD/Agile
  - Each unit test is written **before** or **during** the corresponding code being written

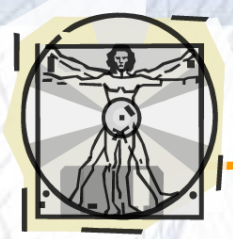
If you find it difficult to write unit tests, it is likely an indication of poor requirements, design or coding practices


UNIVERSITY OF CALGARY

# Section 3

## Automated Test Execution

## JUnit



# Test Automation Framework

- A set of assumptions, concepts, and mainly tools, that support test automation
- Some unit testing frameworks

.NET
- NUnit
- Pex

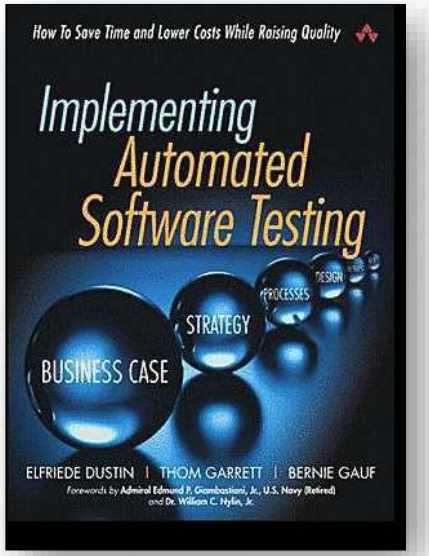
PHP
- PHPUnit

JavaScript
- JSUnit
- Jasmine (not Xunit)
- JasUnit

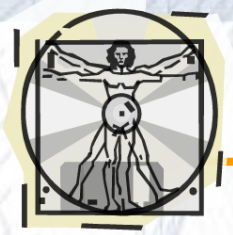
Java
- JUnit
- TestNG

# JUnit

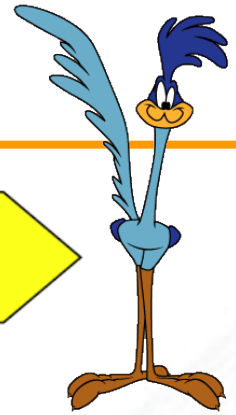
- Java testing framework used to write and run tests
- Open source (junit.org)
- Helps test execution automation
  - Good for large scale testing
  - Repeatable/Automated test procedure
  - Massive input data possible
  - Widely used in industry




- Can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse



# JUnit in a Nutshell



(3) Test Runner

test runner

test suite

another unit test

test case (for one method)

another test case

another unit test

another test case

another test case

another test case

unit test (for one class)

test case (for one method)

another test case

test fixture

Java code of your tests

(2) Test Suite

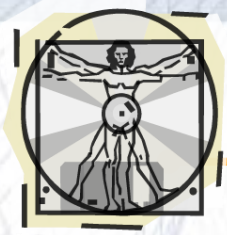
Need this to run your tests on SUT

Need this to setup running env for your tests

(1) SUT

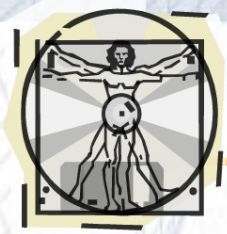
# JUnit: Terminology

- **Test runner**: A test runner is an executable program that runs tests implemented using JUnit framework and reports test results (in command line mode) or update the display (in IDE)
- **Test fixtures**: A test fixture (context) is the set of preconditions or state needed to run a test
- **Test case**: A test case is the most elemental class. All unit tests are inherited from here
- **Test suites**: A test suite is a set of tests that all share the same fixture. The order of the tests shouldn't matter



# JUnit: Terminology

- **Test drivers**: Modules that substiutue temporarily a calling module and give the same output as that of the actual one
- **Test execution**: The execution of an individual unit test proceeds as follows: using
  - **Annotations** @Before @Test @After --or--
  - setup(); Body of test; teardown();
- **Test result formatter**: A test runner produces results in one or more output formats, e.g. XML
- **Assertions**: An assertion is a function or macro that verifies the behavior (or the state) of the unit under test, e.g. *true*, *false*, *null*



# JUnit: Which Version to Use?

- Junit 4.x or 5.x?
- The newer the better but we have to learn the differences
  - Architectural difference
  - JDK compatibility
  - Assertion
  - Annotation
  - Assumptions
  - Parameterized test

**Many older systems test suites are still in JUnit 3 so we need to be familiar with them, too**

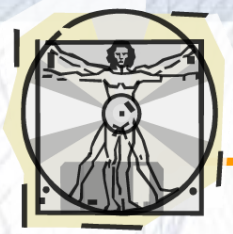
**We will use either JUnit 4 or 5 in the exercises and Assignments**

# JUnit 4 vs. JUnit 5

- JUnit 5 (Sept. 2017) requires Java 8 or higher
- JUnit 4 (v4.13 Nov. 2018) requires Java 5 or higher, it uses a lot from Java 5 annotations, generics, and static import features
- The JUnit 3.x version can work with JDK 1.2+ and any higher versions
- JUnit 5 is backward compatible with JUnit 4 /3 (i.e. JUnit 3 or 4 tests work in ***JUnit Vintage***)

**JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**

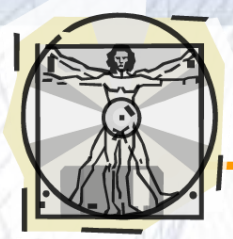
https://howtoprogram.xyz/2016/08/10/junit-5-vs-junit-4/



# JUnit 4 vs. JUnit 5

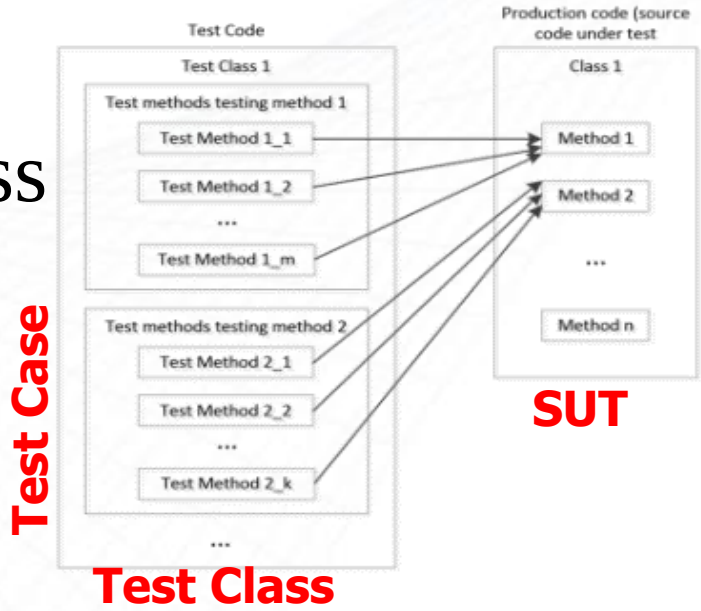
- Defining tests
  - In the JUnit 4 and 5, test classes are no longer required to extend junit.framework.TestCase class
  - Test names are no longer need required to follow the **testXXX** pattern, i.e. do not prefix the test method with "test"
  - Instead, in JUnit4/5, any public class with a zero-argument public constructor can act as a test class, any methods which are considered as a test method should be annotated with the **@Test annotation**
  - JUnit4/5 use one of the **assert()** methods
  - JUnit4 runs the test using JUnit4TestAdapter

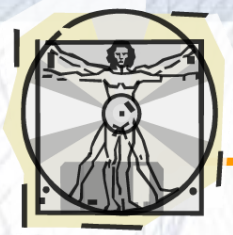
**We will use either of JUnit 4 or 5 in the exercises and Labs**



# Anatomy of Junit Testing

- 1. Get the SUT
- 2. Create test case class(es)
  - Create tests for each method in the SUT
- 3. Create test suite class
- 4. Create test runner class
- 5. Compile and run





# A JUnit Test Case

Method to be tested

```
public class Calc
{
  static public int add (int a, int b)
  {
    return a + b;
  }
}
```

Annotation

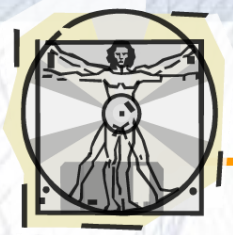
Assertion

Test values

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

public class CalcTest
{
  @Test public void testAdd()
  {
    assertTrue ("Calc sum incorrect",
                5 == Calc.add (2, 3));
  }
}
```

Printed if assert fails

Expected output

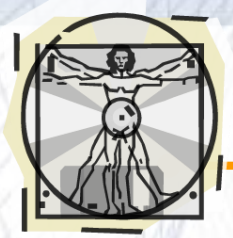


# A JUnit Test Class

- A method with **`@Test`** is flagged as a JUnit test case
- All **`@Test`** methods run when JUnit runs your test class

Standard imports for all JUnit classes

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

public class TestJunit1 {
    ...                 // some code referring to the SUT

    @Test
    public void TestMethod1() { // a test case method
        ...             // put assertions here
    }
}
```



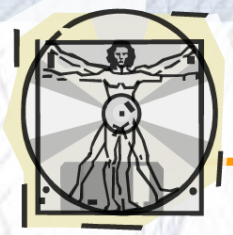
# A JUnit Test Suite

- In JUnit 3, the way of constructing sets of your tests involved writing a **`suite()`** method and manually inserting all tests that you want to be present in the suite
- In JUnit 4-5, the suite construction is done by two annotations: **`@RunWith`** and **`@SuiteClasses`** annotations

**Optional for running tests in command line**

**Used by TestRunner to compile and run tests**

```
package my.package.tests;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestJunit1.class,
    TestJunit2.class,
    ...
})
public class JunitTestSuite {}
```



# A JUnit Test Runner

- Create a java class file named **TestRunner.java**

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(JunitTestSuite.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
    System.out.println(result.wasSuccessful());
  }
}
```

TestSuite you created

- Compile all the java classes using **`javac`**
- Finally run the **`TestRunner`**

# JUnit 3 vs. JUnit 4 / 5

- **Defining test case:**

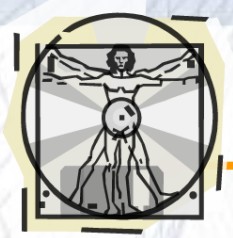
Extending TestCase is required in JUnit 3

```
1  import junit.framework.Assert;
2  import junit.framework.TestCase;
3
4  public class TournamentTest extends TestCase{
5    Tournament tournament;
6
7    public void setUp() throws Exception {
8      System.out.println("Setting up ...");
9      tournament = new Tournament(100, 60);
10   }
11
12   public void tearDown() throws Exception {
13     System.out.println("Tearing down ...");
14     tournament = null;
15   }
16
17   public void testGetBestTeam() {
18     Assert.assertNotNull(tournament);
19
20     Team team = tournament.getBestTeam();
21     Assert.assertNotNull(team);
22     Assert.assertEquals(team.getName(), "Test1");
23   }
24 }
```

JUnit 3

```
1  import junit.framework.Assert;
2
3  import org.junit.After;
4  import org.junit.Before;
5  import org.junit.Test;
6
7  public class TournamentTest {
8    Tournament tournament;
9
10   @Before
11   public void init() throws Exception {
12     System.out.println("Setting up ...");
13     tournament = new Tournament(100, 60);
14   }
15
16   @After
17   public void destroy() throws Exception {
18     System.out.println("Tearing down ...");
19     tournament = null;
20   }
21
22   @Test
23   public void testGetBestTeam() {
24     Assert.assertNotNull(tournament);
25
26     Team team = tournament.getBestTeam();
27     Assert.assertNotNull(team);
28     Assert.assertEquals(team.getName(), "Test1");
29   }
30 }
```

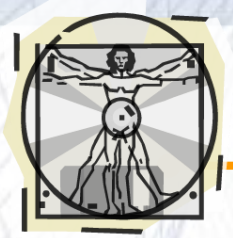
JUnit 4



# JUnit - Annotations

- No annotations in JUnit 3

| Feature | JUnit 3.x | JUnit 4.x |
|---|---|---|
| test annotation | testXXX pattern | @Test |
| run before the first test method in the current class is invoked | None | @BeforeClass |
| run after all the test methods in the current class have been run | None | @AfterClass |
| run before each test method | override setUp() | @Before |
| run after each test method | override tearDown() | @After |
| ignore test | Comment out or remove code | @ignore |
| expected exception | catch exception assert success | @Test(expected = ArithmeticException.class) |
| timeout | None | @Test(timeout = 1000) |



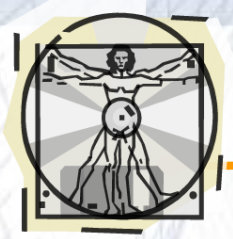
# JUnit - Annotations

- Additional annotations in JUnit 5

| Features | JUnit 5 | JUnit 4 |
|---|---|---|
| **Declares a test method** | @Test | @Test |
| **Denotes that the annotated method will be executed before all test methods in the current class** | @BeforeAll | @BeforeClass |
| **Denotes that the annotated method will be executed after all test methods in the current class** | @AfterAll | @AfterClass |
| **Denotes that the annotated method will be executed before each test method** | @BeforeEach | @Before |
| **Denotes that the annotated method will be executed after each test method** | @AfterEach | @After |
| **Disable a test method or a test class** | @Disable | @Ignore |
| **Denotes a method is a test factory for dynamic tests in JUnit 5** | @TestFactory | N/A |
| **Denotes that the annotated class is a nested, non-static test class** | @Nested | N/A |
| **Declare tags for filtering tests** | @Tag | @Category |
| **Register custom extensions in JUnit 5** | @ExtendWith | N/A |
| **Repeated Tests in JUnit 5** | @RepeatedTest | N/A |

# JUnit - Ignoring a Test

- Ignoring a test
  - In JUnit 3.x comment out the whole test method or rename the method
  - In JUnit 4.x, use @Ignore
  - In JUnit 5, use @Disable



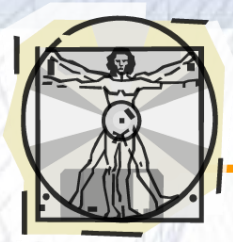
# JUnit - Timeout

- In JUnit 3 Not supported
- In JUnit 4, 5 use timeout parameter added to @Test annotation
  - With timeout parameter, you can specify a value (in milliseconds) that you expect to be the upper limit of the time you spend executing your test

```
@Test(timeout = 1000)
```

  - or

```
@Rule
public Timeout globalTimeout = Timeout.seconds(2);
```



# JUnit - Exception

- To catch (expected) exceptions thrown by JUnit

```
@Test (expected = Exception.class)
```

- Example

```
@Test (expected = NotFoundException.class)
public void example1() throws NotFoundException {
find("something");
// ... this line will never be reached
// when the test is passing
}
```

# JUnit and Eclipse

- JUnit works together with the Eclipse
- If Junit is not included in your IDE
  - To add JUnit to an Eclipse project, click:
  - Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish

Junit home page: http://junit.org/

http://junit.org/junit4/javadoc/latest/index.html

http://www.vogella.com/tutorials/JUnit/article.html
(Many more tutorial videos are available)

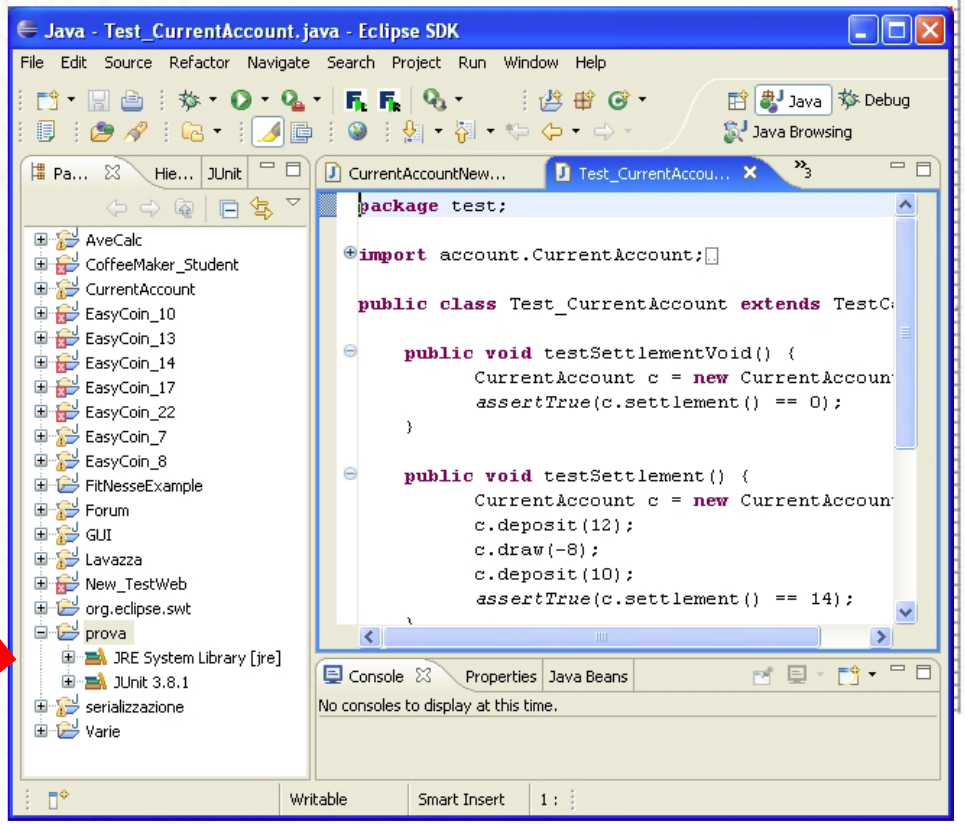

Junit5 User Guide (in D2L)

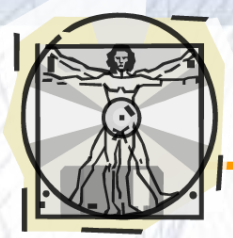
# JUnit in Eclipse - Setup

- In Eclipse
  - Create a new project
  - Open project's property window (File → Properties)
  - Select: Java build path

Select: libraries

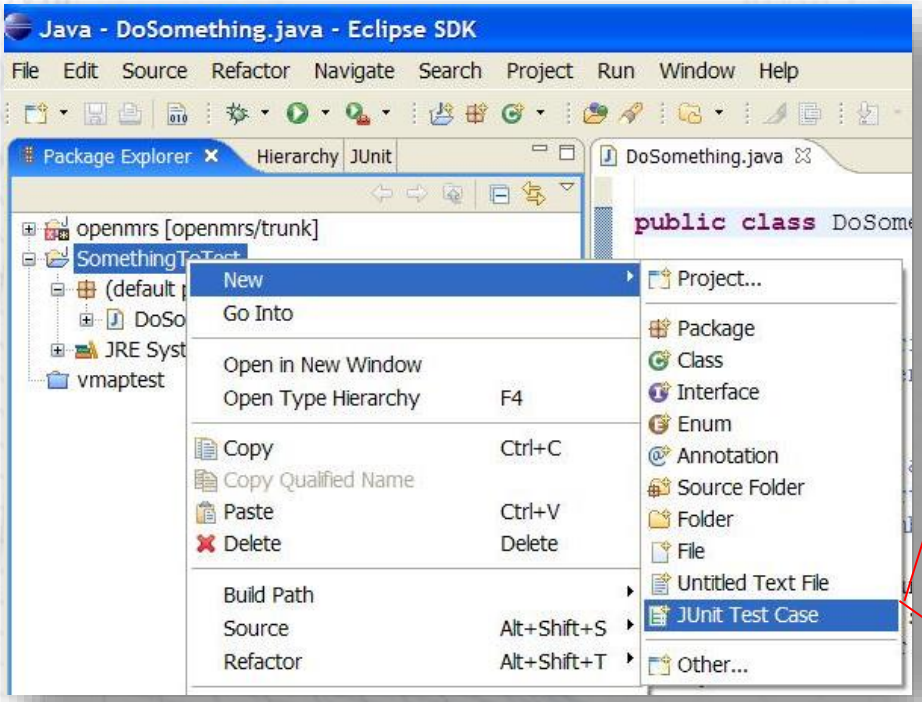
- Add Library
- Select Junit
  - Select the type 4.x or 5.x

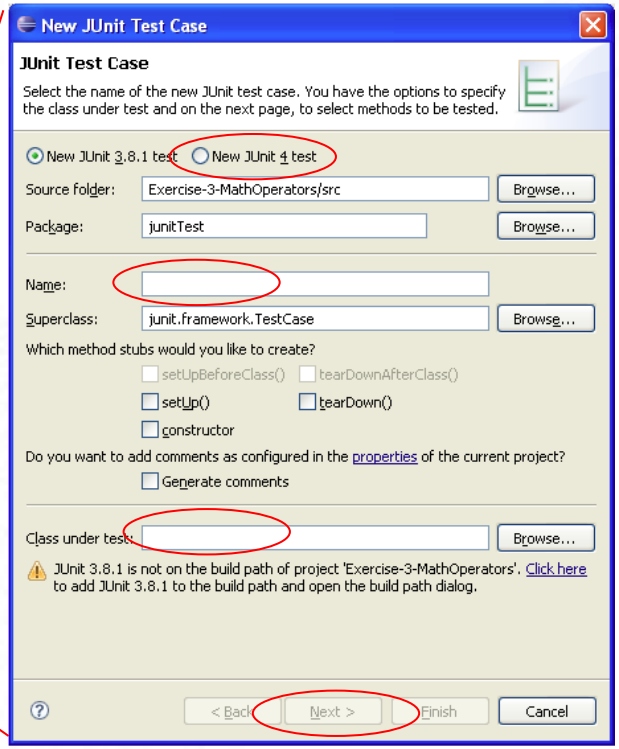




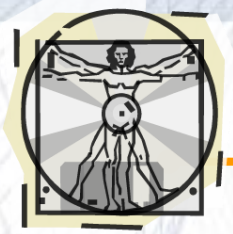
# Creating a Test Case

- Right-click a file and choose New → Test Case
- Or click File → New → JUnit Test Case



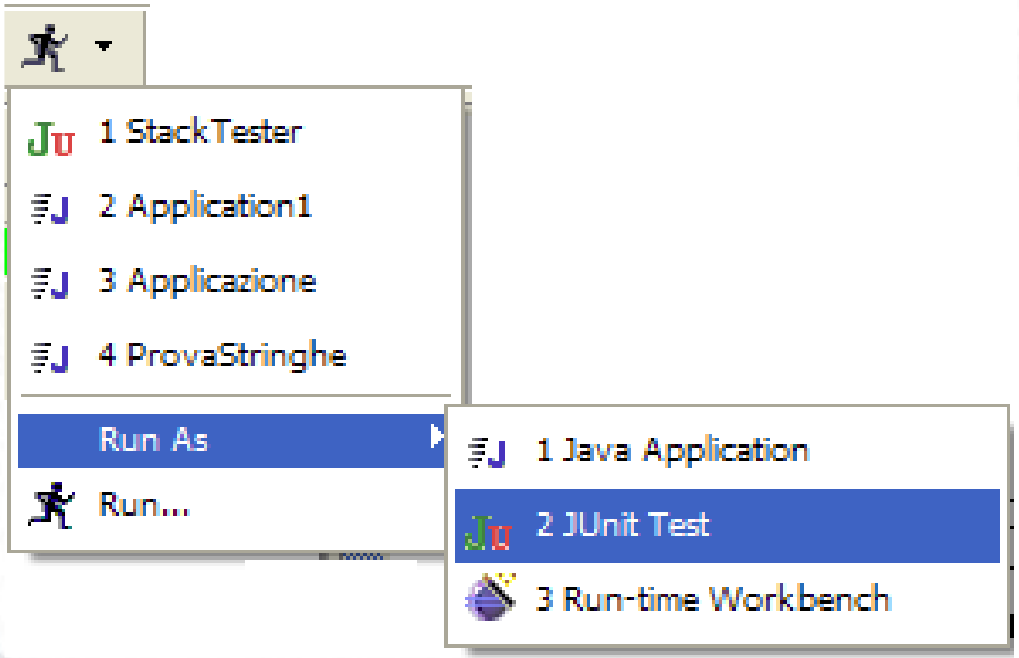



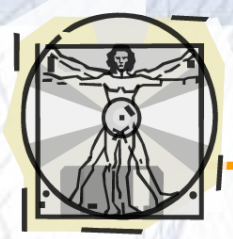

Next → create test case



# Run a JUnit Test

- Run
  - Run As
    - Junit Test





# JUnit in other IDE

- JUnit in Maven?

  https://maven.apache.org/index.html

- JUnit in Visual Studio?

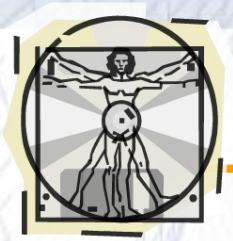
  https://blogs.msdn.microsoft.com/visualstudio/2017/12/01/announcing-junit-support-for-visual-studio-code/

- JUnit in Ant?

  https://ant.apache.org/manual/Tasks/junitlauncher.html

- Junit for Android

  https://developer.android.com/training/testing/unit-testing/local-unit-tests

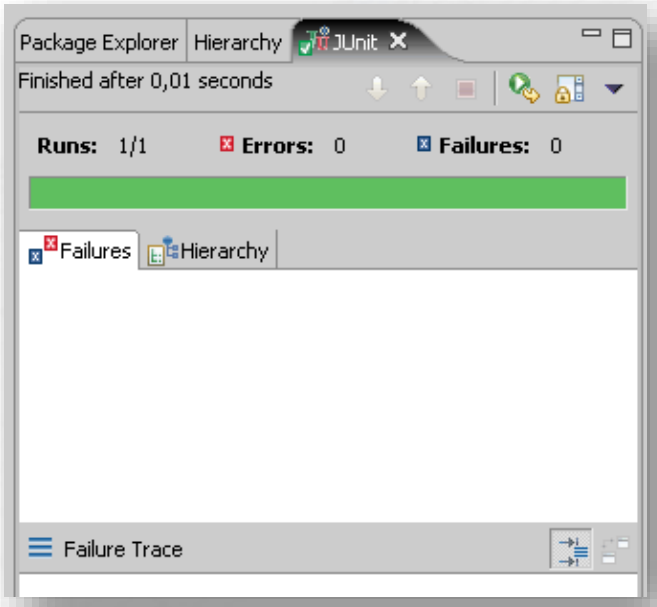


# Running a Test

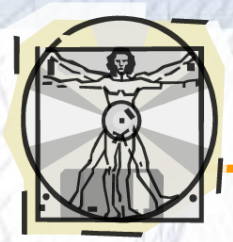
- Right click it in the Eclipse Package Explorer at left; choose:

  **Run As → JUnit Test**


- The JUnit bar will show **green** if all tests pass, **red** if any fail

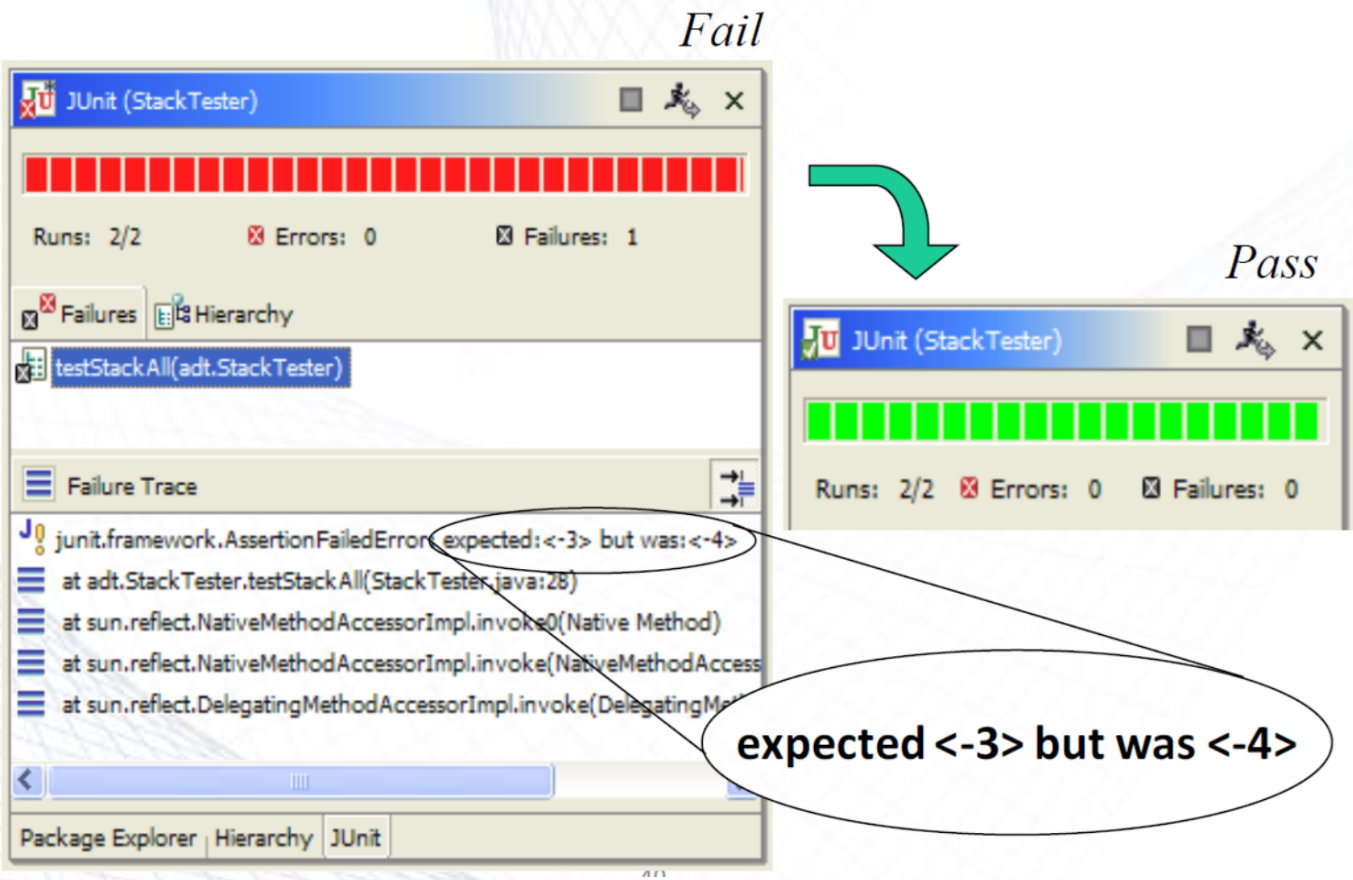


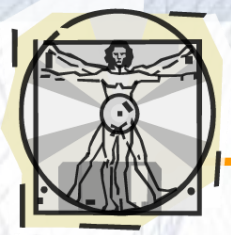

- The Failure Trace shows which tests failed, if any, and why



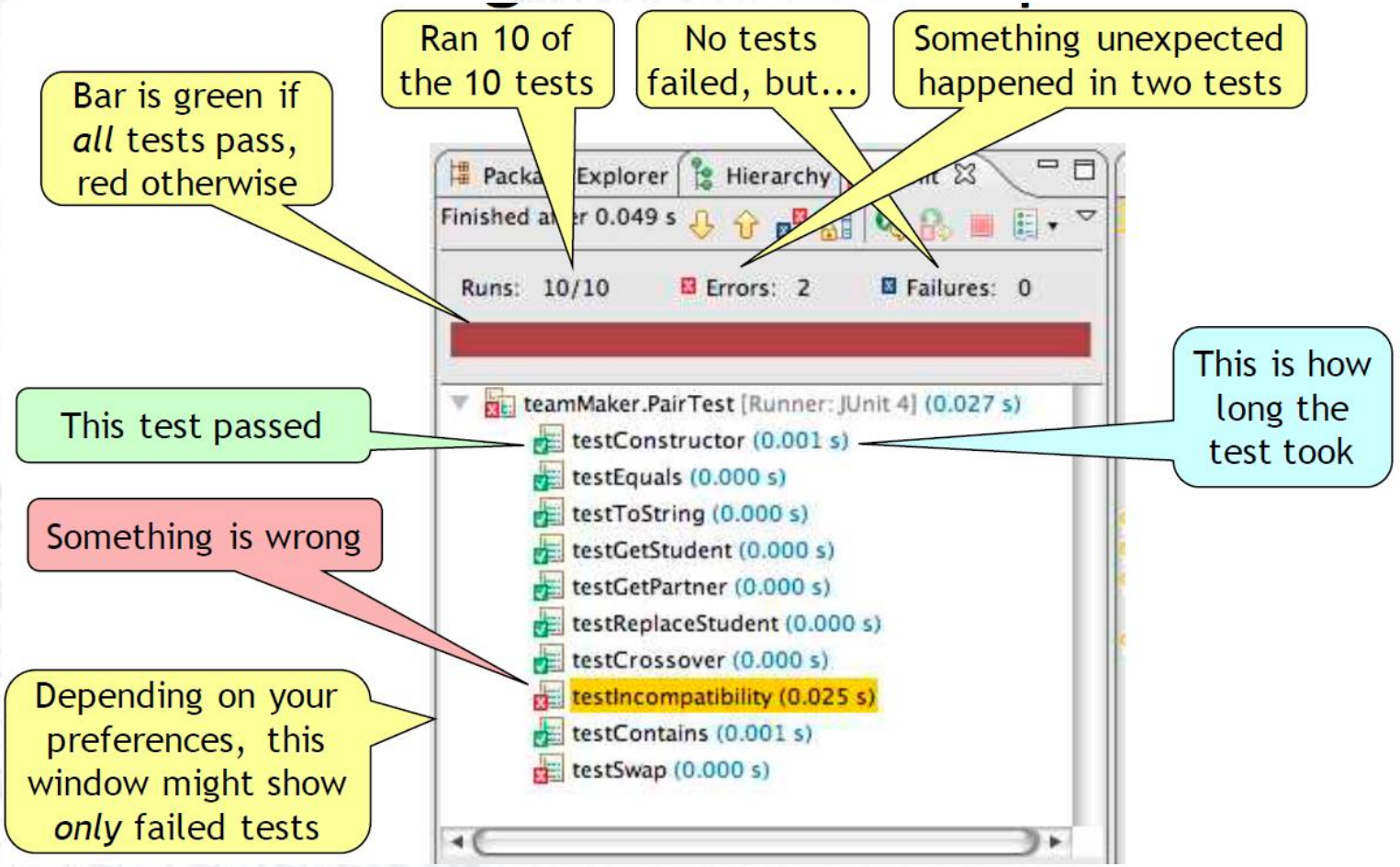
# Running a Test

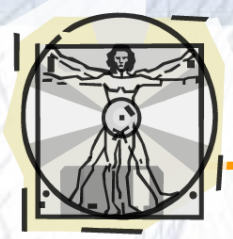
- Examine JUnit red bar





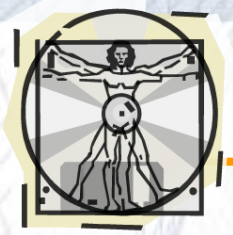
# Viewing Results in Eclipse





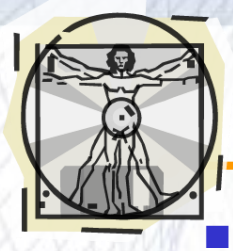
# Anatomy of a Test Case

- JUnit's testing **pattern**:
  - Set up **preconditions** ← **@Before method**
  - **Exercise** functionality being tested ← **@Test method**
  - Check **postconditions** ← **@After method**



# JUnit Test Fixtures

- A test fixture is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (*prefix* values)
  - Reset values (*postfix* values)
- Different tests can use the objects without sharing the state
- Objects used in test fixtures should be declared as instance variables
- They should be initialized in a @Before method
- Can be deallocated or reset in an @After method



# Test Case

- Setup (@Before, @BeforeClass, @BeforeAll)
- Exercise (@Test, calling the code from SUT)
- Verify (@Test, checking the result returned from SUT)
- Tear down (@After, @AfterClass, @AfterAll)

**Setup**

**Exercise**
**Verify**

**TearDown**

```
@Before
public  void setUp() {
    System.out.println("setUp");
    app = new App();
}

@Test
public void testIsNotEqual() {
    int eqValue = app.isEqual("Hi","Hello");
    assertEquals(1, eqValue);
}

@After
public void tearDown(){
    System.out.println("tearDown");
    app = null;
}
```

**Assertion in each @Test method**

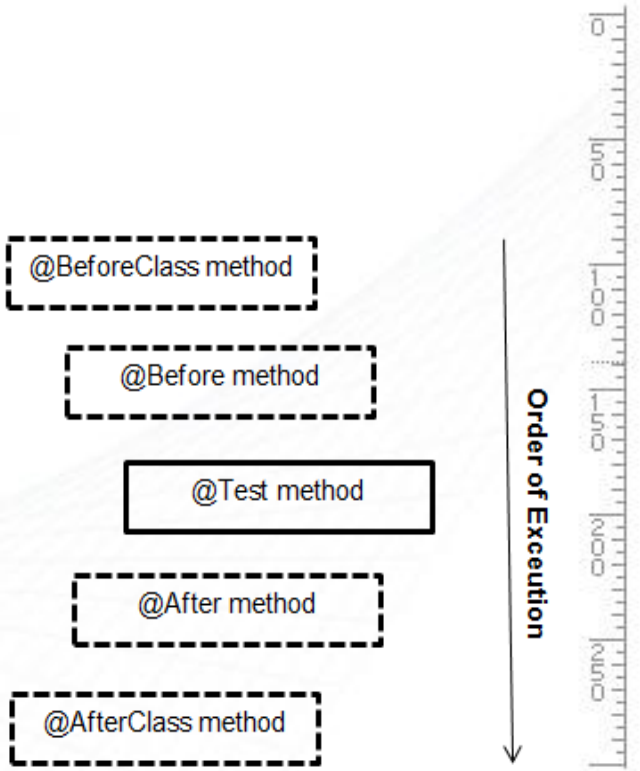
# Setup and Teardown

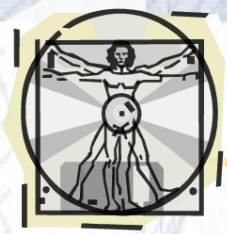
```
@Before
public void name() { ... }
@After
public void name() { ... }
```

Methods to run before/after each test case method is called

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

Methods to run once before/after the entire test class runs





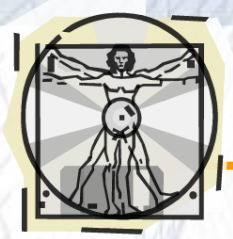
# JUnit assert*() Methods

| | |
|---|---|
| `assertTrue(`**"message", condition**`)` | fails if the boolean condition is `false` |
| `assertFalse(`**"message", condition**`)` | fails if the boolean condition is `true` |
| `assertEquals(`**expected, actual**`)` | fails if the values are not equal |
| `assertSame(`**expected, actual**`)` | fails if the values are not the same (by ==) |
| `assertNotSame(`**expected, actual**`)` | fails if the values *are* the same (by ==) |
| `assertNull(`**"message", value**`)` | fails if the given value is *not* `null` |
| `assertNotNull(`**"message", value**`)` | fails if the given value is `null` |
| `fail(`**"message"**`)` | causes current test to immediately fail |

Each method can pass a string **message** to display/log if it fails:

e.g. `assertEquals(`**"message", expected, actual**`)`

**Complete list:** http://junit.sourceforge.net/javadoc/org/junit/Assert.html

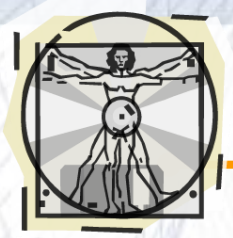


# Assertion - Exercise

- What's the actual use of ‘fail()’ assertion?
  - Mark a test that is incomplete, so it fails and warns you until you can finish it
  - Or force to throw an exception
- Why there is no ‘pass()’ assertion?
  - As long as the test doesn't throw an exception, it passes, so success is the default value

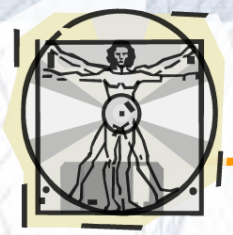
# Assert Method Messages

- Messages **helps documents** the tests
  - Messages provide **additional information when reading failure logs**
- Each assert method has an equivalent version that **does not take a message** – however, this usage is usually not recommended



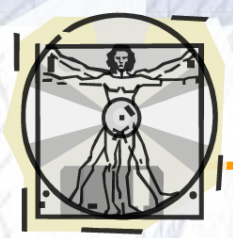
# Assert per Test Case

- Tests cases should normally not contain a large number of ***assert*** statements
  - JUnit **stops processing** a test method as soon as **one test fails**
- Including only one assert per test method leads to **setting up** the same test infrastructure **repeatedly**
  - Can define a **common test infrastructure** in **Before** and **After** methods in the test class



# Test Execution Order

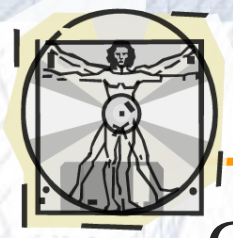
- Junit does not follow the given order of test methods in the test class, for execution
- Tests should be self-contained and not care about each other
- Things to avoid in tests
  - Constrained test order:
    - Test A must run before Test B
    - Tests call each other: Test A calls Test B's method (calling a shared helper is OK, though)



# Test Execution Order

```
import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runners.MethodSorters;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class TestMethodOrder {

    @Test
    public void testA() {
        System.out.println("first");
    }
    @Test
    public void testB() {
        System.out.println("second");
    }
    @Test
    public void testC() {
        System.out.println("third");
    }
}
```

- Test execution order
- Don’t assume any fixed order

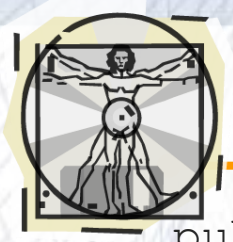


# JUnit Exercise: Black Box

Given a **`Date`** class with the following methods:

```
public Date(int year, int month, int day)
public Date()                          //today
public int getDay(), getMonth(), getYear()
public void addDays(int days) //advance by days
public int daysInMonth()
public String dayOfWeek()      //e.g. Sunday
public boolean equals(Object o)
public boolean isLeapYear()
public void nextDay()          //advance by 1 day
public String toString()
```

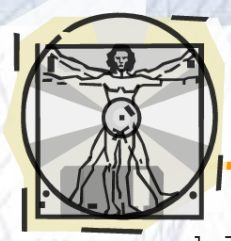
- Come up with unit tests to check the following:
  - That no **`Date`** object can ever get into an invalid state
  - That the **`addDays`** method works properly ← **Our focus**



# What's Wrong with This?

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);          // Advance by 4 days
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);  // February
        assertEquals(d.getDay(), 19);   // 19th
    }
```

**What about Leap year?**

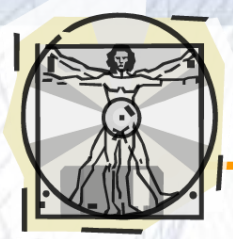


# Well Structured Assertions

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());   // NOTE: expected
        assertEquals(2, d.getMonth());     // value should
        assertEquals(19, d.getDay());      // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }   // test cases should usually have messages explaining
}       // what is being checked, for better failure output
```

# Naming Test Cases

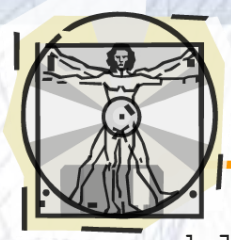
```
public class DateTest {
    @Test
    public void test_1 () {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, actual);
    }
    // give test case methods really long descriptive names
    @Test
    public void test_2() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```



# What's Wrong with This?

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals(
            "should have gotten " + expected + "\n" +
            " but instead got " + actual\n",
            expected, actual);
    }
    ...
}
```

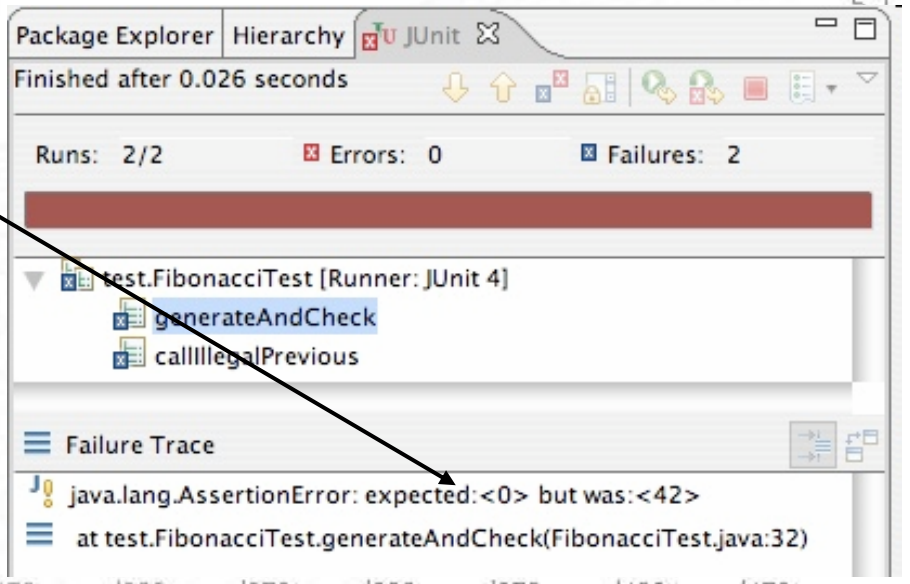
JUnit will already show the expected and actual values in its output and no need to repeat them in the assertion message

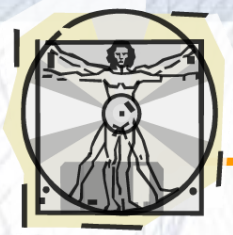


# Good Assertion Messages

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...
}
```

```
// JUnit will already show
// the expected and actual
// values in its output;
//
// don't need to repeat them
// in the assertion message
```



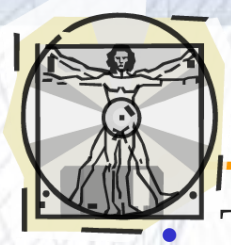


# Annotations - Pervasive Timeouts

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth_1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, d);
    }


    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_wrapToNextMonth_2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }

    // It is good to have a timeout for every test case
    // so it can't lead to an infinite loop;
    // good to set a default, too
    private static final int DEFAULT_TIMEOUT = 2000;
}
```

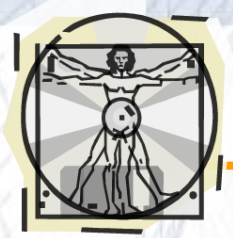
# What's Wrong with This?

```
public class DateTest {
    // test every day of the year
    @Test(timeout = 10000)
    public void tortureTest() {
        Date date = new Date(2050, 1, 1);
        int month = 1;
        int day = 1;
        for (int i = 1; i < 365; i++) {
            date.addDays(1);
            if (day < DAYS_PER_MONTH[month]) {day++;}
            else                             {month++; day=1;}
            assertEquals(new Date(2050, month, day), date);
        }
    }
    private static final int[] DAYS_PER_MONTH = {
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    }; // Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
}
```



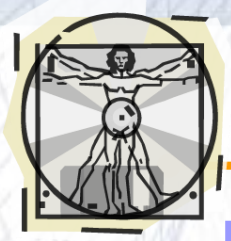
# Best Practice in Unit Testing

- Test one thing at a time per test method
  - 10 small tests are much better than 1 test 10x as large
- Each test method should have few (likely 1) assert statements
  - If you assert many things, the first that fails stops the test
  - You won't know whether a later assertion would have failed
- Tests should avoid logic
  - minimize `if/else`, `loops`, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it
- Torture tests are okay, but only *in addition to* simple tests



# Data-Driven Tests

- Problem: Testing a function multiple times with similar values
  - How to avoid test code bloat?
- Simple example: Adding two numbers
  - Adding a given pair of numbers is just like adding any other pair
  - You really only want to write one test
- Data-driven unit tests call a constructor for each collection of test values
  - Same tests are then run on each set of data values
  - Collection of data values defined by method tagged with @Parameters annotation



# Data-Driven Test Example

```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

@RunWith (Parameterized.class)
public class DataDrivenCalcTest
{  public int a, b, sum;

   public DataDrivenCalcTest (int v1, int v2, int expected)
   { this.a = v1; this.b = v2; this.sum = expected; }

  @Parameters public static Collection<Object[]> parameters()
  { return Arrays.asList (new Object [][] {{1, 1, 2}, {2, 3, 5}}); }

  @Test public void additionTest()
  { assertTrue ("Addition Test", sum == Calc.add (a, b)); }
}
```

Constructor is called for each triple of values

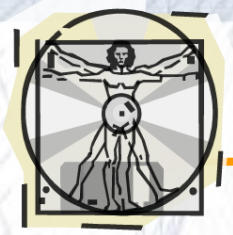
Test 1
Test values: 1, 1
Expected: 2

Test 2
Test values: 2, 3
Expected: 5

Test method

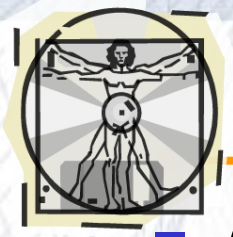
# JUnit - Parameterized Tests

- In JUnit 4.x, use @RunWith annotation to define a test runner to use
- Another test runner that's bundled with the JUnit distribution is the **parameterized** test runner
- Parameterized tests allows running the same test repetitively using different input test data
- JUnit 5 supports parameterized tests by default



# JUnit - Parameterized Tests

- Five steps to create a parameterized test:
  1. Annotate test class with `@RunWith(Parameterized.class)`
  2. Create a public static method annotated with `@Parameters` that returns a collection of objects (as Array) as test data set
  3. Create a public constructor that takes in what is equivalent to one "row" of test data
  4. Create an instance variable for each "column" of test data
  5. Create your test case(s) using the instance variables as the source of the test data



# Parameterized Class

- @RunWith(Parameterized.class)

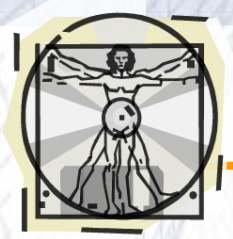
```
@BeforeClass
public static void setUpBeforeClass() throws Exception {
    calc = new calculatorService();
}

public calculatorServiceTest_round_parametric(double expected, double number, int digits){
    this.expected = expected;
    this.number = number;
    this.digits = digits;
}

@Parameters
public static Collection<Object[]> testData(){
    Object[][] data = new Object[][] { {1.12346,1.1234567890,5},{1,1.1234567890,0},{1.12346,1.1234567890,6}};
    return Arrays.asList(data);
}

@Test
public void testAdd() {
    double number = this.number;
    int digits = this.digits;
    double actualResult = calc.round(number, digits);
    double expectedResult = this.expected;
    assertEquals("The result of testing the largest decimal digists", expectedResult, actualResult,0);
}

@AfterClass
public static void tearDownAfterClass() throws Exception {
    calc = null;
}
```


UNIVERSITY OF CALGARY

# Section 4

## Dependencies using Doubles

## Stubs – Mocks



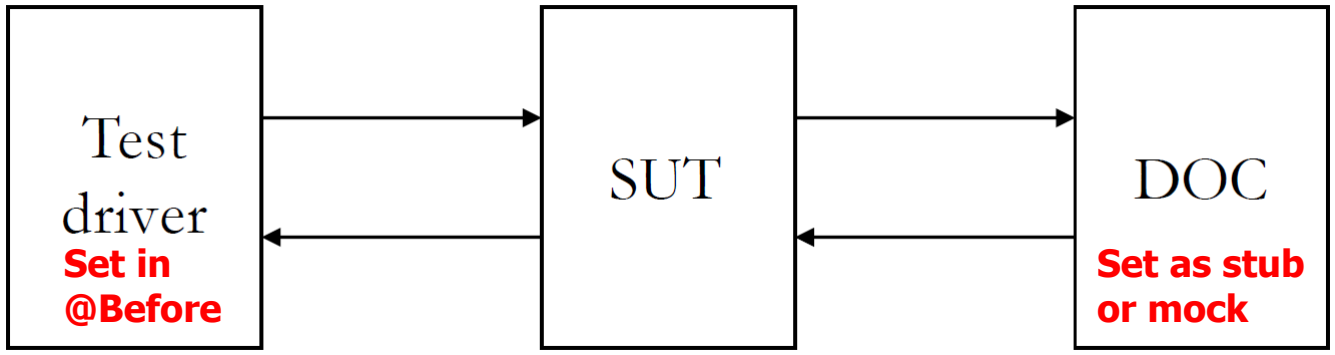
# Isolating Unit Tests

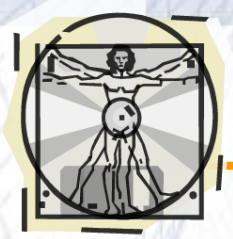
- SUTs that have no collaborators should be straightforward to test
- But typically SUTs have dependencies
  - Remember UML's collaboration diagram

- Unit tests should NOT have dependencies (i.e. independent runs of unit tests)
- Testing collaborative units together ← **integration test**, **feature test** will come later

# The Test Scenario

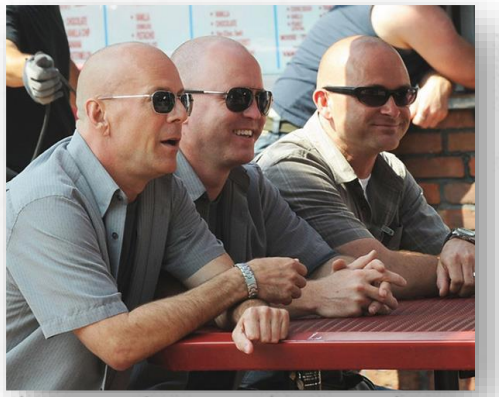
- SUT is the portion of a system that is being tested (may be one method or class or a collection of classes or components)
- Test Driver: Partial implementation of a component that depends on an SUT
- If the SUT collaborates with any other classes, those classes are referred to as "depended on components" (DOC's)

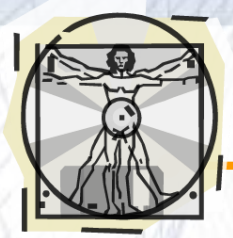




# Doubles

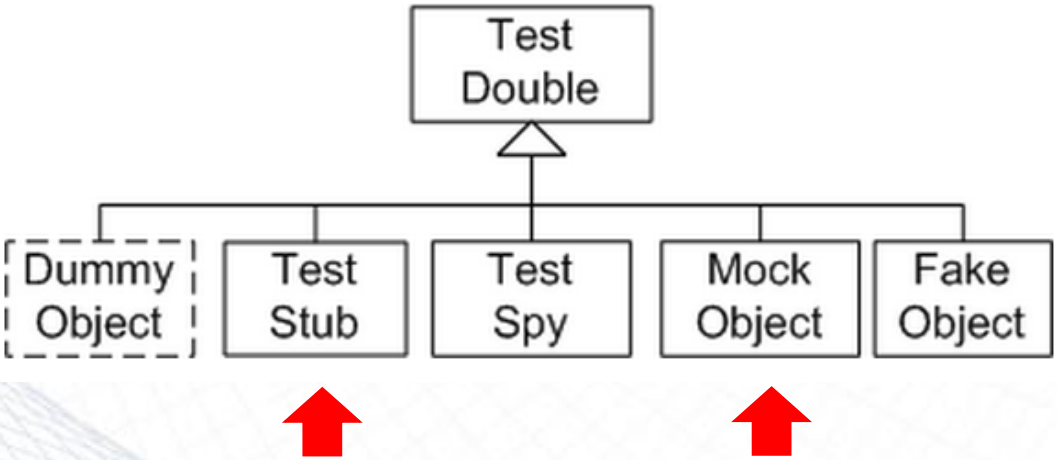
- A test double is a **replacement** for a **DOC** (Dependent On Component)
- Example
  - A system might **send an email** if a particular request fails (e.g. insufficient funds, insufficient stock on hand, security violation, etc.)
  - Since we **don't really want to send the email**, we need to provide replacement code that appears to send the email
  - We will typically want to **verify** that the email was "**sent**"

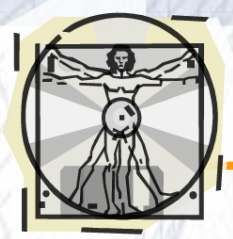




# Test Doubles

- Several of them exist
- In this course we focus on **stubs** and **mock objects**

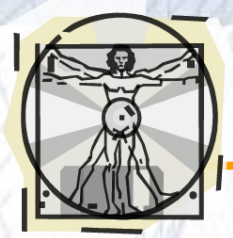




# Why Stubs – Mocks?

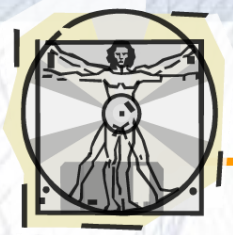
- Avoiding external dependencies:
  - Other classes and methods, file reading, database connection, sending email, etc.
- External dependencies must be removed for true Unit testing, i.e., isolation of functionalities
- Typical targets for stubs/mocks:
  - Database connections
  - Web services
  - Classes that are slow (may pass timeout)

**Mocks and stubs are fake Java classes that replace these external dependencies**

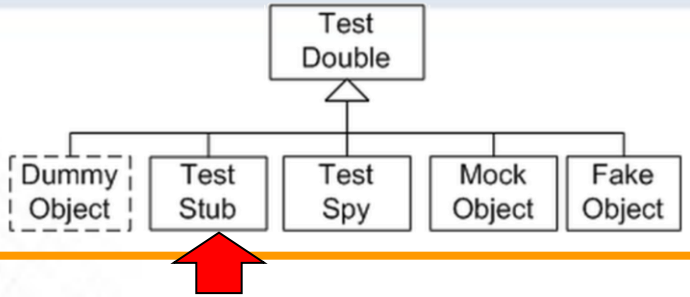


# Stub – Mock

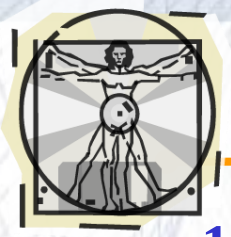
- A stub is a fake class that comes with preprogrammed return values. It's injected into the class under test to give control over what's being tested as input
  - **Example:** A database connection that allows you to mimic any connection scenario without having a real database
- A mock is a fake class that replaces the depended class and can be examined after the test is finished for its interactions with the class under test
  - **Example:** you can ask it whether a method was called or how many times it was called
  - Typical mocks are classes with side effects that need to be examined, e.g., a class that sends emails or sends data to another external service



# Stubs

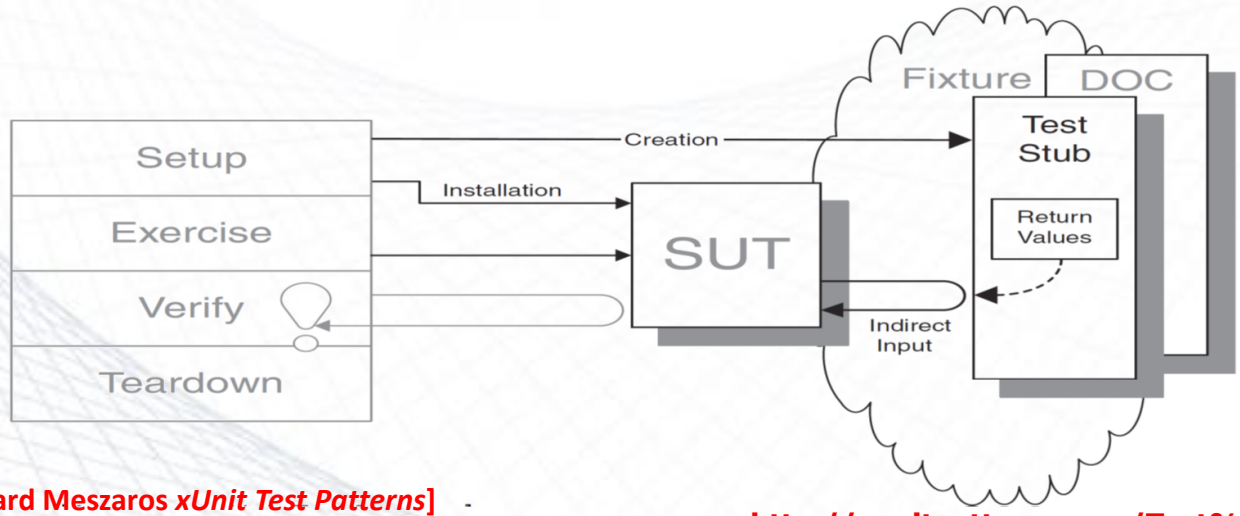



- A simple test double
- **Supplies responds to requests** from the SUT
  - Stub will contain **pre-defined responses** to specific requests
    - e.g. getStudent("100") causes a Student object to be returned
      - The Student object may be Student 100 or the same object may returned regardless of the student number
    - e.g. a program that reads from a sensor feed can be tested by getting the same info from a file
  - A stub is often **hand-coded** in the implementation language, i.e. no need to use a mock framework but it is ok (and usually recommended) to use one



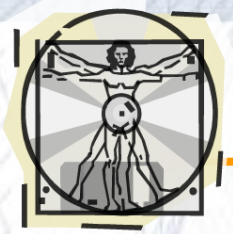
# Test Lifecycle with Stubs

1. **Setup** - Prepare SUT that is being tested and its stubs collaborators ← **usually in @Before**
2. **Exercise** - Test the functionality ← **in @Test**
3. **Verify state** - Use asserts to check object's state
4. **Teardown** - Clean up resources ← **usually in @After**

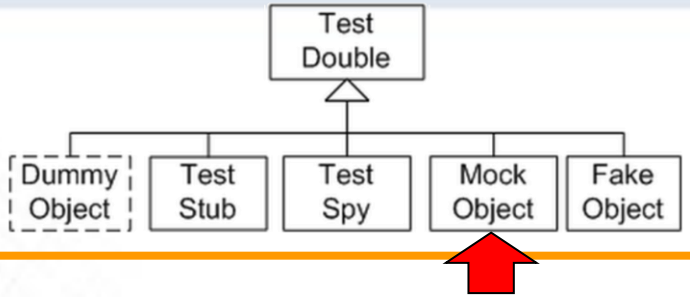



[Gerard Meszaros *xUnit Test Patterns*]

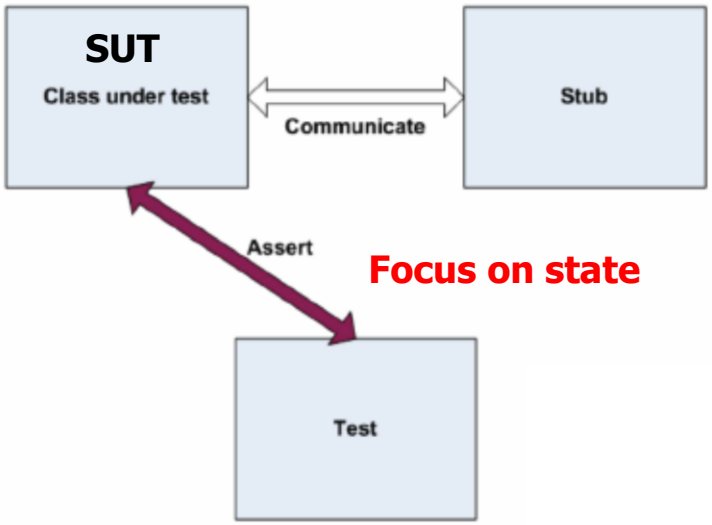
http://xunitpatterns.com/Test%20Stub.html

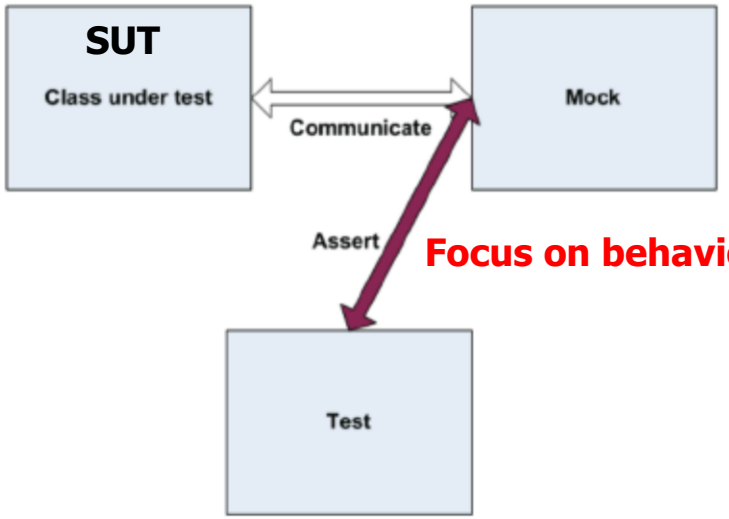


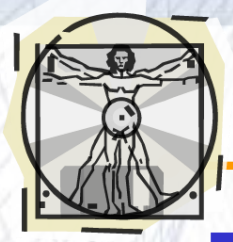
# Mock Objects




- A **fake object** that decides whether a unit test has passed or failed by **watching interactions between objects**
- Observe the difference below:

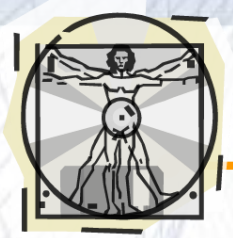




# Mock Objects

- Why do we need **mock** object?
  - When a unit of code depends upon an external object
- Mock object
  - Dummy implementation for an interface or a class in Mock framework
- Mock framework
  - jMock, Mockito, PowerMocks, EasyMock (Java)
  - SimpleTest / PHPUnit (PHP)
  - FlexMock / Mocha (Ruby)
  - etc.

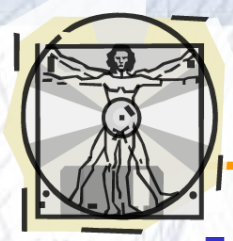


# Mock vs. Stub

- Can I use stub and mock interchangeably?

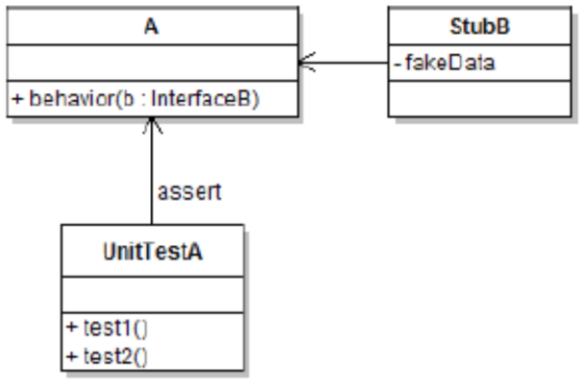
- **State testing:** *"What is the result?"*
  - Both mocks and stubs
- **Behavioral testing:** *"How the result has been achieved?"*
  - Only mocks

**Learn to write stubs/mocks. They can be used everywhere!**

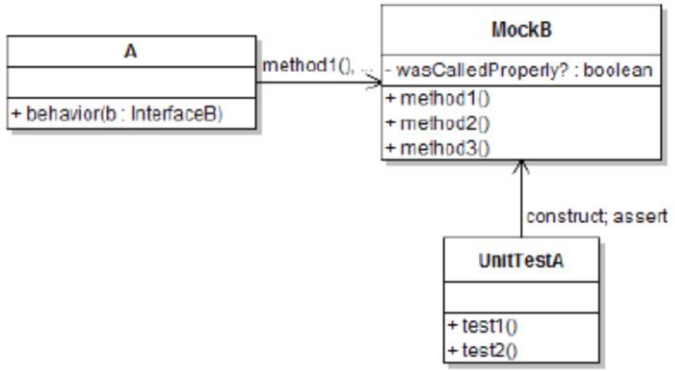


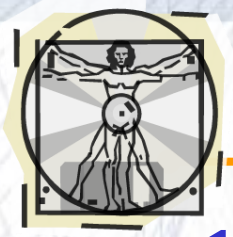
# Mock vs. Stub

- A **stub** gives out data that goes to the object/class under test
- The unit test directly asserts against class under test, to make sure it gives the right result when fed this data




- A **mock** is created and waits to be called by the class under test (A)
  - Maybe it has several methods expecting a call from A
- It makes sure that it was contacted in exactly the right way
  - If A interacts with B the way it should, the test passes

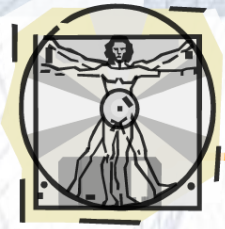




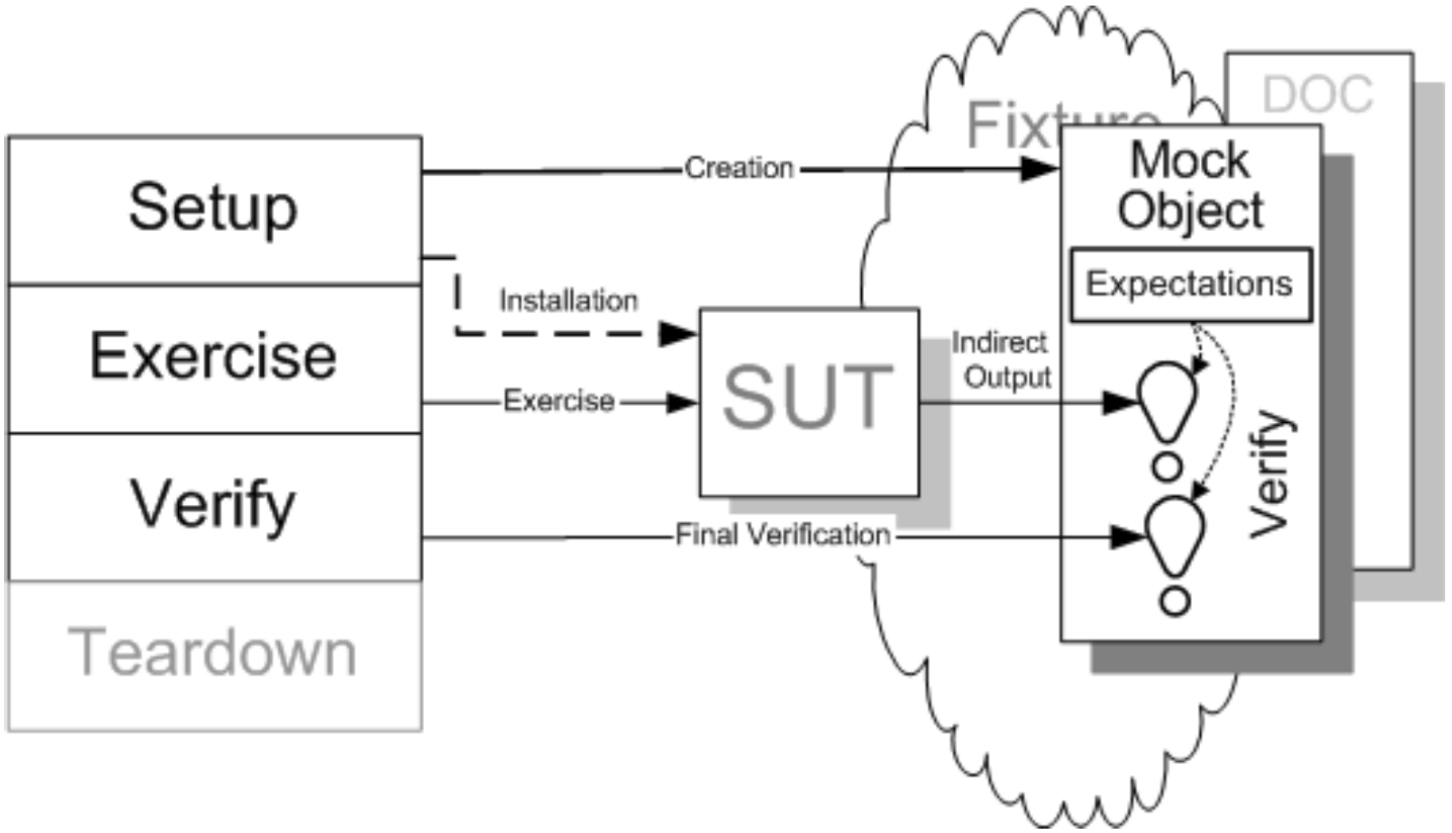
# Test Lifecycle with Mocks

1. **Setup data -** Prepare object that is being tested
2. **Setup expectation -** Prepare expectations in mock that is being used by primary object
3. **Exercise -** Test the functionality
4. **Verify expectations -** Verify that correct methods has been invoked in mock
5. **Verify state -** Use asserts to check object's state
6. **Teardown -** Clean up resources

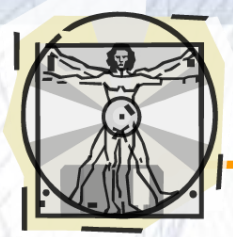
http://martinfowler.com/articles/mocksArentStubs.html



# Mock - Summary




http://xunitpatterns.com/Mock%20Object.html

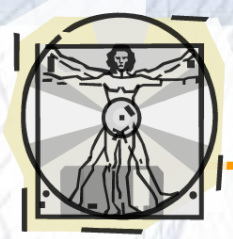


# Mocking & Integration Testing

- The definitions are not very clear
- In this course when testing interactions between two objects:
  - Using a stub/mock → unit testing
  - Using actual implementations → integration testing

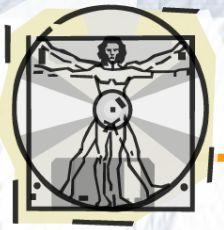
Remember we shouldn't mock every interaction:
e.g., a library call

We will talk about integration testing later



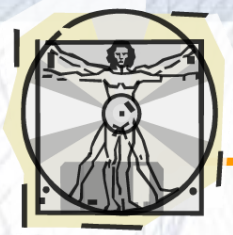
# Mock Object Frameworks

- Stubs and mocks are usually created by various Mock object frameworks
- Mock frameworks provide the following (via API):
  - Auto-generation of mock objects that implement a given interface
  - Methods/primitives for declaring and asserting your expectations ← similar to unit testing assertions
  - Logging of what calls are performed on the mock objects



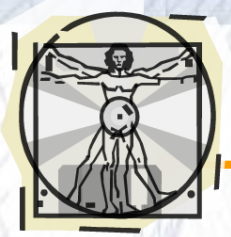
# Mock Object Frameworks

- jMock (https://github.com/jmock-developers)
- EasyMock (http://easymock.org/)
- Mockito (https://github.com/mockito/Mockito)
- PowerMock (https://github.com/powermock/powermoc/)
- etc.



# Let's Create – Stubs

- Let's use Mockito framework to create a stub, by using **`mock(class)`** ← This is to replace the DOC
- Then use **`when(mock).thenReturn(value)`** to specify the stub's return value for a method
- If you specify more than one value, they will be returned in sequence until the last one is used, after which point the last specified value gets returned
  - So to have a method return the same value always, just specify it once



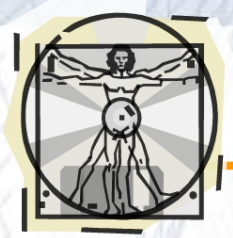
# Let's Create - Stubs

- **Example 1:** This example creates a mock **`Iterator`** and makes it return "Hello" the first time method **`next()`** is called and calls after that return "World"

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;
import java.util.*;
import org.junit.*;
{
....
 @Test
 public void iterator_will_return_hello_world(){
  //arrange
  Iterator i=mock(Iterator.class);
  when(i.next()).thenReturn("Hello").thenReturn("World");
  //act
  String result=i.next()+" "+i.next();
  //normal assertion
  assertEquals("Hello World", result);
 }
```

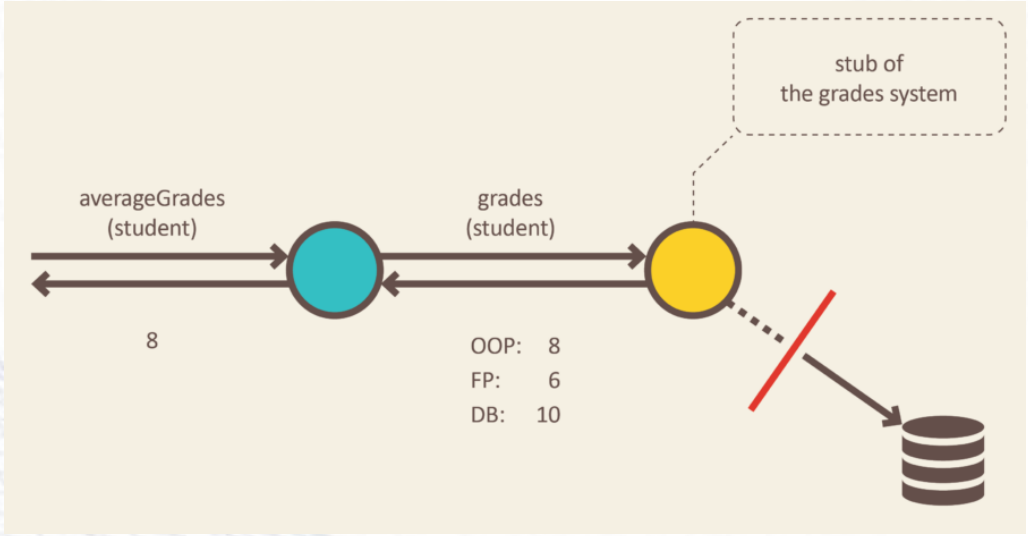
Can be put in @Before

https://gojko.net/2009/10/23/mockito-in-six-easy-examples/

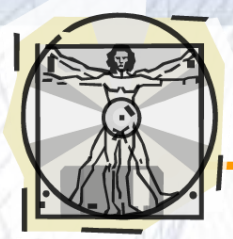


# Let's Create - Stubs

- **Example 2:** average grade
  - An object that needs to grab some data from the database to respond to a method call
  - Instead of the real object, we use a stub and defined what data should be returned



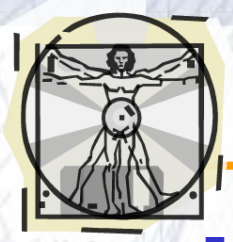

https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da



# Let's Create - Stubs

- SUT

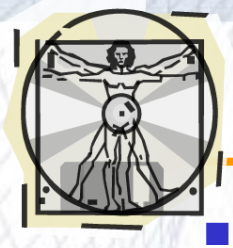
```
public class GradesService {
    private final Gradebook gradebook;

    public GradesService(Gradebook gradebook) {
        this.gradebook = gradebook;
    }

    Double averageGrades(Student student) {
        return average(gradebook.gradesFor(student));
    }
}
```



# Let's Create - Stubs

- Instead of calling database from Gradebook store to get real students grades, we preconfigure stub with grades that will be returned

```
public class GradesServiceTest {
    private Student student;
    private Gradebook gradebook;
    @Before
    public void setUp() throws Exception {
        gradebook = mock(Gradebook.class); //create stub
        student = new Student();
    }
    @Test
    public void calculates_grades_average_for_student() {
        //stubbing next: use stub
        when(gradebook.gradesFor(student)).thenReturn(grades(8, 6, 10));
        double averageGrades =
                new GradesService(gradebook).averageGrades(student);
        //normal assertion next:
        assertThat(averageGrades).isEqualTo(8.0);
    }
}
```



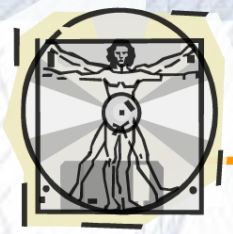
# Mocking using jMock

- jMock has a very strong and handy API
- Specifying objects and calls:

```
oneOf(mock), exactly(count).of(mock),
atLeast(count).of(mock), atMost(count).of(mock),
between(min, max).of(mock)
allowing(mock), never(mock)
```

- The below accepts a mock object and returns a descriptor that you can call methods on, as a way of saying that you demand that those methods be called by the class under test
  - **`atLeast(3).of(mockB).method1();`**
  - Meaning:

"I expect that **`method1`** will be called on **`mockB`** 3 times here"



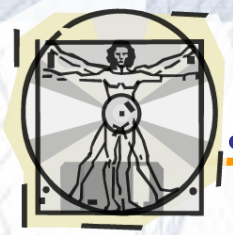
# JMock Expected Actions

- **`will(action)`**
  - actions: **`returnValue(v), throwException(e)`**
- values:

  equal(**value**), same(**value**), any(**type**), aNull(**type**), aNonNull(**type**), not(**value**), anyOf(**value1,...,valueN**)
- **Example:**

```
oneOf(mockB).method1();
will(returnValue(anyOf(1,4,-3)));
```

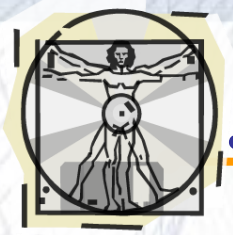
  - "I expect that **`method1`** will be called on **`mockB`** once here, and that it will return either 1, 4, or -3."



# jMock How to Use it?

- With the help of jMock we can test a class independently when it depends on another class by creating a mock of dependent class
- What we need?
  - Eclipse and JUnit and jMock jars included in the classpath (and regular imports in the program)
  - Perhaps some additional libraries (in case something is wrong):

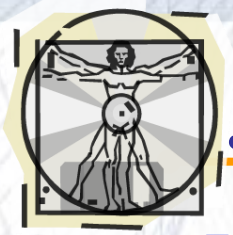
    hamcrest-core-1.3.jar, hamcrest-library-1.3.jar



# jMock How to Use it?

- Let's create an interface named **`ITestInterface`** and put the following code in it:

```
package test;
public interface ITestInterface
{
    public int test();
}
```

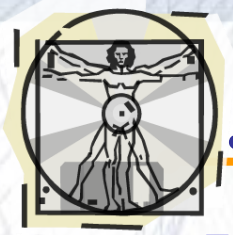


# jMock How to Use it?

- Create **`TestClass1`**. Put the following code in this class
- This is the class that SUT depends on, i.e., the DOC

```
package test;

public class TestClass1 implements ITestInterface
{
    public int test()
    {
      return 3;
    }
}
```

**Here TestClass1 is a depended on component (DOC)**



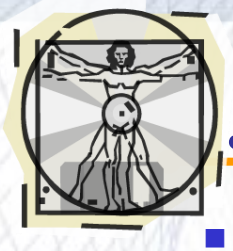
# jMock How to Use it?

- Create **`TestClass2`**. Put the following code in this class
- This is our SUT

```
package test;
public class TestClass2
{
    ITestInterface testInterface;
    public TestClass2()
     {
       this.testInterface=testInterface;
     }
    public int testMock()
     {
       int result=testInterface.test();  //Dependency here
       return result;
     }
}
```

SUT's testMock() method cannot run independent of the TestClass1

**Here TestClass1 is a depended on component (DOC)**



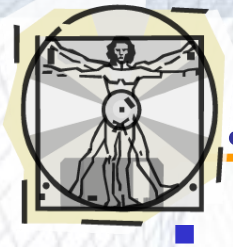
# jMock How to Use it?

- Create **`TestClass3`**. Put the following code in this class
- This is our JUnit testing code

```
package test;
import junit.framework.Assert;
import org.jmock.Mockery;
import org.junit.Test;
public class TestClass3      //Junit Test
{
  @Test
  public void testJmock()
  {
    org.jmock.Mockery TestInterfaceMock= new Mockery();
    final ITestInterface testInterface=TestInterfaceMock.mock(ITestInterface.class);
    TestInterfaceMock.checking(new org.jmock.Expectations()
    {{
       oneOf(testInterface).test();
       will(returnValue((3)));    // will return by mock object
     }});
    int j=testInterface.test();
    System.out.println(j);
    Assert.assertEquals(j, 3);  //Pass if returns 3
  }
}
```

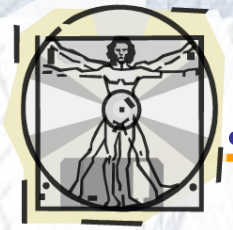
**Mock object**

**A mock object has the same method as the called object but not the implementation of it. It returns the expected outcome only.**



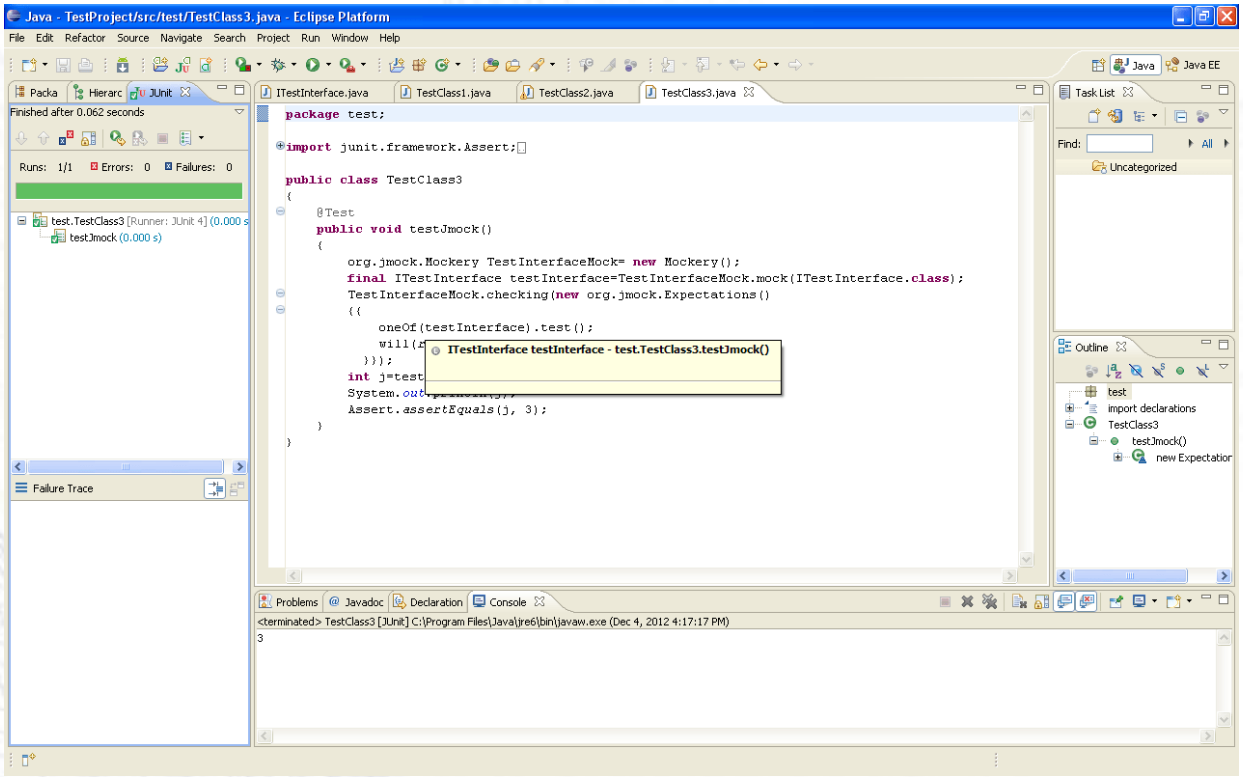
# jMock How to Use it?

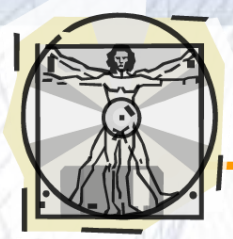
- **`TestClass1`** implements the interface **`ITestInterface`**
- Here **`TestClass2`** is dependent on **`TestClass1`**
- In unit testing we are not doing new of **`TestClass1`** in **`TestClass2`** We use mock object in unit testing it
- In **`TestClass3`** we are making the Mock object of **`ITestInterface`** We are setting the expectation for this object to return 3 when it is called during the unit testing of **`TestClass2`**
- For example in **`TestClass3`** when we make a call to **`testInterface.test()`** the call goes to **`TestClass2`**. In **`TestClass2`** when we call **`testInterface.test()`** the value is returned by the mock object. In this way we are testing **`TestClass2`** independent of **`TestClass1`**



# jMock How to Use it?

- Right click on **`TestClass3`** and select run as JUnit test and check test executed successfully





# Recommendations

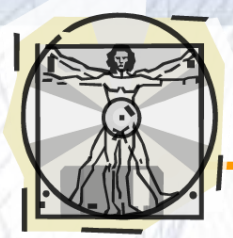
- Use jMock or Mockito for all stubbing and mocking jobs in Java
- jMock: more documented, lots of examples

jMock

- Mockito: more powerful, easy to use
  - Mockito offers two ways of mocking:
    - Using static methods
    - Using @Mock annotations

mockito

**Download Mockito:** https://site.mockito.org/
**Mockito Tutorials:** https://semaphoreci.com/community/tutorials/stubbing-and-mocking-with-mockito-2-and-junit



# Exercise

- Should I mock an external library that my code depends upon?

  **No**

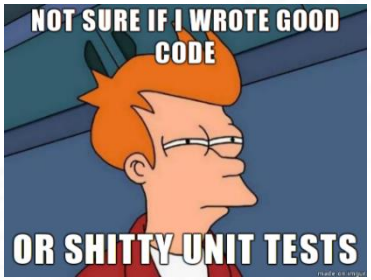
- Should I mock a two way call?

  **No, change the design!**

# JUnit - Summary

- Tests need *failure atomicity* (ability to know exactly what failed)
  - Each test should have a clear, long, descriptive name
  - Assertions should always have clear messages to know what failed
  - Write many small tests, not one big test
    - Each test should have roughly just 1 assertion at its end
- Test for expected errors / exceptions
- Choose a descriptive assert method, i.e.,

  not always `assertTrue`




Next week

- Choose representative test cases from equivalent input classes
- Avoid complex logic in test methods
- Use helpers, **`@Before`**, **`@After`** to reduce redundancy between tests
- Learn to write stubs and mocks