SCHULICH School of Engineering

UNIVERSITY OF CALGARY

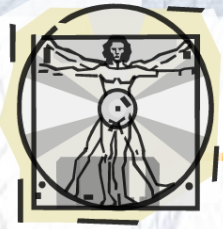
SENG 438
Software Testing, Reliability & Quality

Chapter 3: Criteria-Based Test Design
Unit Testing

Department of Electrical & Computer 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

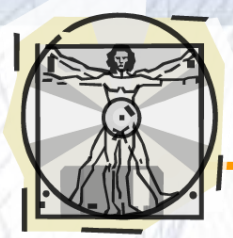
- Functional Testing: testing functional requirements
  - **Functional testing is checking correct functionality of a system**

Our focus

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

  e.g. performance, reliability, security

Later



What are Non-functions?

- Performance testing
- Localization/Internationalization testing
- Recovery testing
- Security testing
- Portability testing
- Compatibility testing
- Usability testing
- Scalability testing
- Reliability testing
- …ilities!

quality time

Editors: Nancy Eickelmann ■ Motorola ■ nancy.eickelmann@motorola.com
Jane Huffman Hayes ■ University of Kentucky ■ hayes@cs.uky.edu

Software's Secret Sauce: The "-ilities"

Jeffrey Voas

If beauty is in the eye of the beholder, then quality must be as well. We live in a world where beauty to one is a complete turnoff to another. Software quality is no different. We have the developer's perspective, the end users' perspective, the testers' perspective, and so forth. So, for instance, if the software complies with the requirements, is that quality software? Many would say yes, but then, what if the software isn't fit for any reasonable purpose? Or is quality software simply software that was developed according to particular standards and regulations?

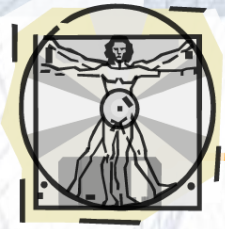
As you can see, meeting the requirements might be different from being fit for a purpose, which can also be different from complying with rules and regulations on how to develop and deploy the software. Yet we can think of all three perspectives as ways to determine how to judge and assess software quality.

when added. To use an analogy, an -ility in an application or system is like a condiment on an entrée: not valuable as a stand-alone item but capable of significantly enhancing the flavor when added properly. Examples of these flavor-enhancing software -ilities include maintainability, reliability, usability, efficiency, adaptability, availability, security, portability, scalability, safety, fault tolerance, testability, usability, reusability, and sustainability.

So, if we look at the issue of the software meeting its requirements, and if those requirements are solely functional and prescribe no -ilities, then clearly the software can meet the requirements but could be unable to fulfill any reasonable purpose. Furthermore, the -ilities will unlikely be associated in any way with how the software is developed, because development standards rarely drill down low enough to provide the guidance needed to attain particular degrees of any -ility.

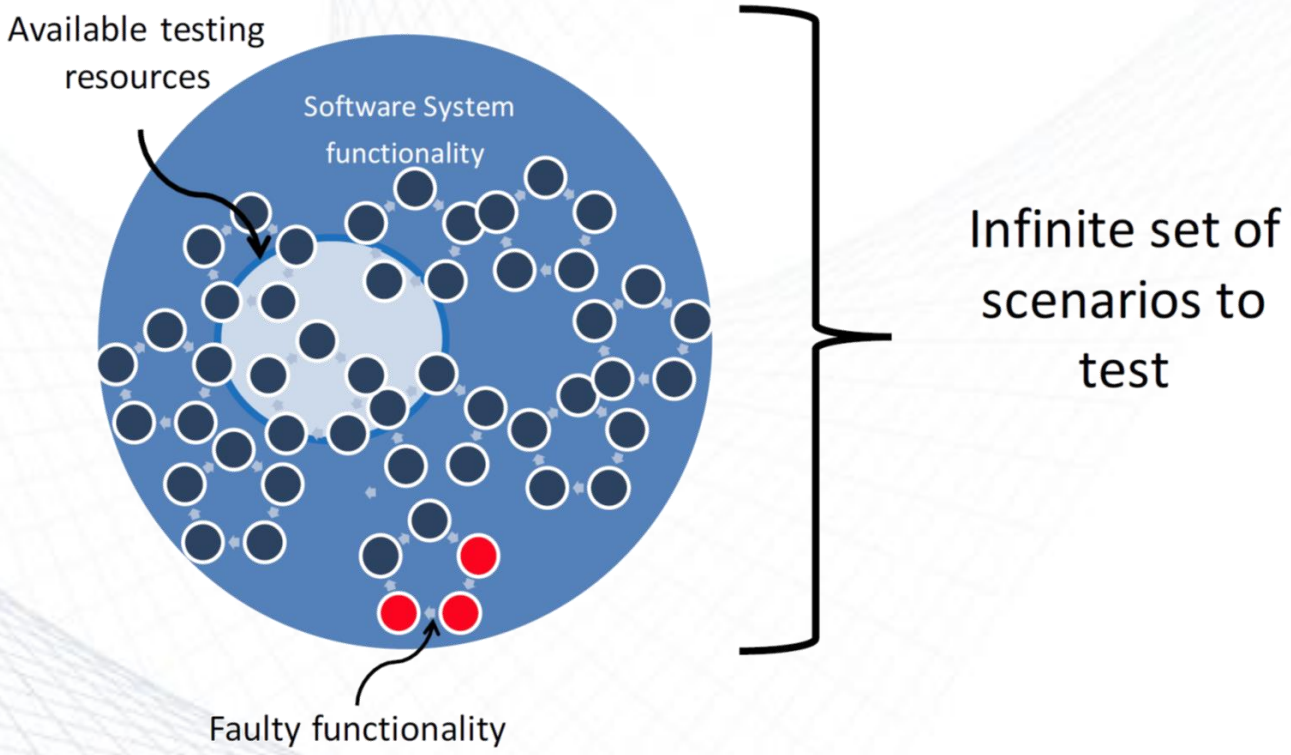
Measure for measurement

Note that degree is important. For example, a little salt on a steak usually tastes great.



The Testing Dilemma

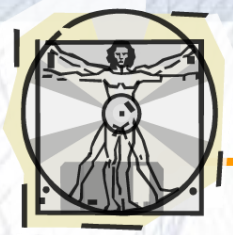
- Using limited resources (time, personnel, budget) efficiently



Criteria Based Testing

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

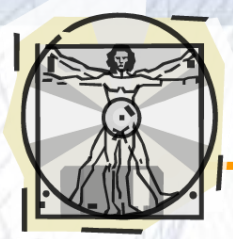
Partitioning
Coverage (i.e. statement, decision, condition, path)
Mutation
Acceptance
etc.



Testing Approaches

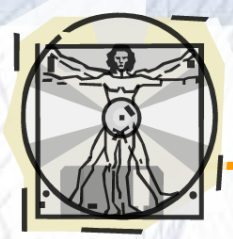
- Exhaustive testing
  - Pros: guarantee
  - Cons: infeasible in most real world applications
- Random testing (ad-hoc, exploratory)
  - Pros: cheap
  - Cons: inefficient in many situations
- Partitioning
  - Pros: effective in theory
  - Cons: practical solutions?

Done Already



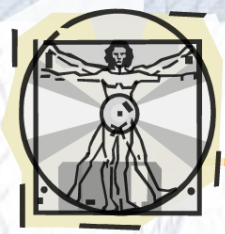
Exhaustive Testing

- Exhaustive testing, i.e., testing a software system using all the possible inputs, is impossible (most of the time)
- Example:
  - A compiler (e.g., javac)
  - Exhaustive testing = running the (Java) compiler with any possible (Java) program (i.e., source code)



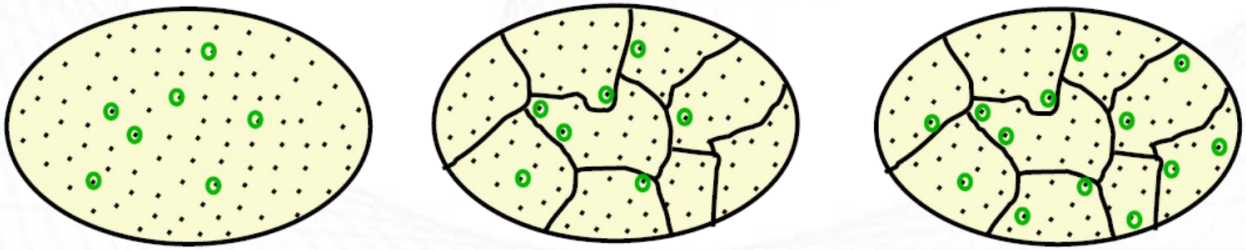
Partitioning Techniques

- Non-systematic - human-based
  - Exploratory ⟵ **Covered already**
- Systematic
  - Criteria-based ⟵ **Next two weeks**

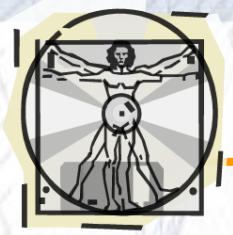


Partitioning: Systematic

- Partition the set of possible behaviors of the system
- Choose representative samples from each partition
- Make sure we covered all partitions

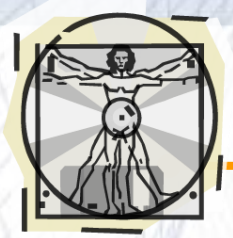


- How do you identify suitable partitions?
- That is what testing is mostly about!!!"
  - Methods:
    - black box, white box, ...
    - path based, state based, risk based, scenario based, …



Criteria Based Technique

- **Black-box**
  - Access to specification is granted
  - Goal: Increase input domain coverage
  - Partitions are based on slices of requirements and their expected behavior
- **White-box**
  - Access to source code is granted
  - Goal: Increase code coverage
  - Partitions are based on grouping input domain elements into classes that cover different statements/branches/paths in the source code



Example: Black Box

- **Specification of Compute Factorial Number:**
  - if the input value n < 0, then an error message must be printed
  - if 0 <= n < 20, then the exact value of n! must be printed
  - if 20 <= n < 200, then an approximate value of n! must be printed in floating point format, e.g., using some approximate method of numerical calculus (admissible error is 0.1% of the exact value)
  - Finally, if n>=200, the input can be rejected by printing an error message

Classes: {n<0}, {0<= n <20}, {20 <= n < 200}, {n >= 200}

Example: White Box

```
1. if x > y then
2. Max := x;
3. else
4. Max :=x ;   // BUG!
```

Class 1: x>y

Class 2: x<=y

Test suite: {x=3, y=2} cannot detect the fault

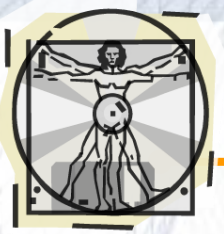
Statement coverage = 2/4 (lines 1 and 2)

Test suite: {x=3, y=2} {x=4, y=3} {x=5, y=1} is larger but cannot detect the fault

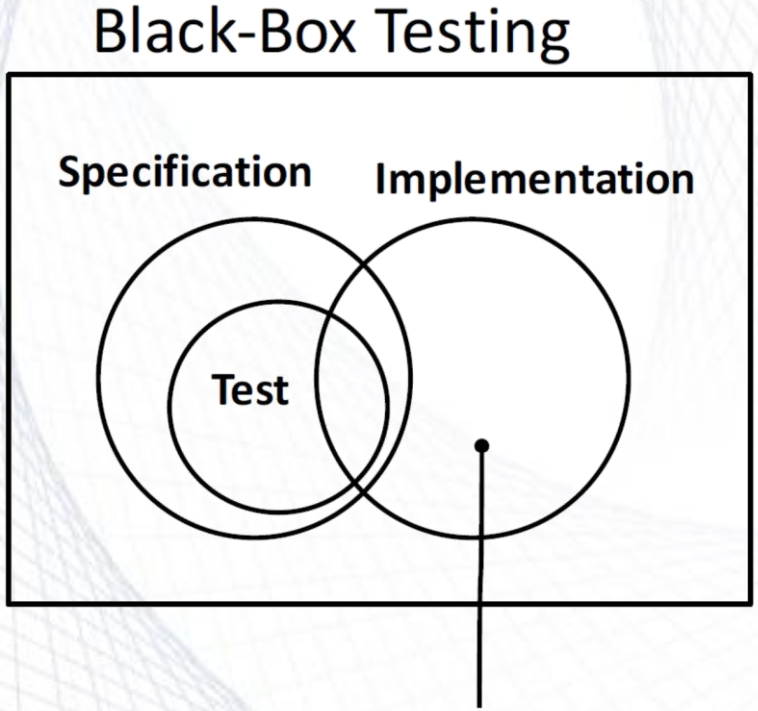
Statement coverage = 2/4 (lines 1 and 2)

Test suite: {x=3, y=2} {x=2, y=3} can detect the fault

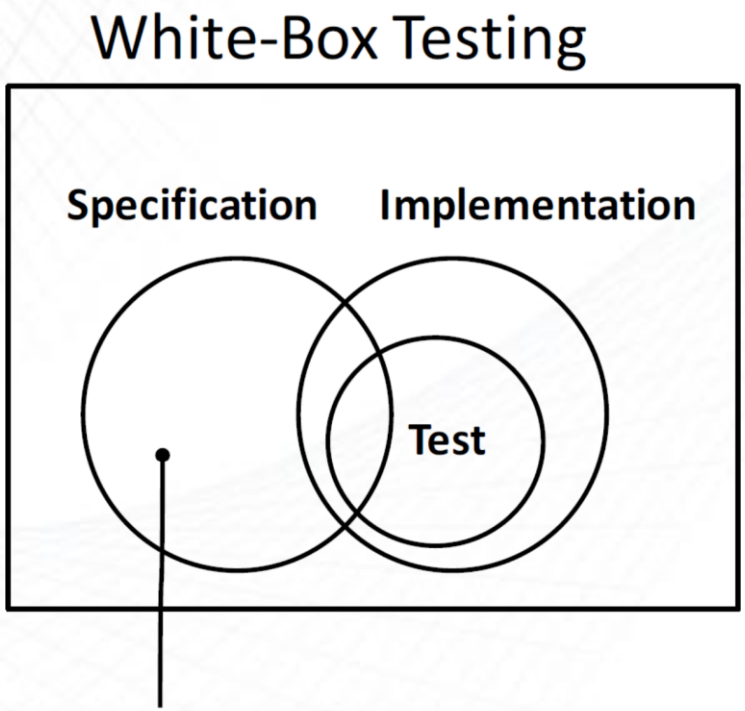
Statement coverage = 4/4 (all lines)



Black vs. White Box Testing



Unexpected functionality:
Cannot be revealed by black-box techniques



Missing functionality:
Cannot be revealed by white-box techniques

Black vs. White Box Testing

Black-box

- It scales up (different techniques at different granularity levels)
- It depends on the specification notation and degree of detail
- Do not detect unspecified task.

White-box

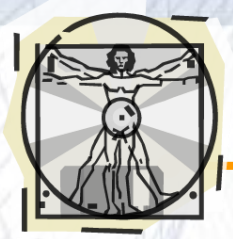
- It allows you to be confident about test coverage
- It does not scale up (mostly applicable at unit and integration testing levels)
- Unlike black-box techniques, it cannot reveal missing functionalities

SCHULICH School of Engineering

UNIVERSITY OF CALGARY

Section 2

Unit Testing

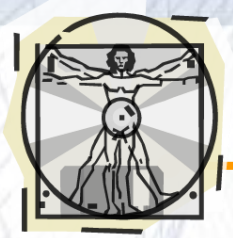


Unit Testing Scenario

- 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

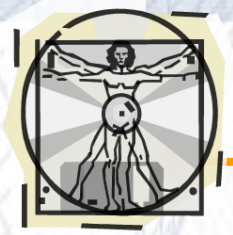
How realistic is this scenario?
What difficulties it may cause?

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 based on **white-box** testing

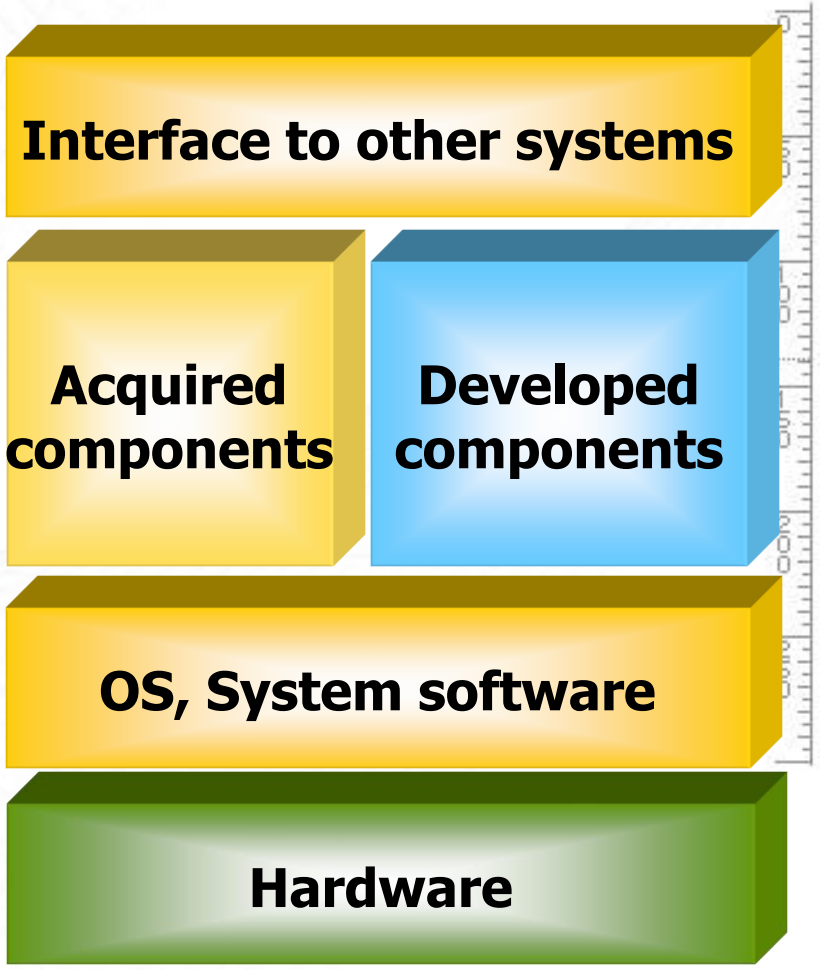


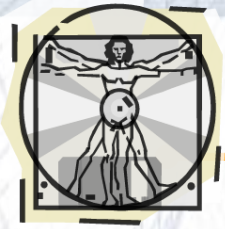
Unit Testing: Characteristics

- A unit test must only test **one specific unit of functionality** (e.g. **class** or **method**)
- It is **fast**
  - 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?

- 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 set: <0, 1, 2, 3, 4, ... , 10>
  - Boundary analysis <-1, 0, 1, 9, 10, 11> One sample from each class





Unit Testing: When to Test?

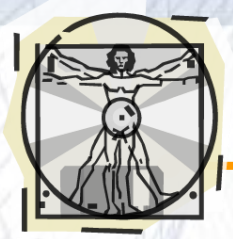
- Waterfall
  - Typically written **as (or after)** the system has been developed
- TDD/Agile
  - Each unit test is written **before** or **during** the corresponding code being written

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

Section 3

Automated Test Execution

JUnit



Test Automation Framework

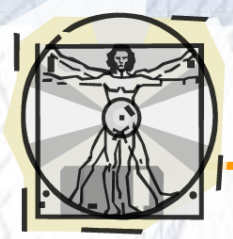
- A set of assumptions, concepts, and mainly tools, that support test automation
- Some unit testing frameworks

.NET
- NUnit
- Pex

PHP
- PHPUnit

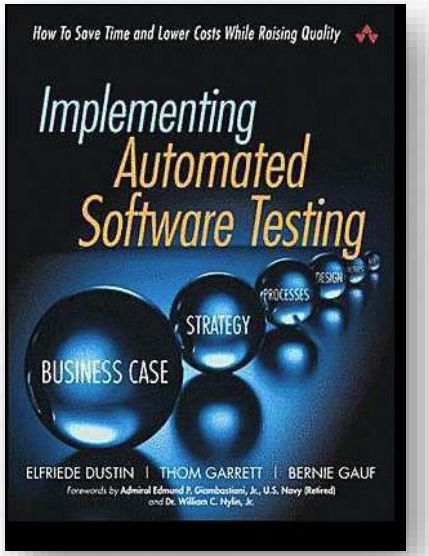
JavaScript
- JSUnit
- Jasmine (not Xunit)
- JasUnit

Java
- JUnit
- TestNG



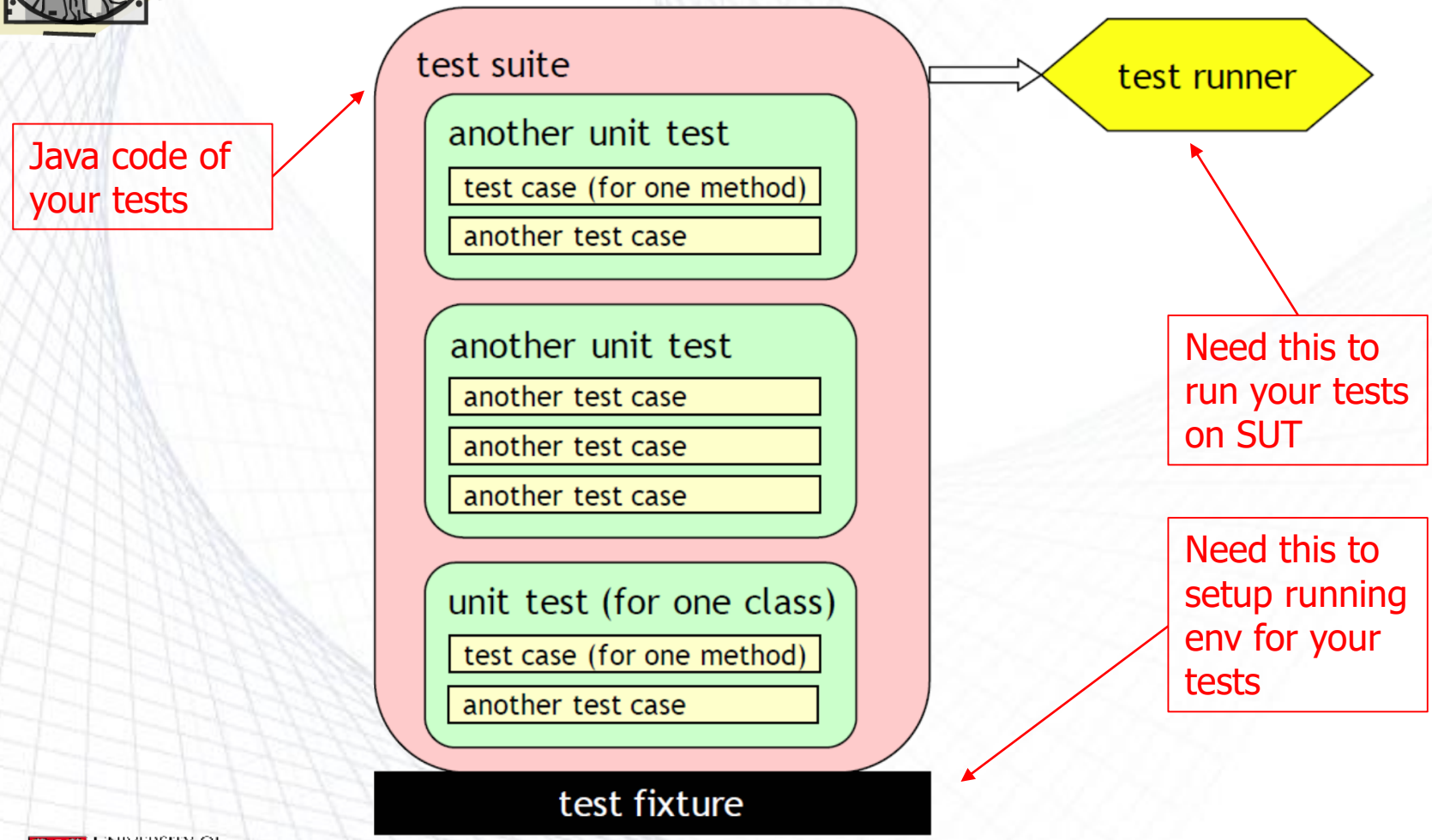
JUnit

- 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



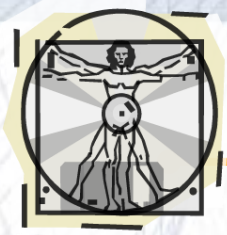
- Can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse

JUnit in a Nutshell



JUnit: Terminology

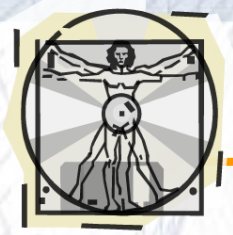
- **Test runner**: A test runner is an executable program that runs tests implemented using JUnit framework and reports test results (in command line mode) or update the display (in IDE)
- **Test fixtures**: A test fixture (context) is the set of preconditions or state needed to run a test
- **Test case**: 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 act as temporary replacement for a calling module and give the same output as that of the actual product
- **Test execution**: The execution of an individual unit test proceeds as follows: annotoations `@Before` `@After` `@Test`

  --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

Some older systems test suites are still in JUnit 3 so we need to be familiar with them all

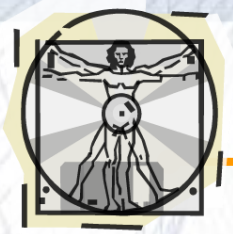
We will use either of JUnit 4 or 5 in the exercises and Labs

JUnit 4 vs. JUnit 5

- JUnit 5 (Sept 2017) requires Java 8 or higher
- JUnit 4 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

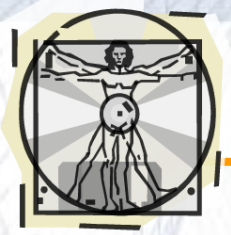
https://howtoprogram.xyz/2016/08/10/junit-5-vs-junit-4/



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



Simple JUnit Example

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

```
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));
   }
}
```

Annotation

Assertion

Test values

Printed if assert fails

Expected output

JUnit 3 vs. JUnit 4 / 5

- **Defining test case:**

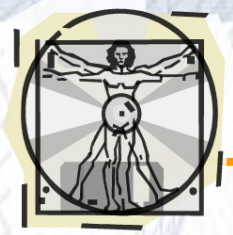
Extending TestCase is required in JUnit 3

```java
import junit.framework.Assert;
import junit.framework.TestCase;

public class TournamentTest extends TestCase{
  Tournament tournament;

  public void setUp() throws Exception {
    System.out.println("Setting up ...");
    tournament = new Tournament(100, 60);
  }

  public void tearDown() throws Exception {
    System.out.println("Tearing down ...");
    tournament = null;
  }

  public void testGetBestTeam() {
    Assert.assertNotNull(tournament);

    Team team = tournament.getBestTeam();
    Assert.assertNotNull(team);
    Assert.assertEquals(team.getName(), "Test1");
  }
}
```

JUnit 3

```java
import junit.framework.Assert;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class TournamentTest {
  Tournament tournament;

  @Before
  public void init() throws Exception {
    System.out.println("Setting up ...");
    tournament = new Tournament(100, 60);
  }

  @After
  public void destroy() throws Exception {
    System.out.println("Tearing down ...");
    tournament = null;
  }

  @Test
  public void testGetBestTeam() {
    Assert.assertNotNull(tournament);

    Team team = tournament.getBestTeam();
    Assert.assertNotNull(team);
    Assert.assertEquals(team.getName(), "Test1");
  }
}
```

JUnit 4



JUnit 3 vs. JUnit 4 / 5

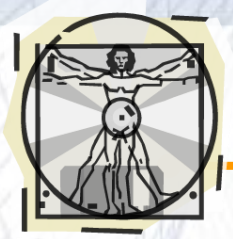
- Annotations and new features

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 4 vs. JUnit 5

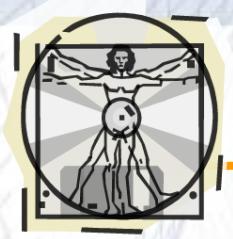
- Annotations

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 3 vs. JUnit 4 / 5

- Ignoring a test
  - In the 3.x version of JUnit, the only way to do that is to comment the whole test method or rename the method
  - This is significantly improved in JUnit 4.x, by the introduction of the @Ignore or in JUnit 5 by @Disable annotation



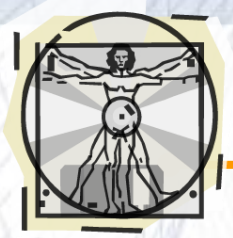
JUnit 3 vs. JUnit 4 / 5

- Timeout testing
  - Another parameter you can add to the @Test annotation is the timeout parameter
  - This feature was also missing in the “old” JUnit
  - With this parameter, you can specify a value in milliseconds that you expect to be the upper limit of the time you spend executing your test

JUnit 3 vs. JUnit 4 / 5

- Test suites
  - In JUnit 3, the old way of constructing sets of your tests involved writing a suite() method and manually inserting all the tests that you want to be present in the suite
  - In JUnit 4, the suite construction is done not by one annotation but by two annotations: the @RunWith and @Suite Classes annotations

```
@RunWith(Suite.class)
@SuiteClasses(ATest.class, BTest.class, CTest.class)
public class ABCSuite {
 }
```



JUnit 3 vs. JUnit 4 / 5

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

JUnit and Eclipse

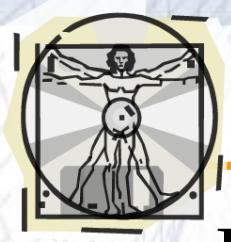
- JUnit works together with the Eclipse
- If Junit is not included in your IDE
  - To add JUnit to an Eclipse project, click:
  - Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish

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

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

http://www.vogella.com/tutorials/JUnit/article.html

Junit5 User Guide (in D2L)

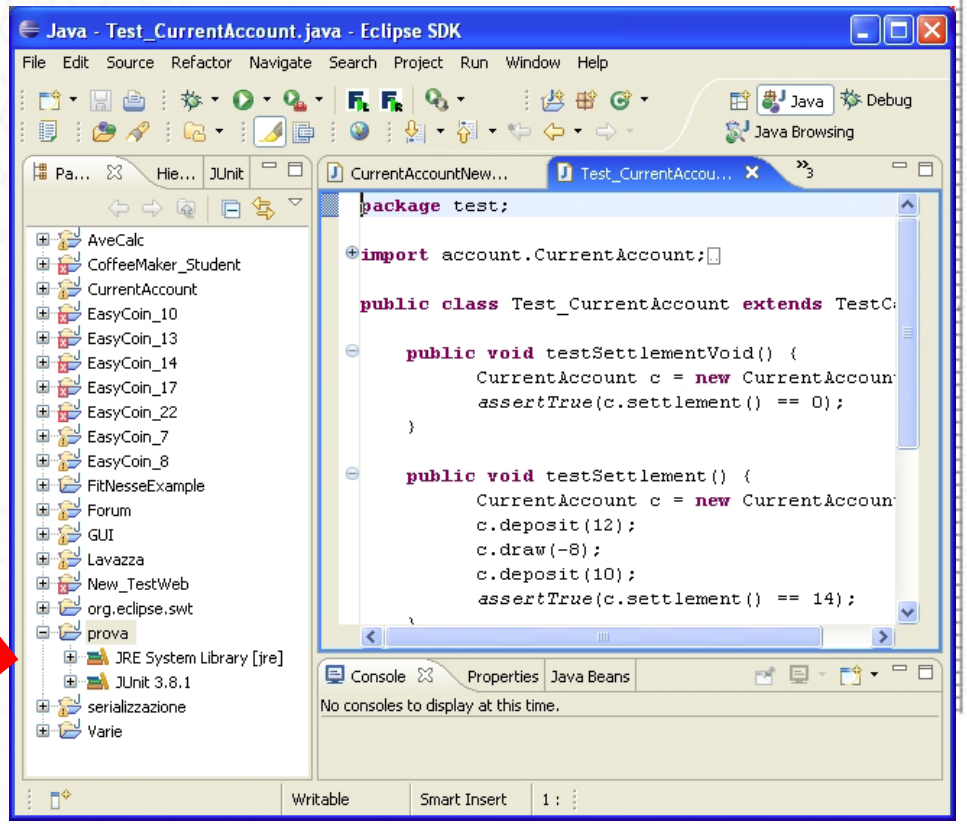


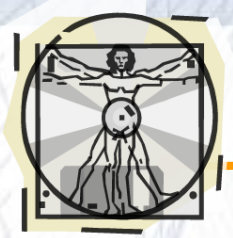
JUnit in Eclipse - Setup

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

Select: libraries

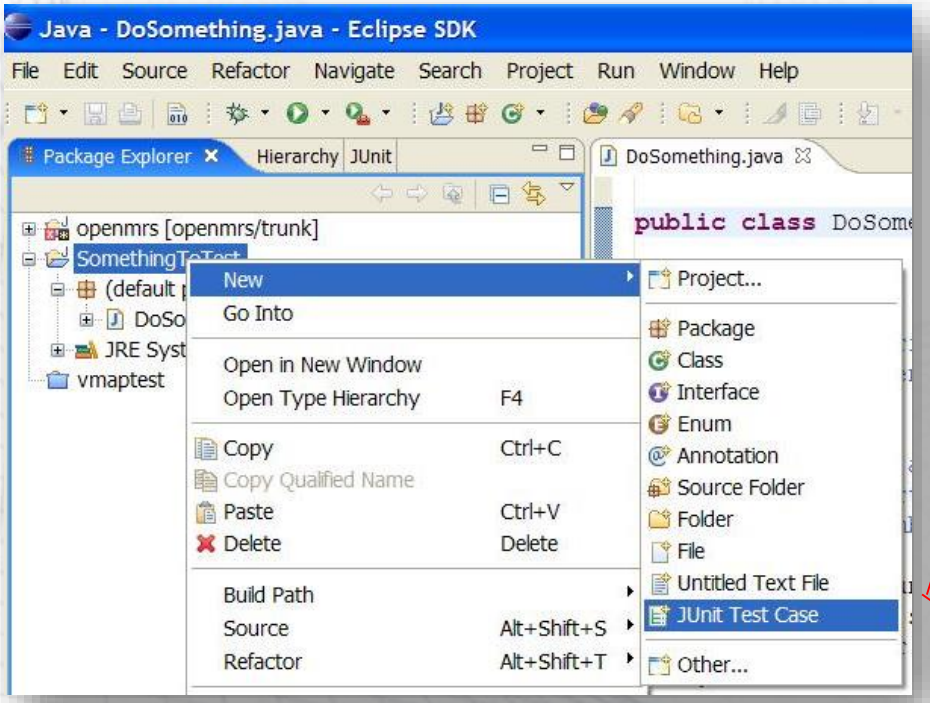
  - Add Library
  - Select Junit
    - Select the type 4.x or 5.x

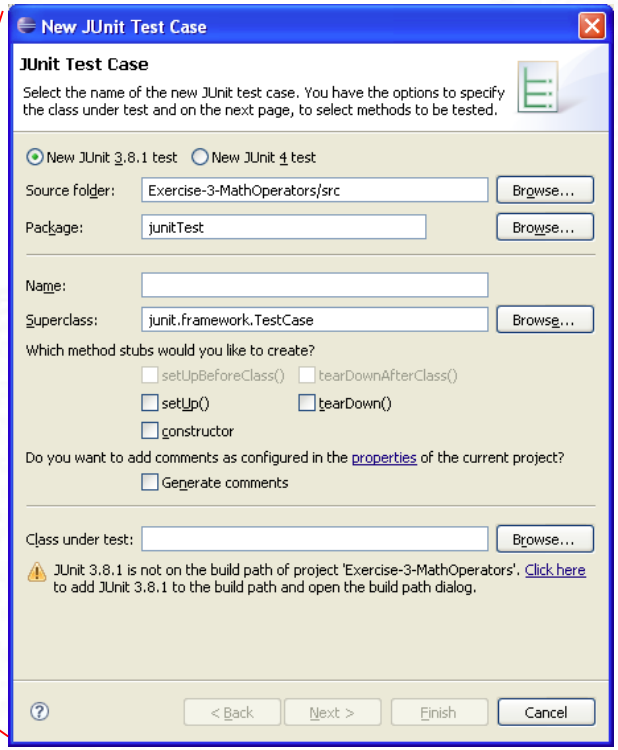


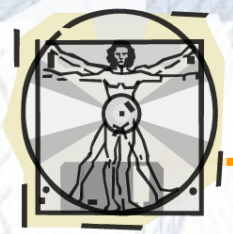


Creating a Test Case

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

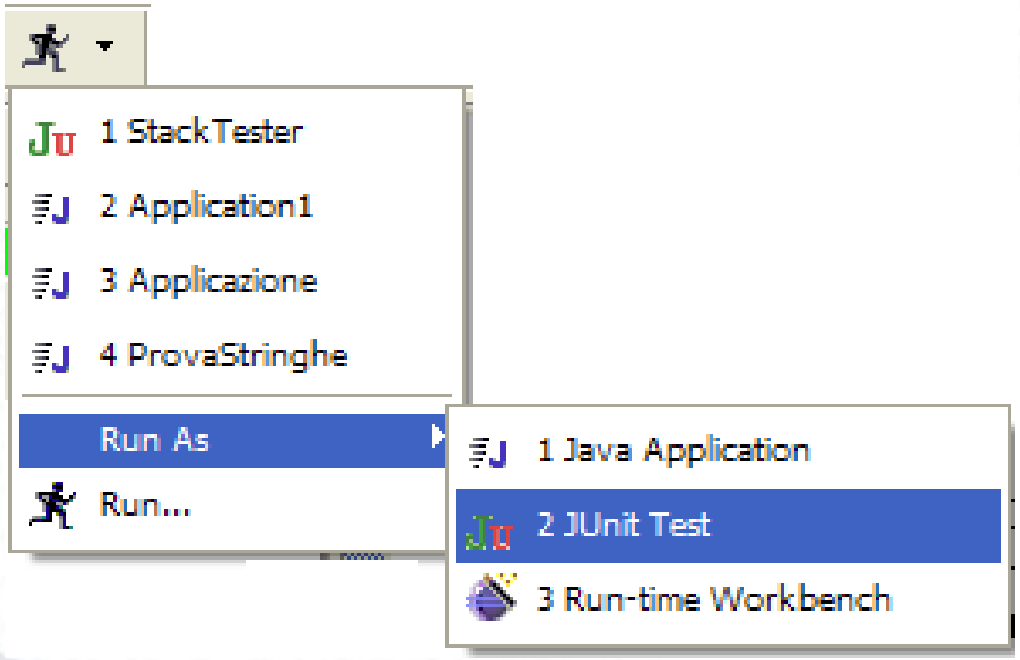




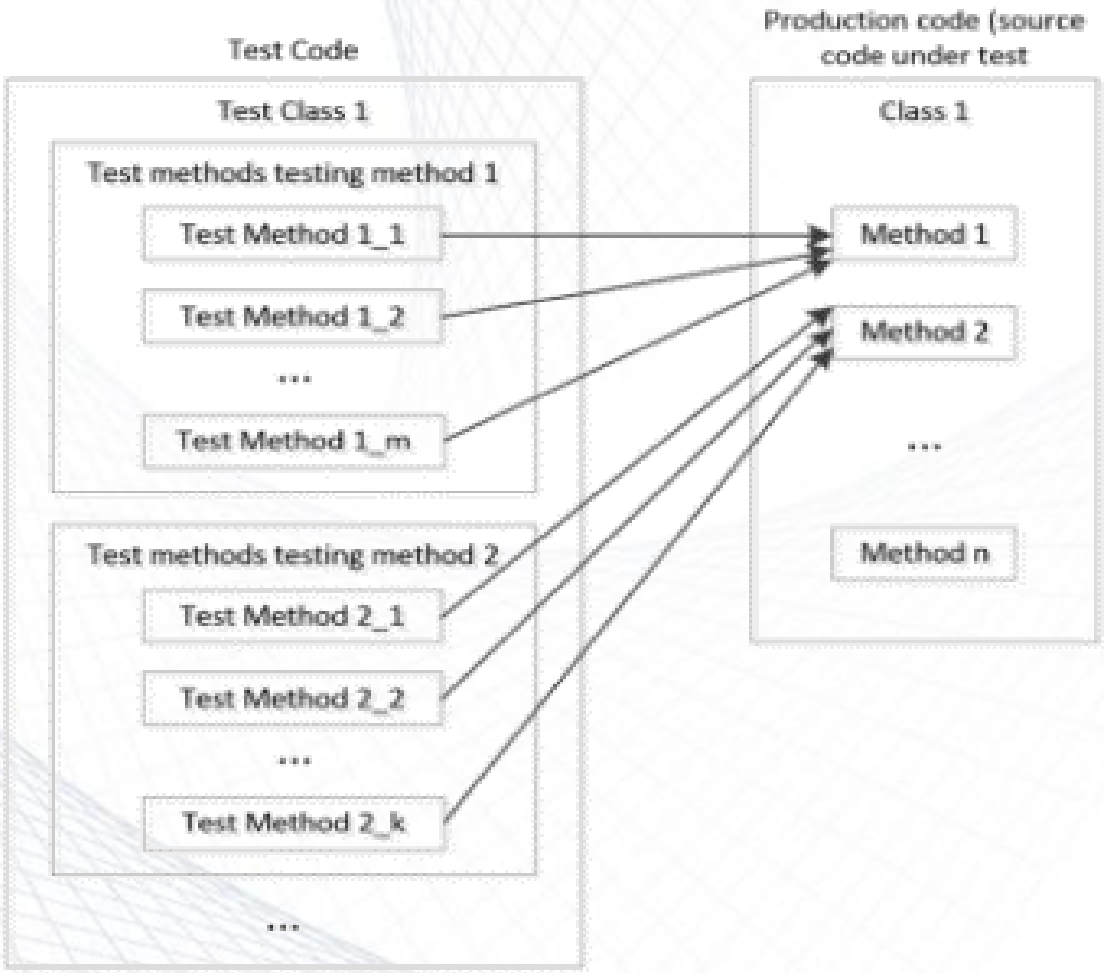


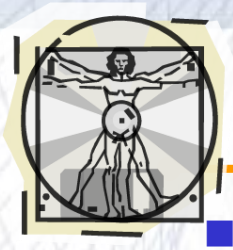
Run a JUnit Test

- Run
  - Run As
    - Junit Test



Test Case Structure





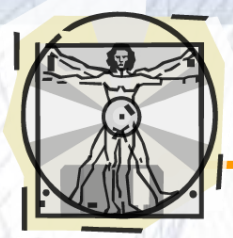
Test Suites

- **Test suite**: One class that runs many JUnit tests
  - An easy way to run all of your app's tests at once

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  TestFeatureLogin.class,
  TestFeatureLogout.class,
  TestFeatureNavigate.class,
  TestFeatureUpdate.class
})

public class FeatureTestSuite {
  // the class remains empty,
  // used only as a holder for the above annotations
}
```

Actual class to be tested in your code

A test class



Another Test Suite Example

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HW2Tests {}
```

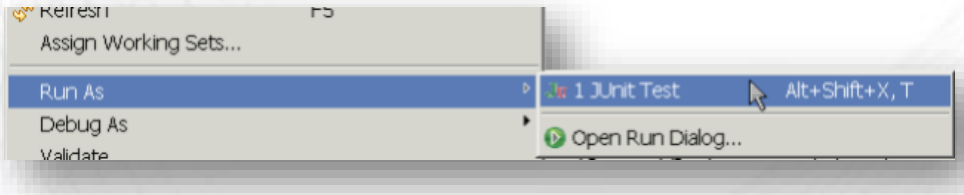
Actual SUT classes to be tested

Actual SUT to be tested

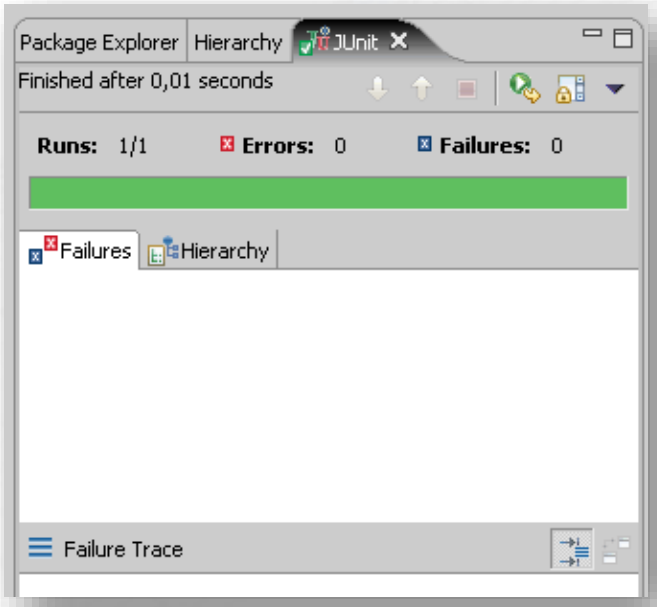
Running a Test

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

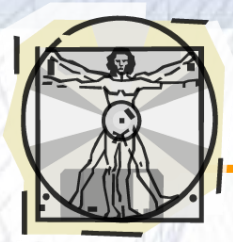
  Run As → JUnit Test



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

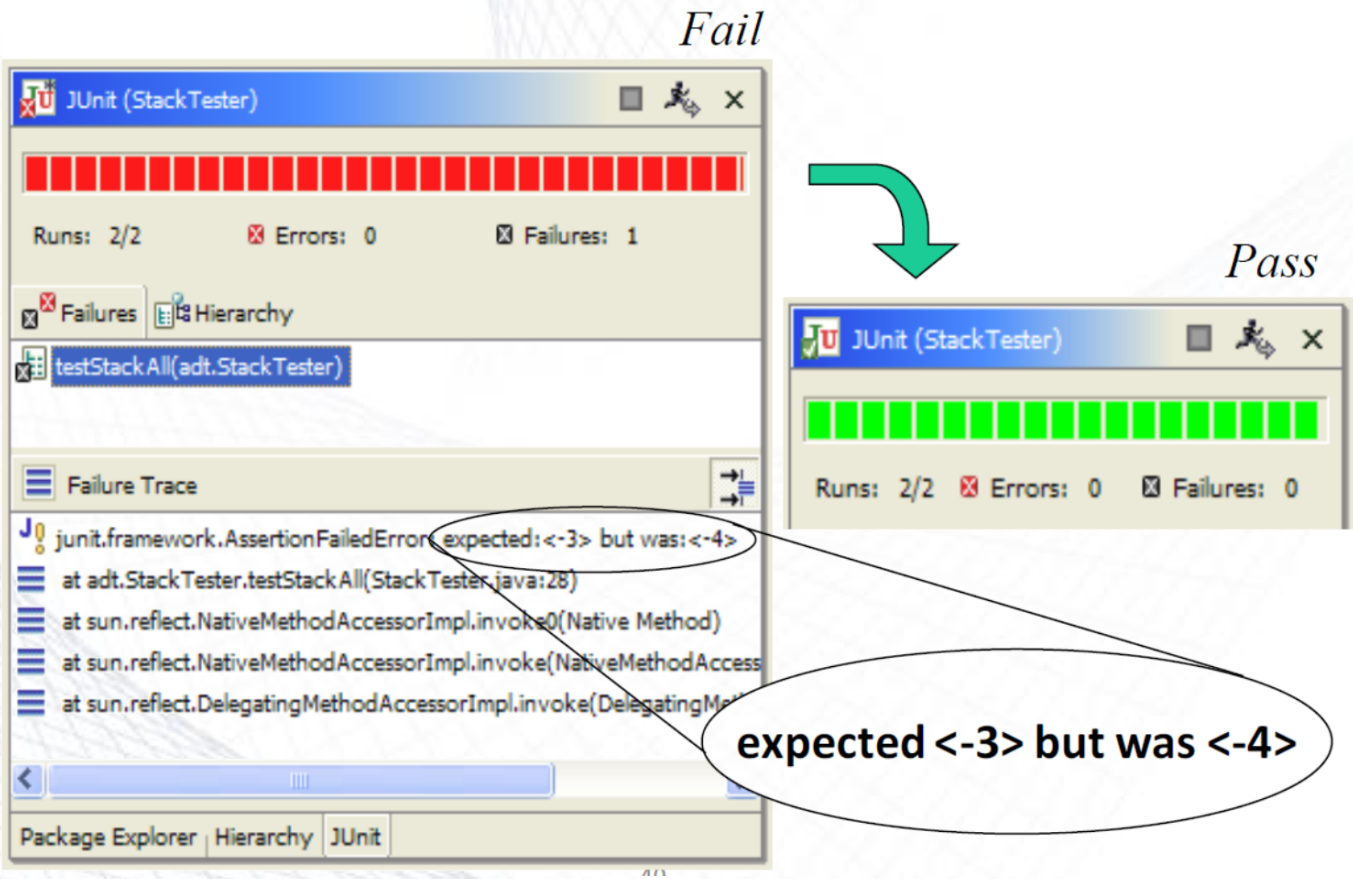


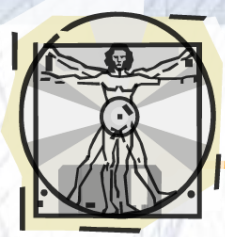
- The Failure Trace shows which tests failed, if any, and why



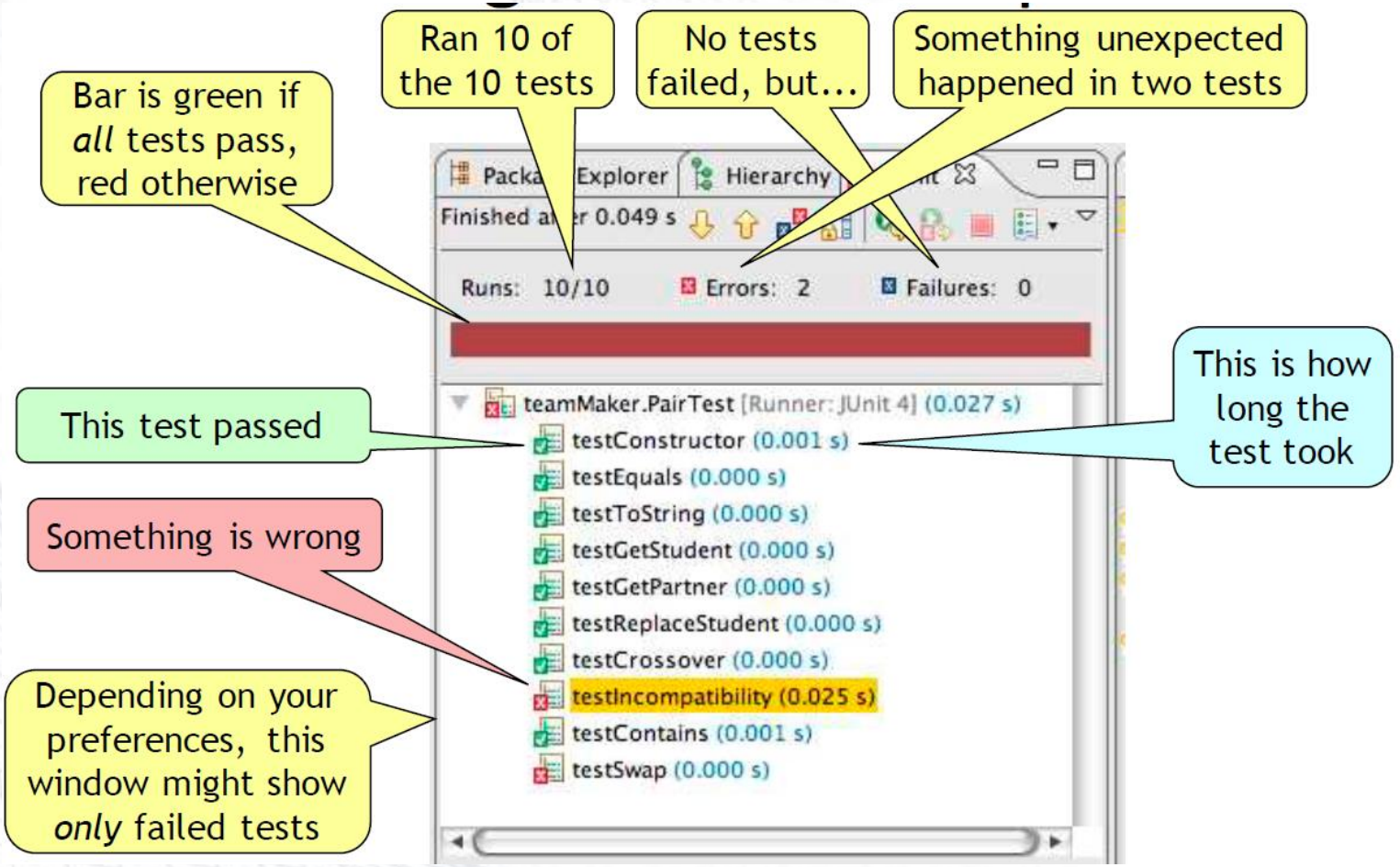
Running a Test

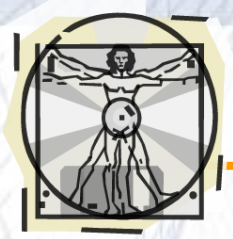
- Examine JUnit red bar





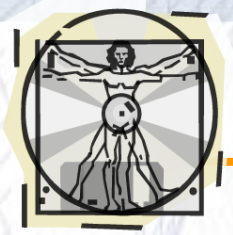
Viewing Results in Eclipse





Creating a Test Case

- In JUnit 3.x
  - Import junit.framework.*
  - Extend TestCase
  - Name the test methods with a prefix of 'test'
  - Validate conditions using one of the several assert methods
- In JUnit 4.x and later
  - Do not extend from Junit.framework.TestCase
  - Do not prefix the test method with "test"
  - Use one of the assert methods
  - Run the test using JUnit4TestAdapter
  - Use @NAME syntax (annotations)

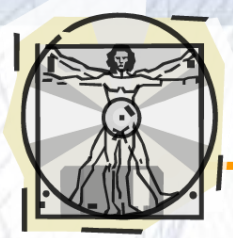


A JUnit Test Class

- 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 name {
    ...

    @Test
    public void name() {   // a test case method
        ...
    }
}
```

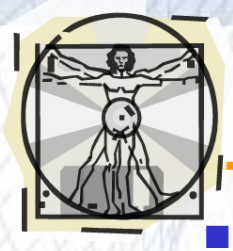


Writing methods in Test Case

- 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

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

Exercise
Verify

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

TearDown

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

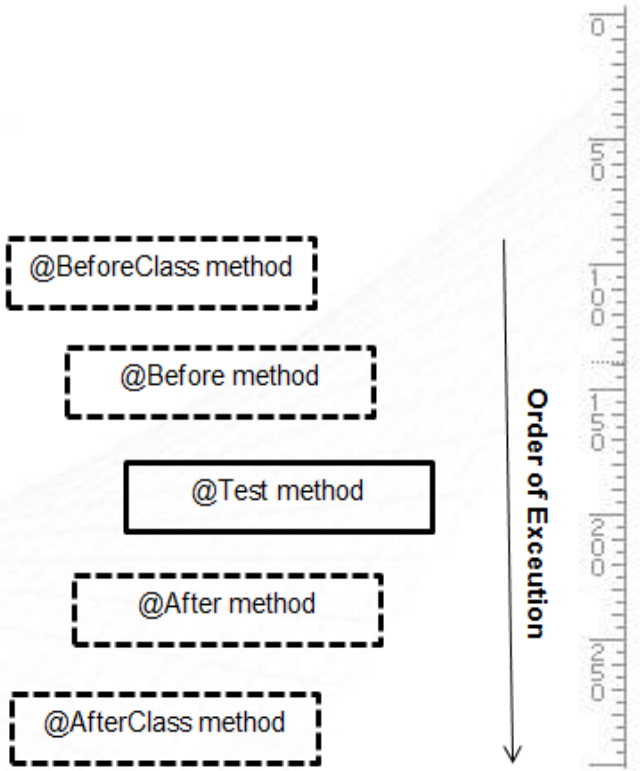
Setup and Teardown

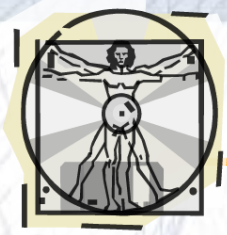
```
@Before
public void name() { ... }
@After
public void name() { ... }
```

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





Assertions

- Oracle

void assertEquals(boolean expected, boolean actual)

void assertTrue(boolean expected, boolean actual)

void assertFalse(boolean condition)

void assertNotNull(Object object)

void assertNull(Object object)

void assertSame(boolean condition)

void assertNotSame(boolean condition)

void assertArrayEquals(expectedArray, resultArray)

http://junit.sourceforge.net/javadoc/org/junit/Assert.html

JUnit Assert*() Methods

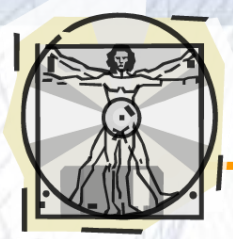
`assertTrue(`**"message for fail", condition**`)`	fails if the boolean message is `false`
`assertFalse(`**"message", condition**`)`	fails if the boolean message is `true`
`assertEquals(`**expected, actual**`)`	fails if the values are not equal
`assertSame(`**expected, actual**`)`	fails if the values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

Each method can also be passed a string to display/log if it fails:

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

JUnit Assert*() Methods

- For a boolean condition
  - **assertTrue("message for fail", condition);**
  - **assertFalse("message", condition);**
- For object, int, long, and byte values
  - **assertEquals(expected_value, actual_value);**
- For float and double values
  - **assertEquals(expected, actual, error);**
- For objects references
  - **assertNull(reference)**
  - **assertNotNull(reference)**
- Immediate fail
  - **fail()**



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

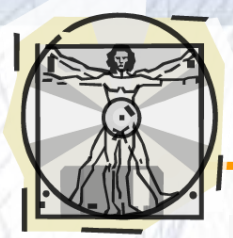
- Messages **helps documents** the tests
  - Messages provide **additional information when reading failure logs**
- Each assert method has an equivalent version that **does not take a message** – however, this usage is usually not recommended

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 **setup** and **teardown** methods in the test class

Test Code Smells

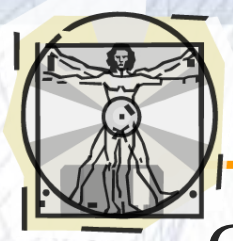
- Tests should be self-contained and not care about each other
  - "Smells" (bad things to avoid) in tests
    - Constrained test order:
      - Test A must run before Test B. (usually a misguided attempt to test order/flow)
      - Tests call each other: Test A calls Test B's method (calling a shared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



Test Code Smells

```
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 fix order

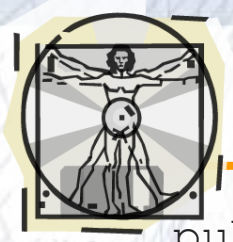


JUnit Exercise: Black Box

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()
```

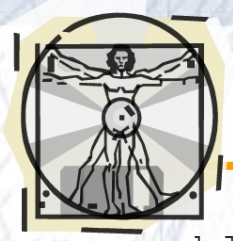
- Come up with unit tests to check the following:
  - That no `Date` object can ever get into an invalid state
  - That the `addDays` method works properly ← **Our focus**



What's Wrong with This?

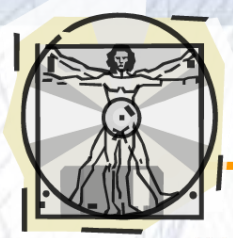
```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);
        assertEquals(d.getDay(), 19);
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);      // Test Feb for 28 days
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 3);
        assertEquals(d.getDay(), 1);
    }
```

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());   // 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
```

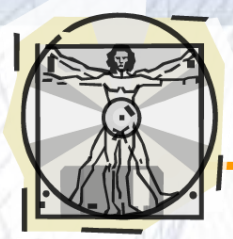


Expected Answer Objects

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d);  // use an expected answer
    }                               // object to minimize tests

                                    // (Date must have toString
    @Test                           //  and equals methods)
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

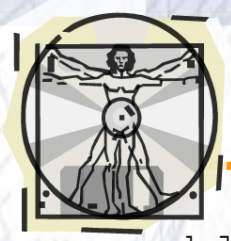
Naming Test Cases

```
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?

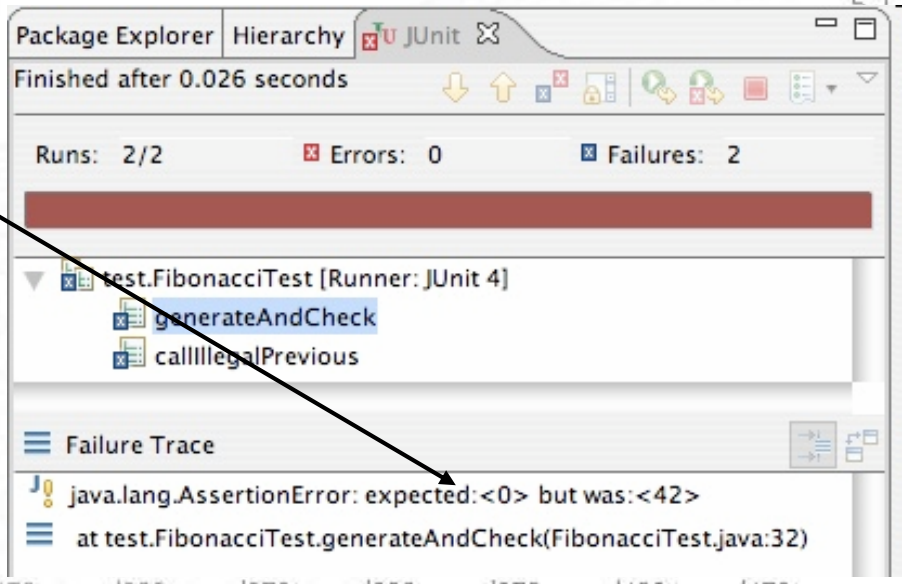
```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals(
            "should have gotten " + expected + "\n" +
            " but instead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



Good Assertion Messages

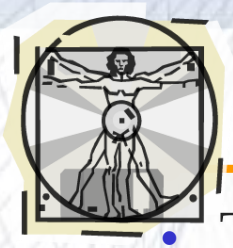
```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...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

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



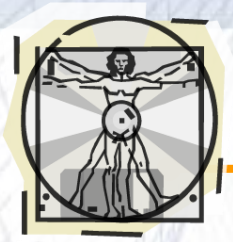
Annotations - Pervasive Timeouts

```
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;
}
```



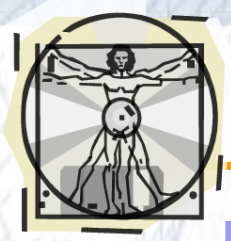
Trustworthy Tests

- 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

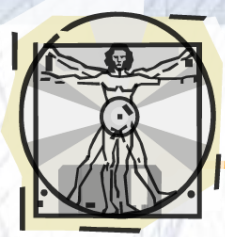
Test method

UNIVERSITY OF CALGARY

Section 4

Dependencies

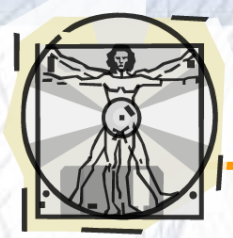
Stubbing – Mocking



Isolating Unit Tests

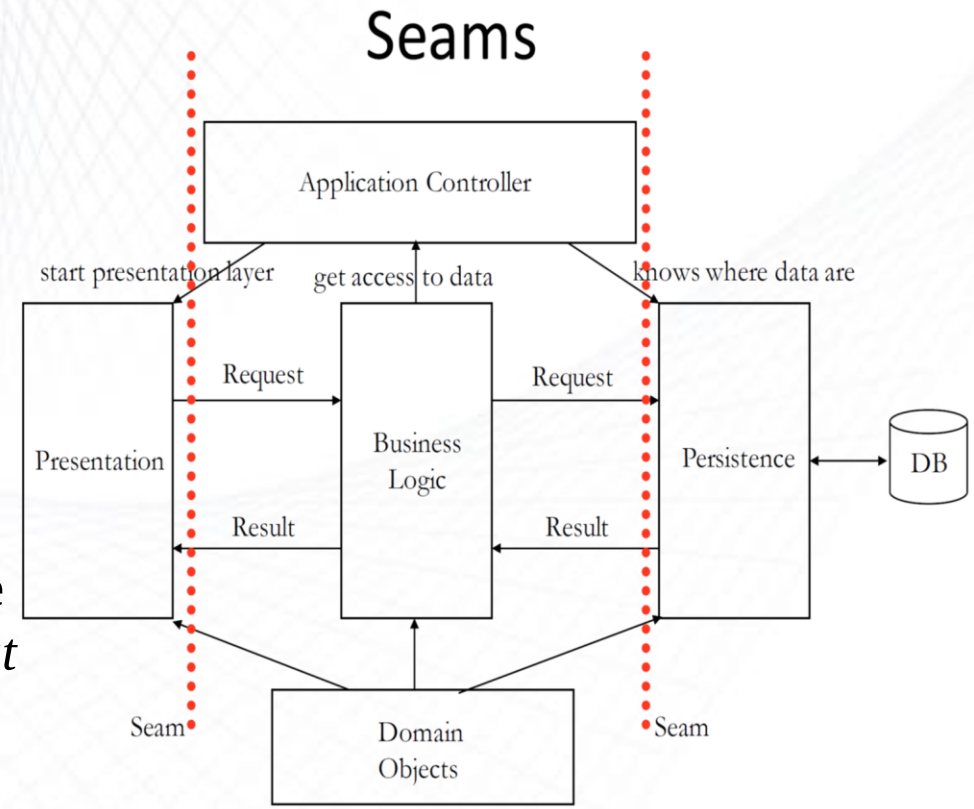
- SUT's that have **no collaborators** should be **straightforward** to test
- But typically units have dependencies
  - Remember UML's collaboration diagram

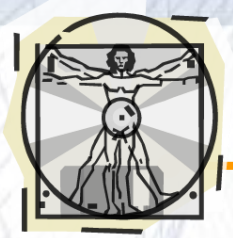
- Unit tests should NOT have dependencies, though (independent runs)
- Testing collaborative units together ← **feature test**



Seams

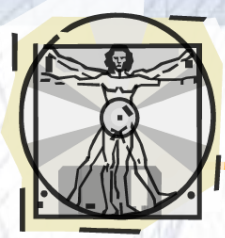
- In order to **avoid** accessing **external units**, there should be clear "**seams**" that separate units from each other
  - A seam is a place where you can **substitute alternative functionality**, e.g., *test doubles*





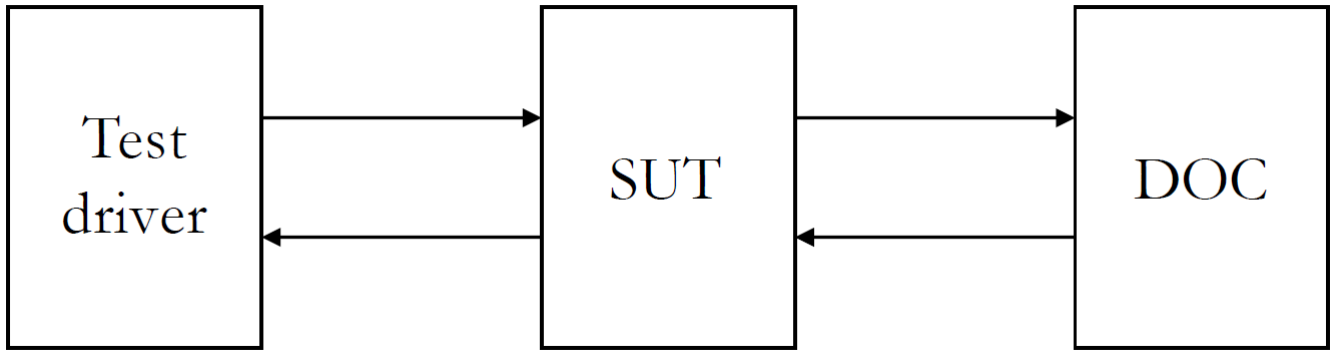
Doubles

- A test double is a **replacement** for a **DOC** (Dependent On Component)
- Example
  - A system might **send an email** if a particular request fails (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**"



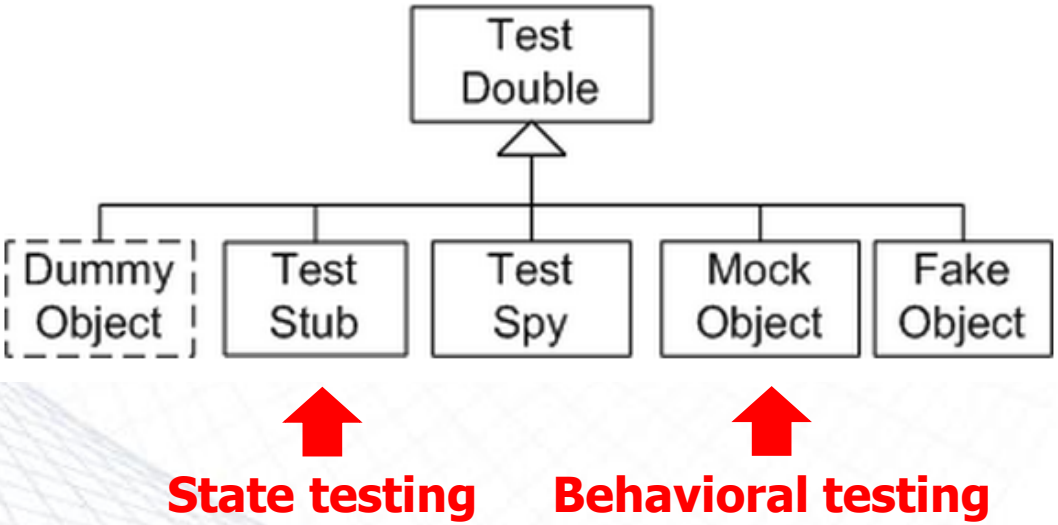
The SUT

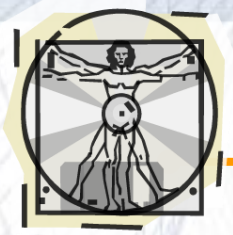
- SUT is the portion of a system that is being tested (may be one method or class or a collection of classes or components)
- Test Driver: Partial implementation of a component that depends on an SUT
- If the SUT collaborates with any other classes, those classes are referred to as "depended on components"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



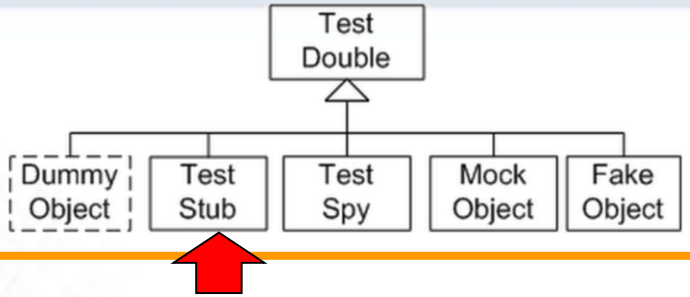
Test Doubles

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

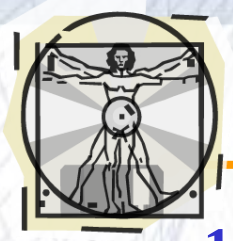




Stubs

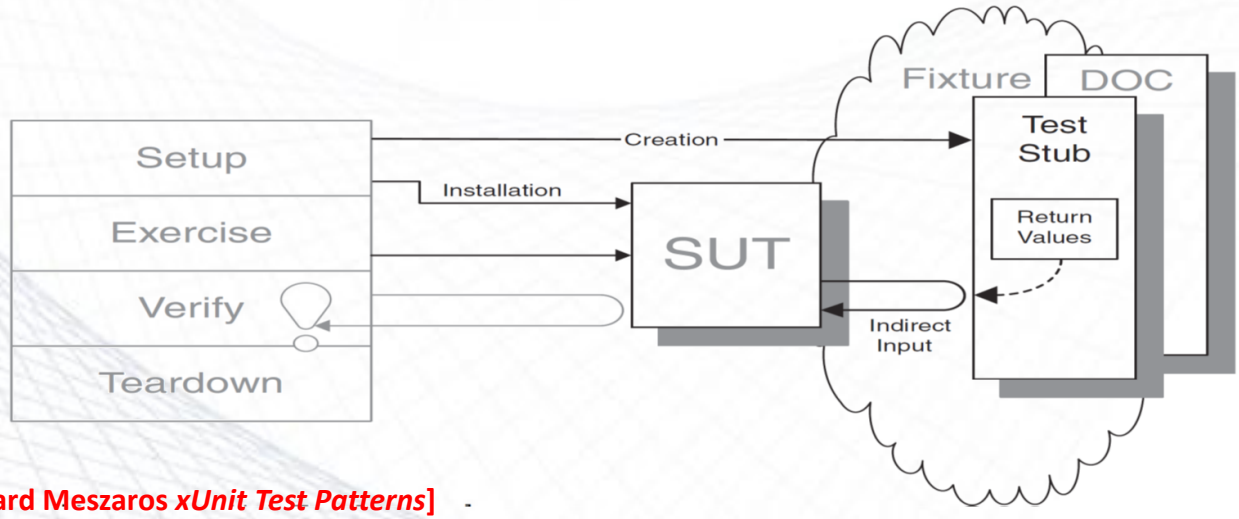


- A simple test double
- **Supplies responds to requests** from the SUT
  - Stub will contain **pre-defined responses** to specific requests
    - e.g. getStudent("100") causes a Student object to be returned
      - The Student object may be Student 100 or the same object may returned regardless of 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

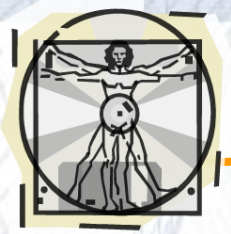


Test Lifecycle with Stubs

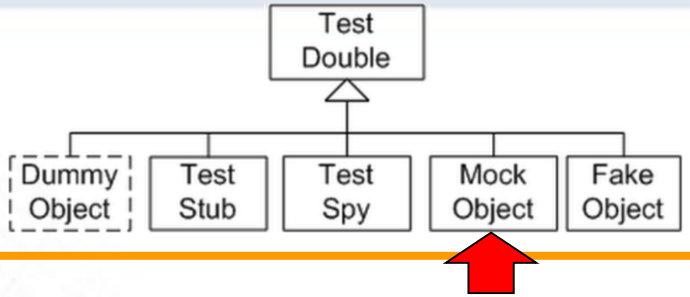
1. **Setup** - Prepare object that is being tested and its stubs collaborators ← **usually in @Before**
2. **Exercise** - Test the functionality ← **in @Test**
3. **Verify state** - Use asserts to check object's state
4. **Teardown** - Clean up resources ← **usually in @After**



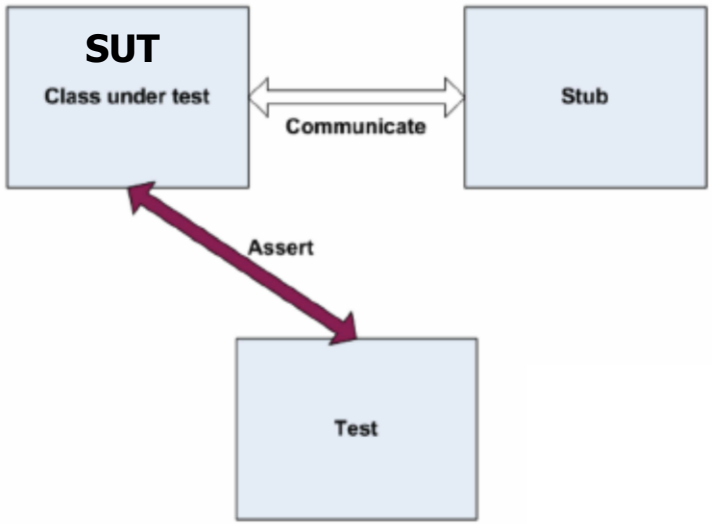
[Gerard Meszaros *xUnit Test Patterns*]

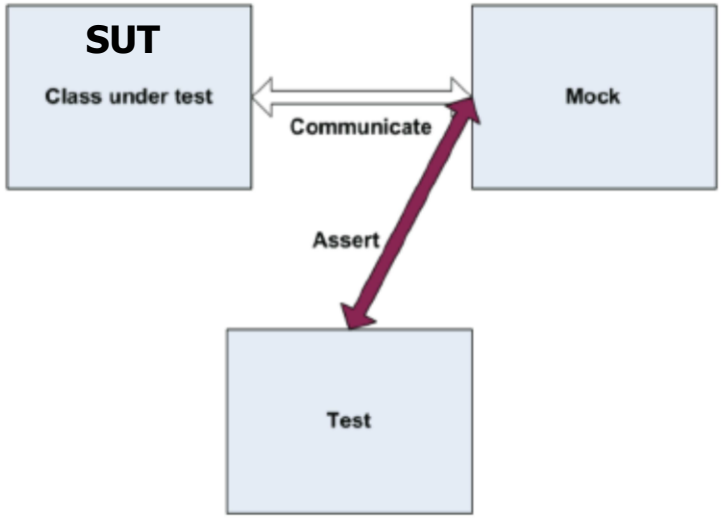


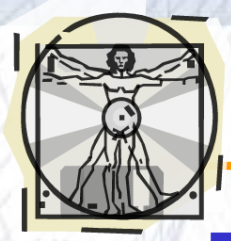
Mock Objects



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







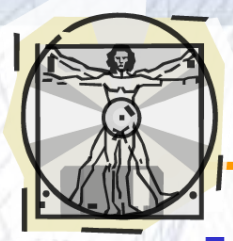
Mock Objects

- 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

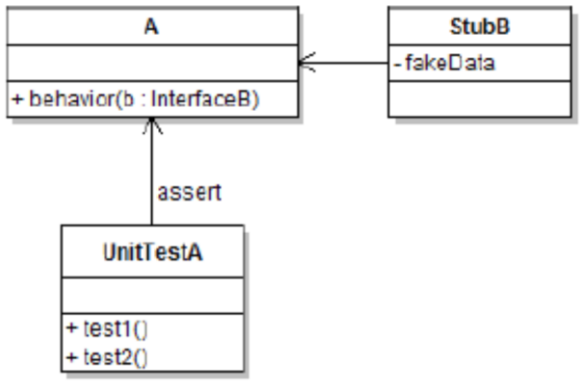
- **State testing:** *"What is the result?"*
  - Both mocks and stubs
- **Behavioral testing:** *"How the result has been achieved?"*
  - Only mocks

Learn to write mocks. They can be used everywhere!

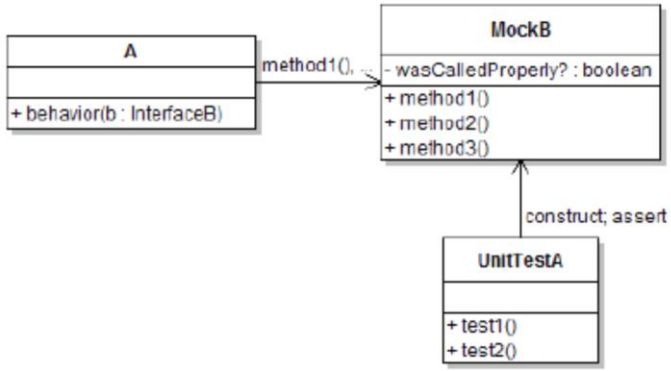


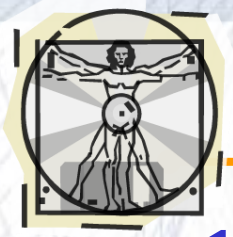
Mock vs. Stub

- 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

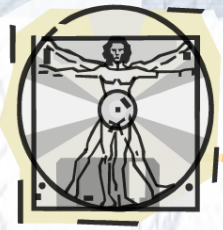




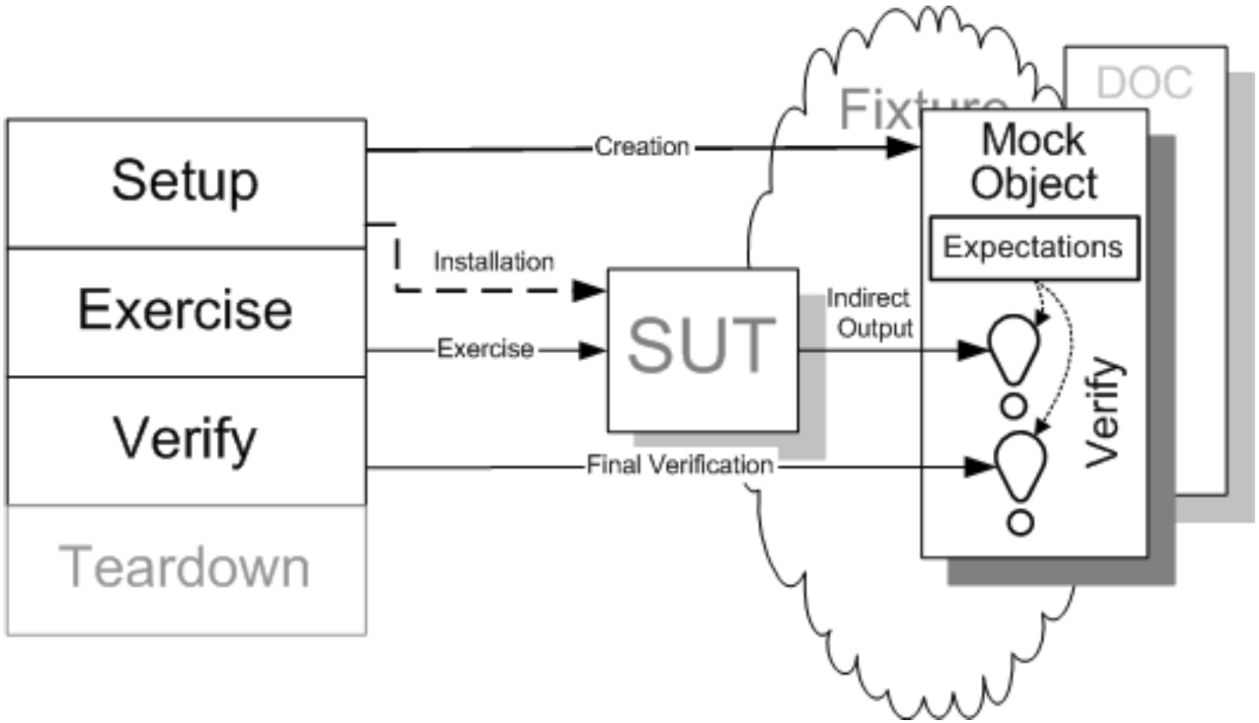
Test Lifecycle with Mocks

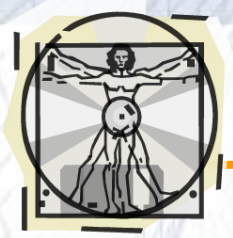
1. **Setup data -** Prepare object that is being tested
2. **Setup expectation -** Prepare expectations in mock that is being used by primary object
3. **Exercise -** Test the functionality
4. **Verify expectations -** Verify that correct methods has been invoked in mock
5. **Verify state -** Use asserts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

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



Mock - Summary

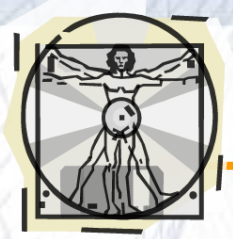




Mocking & Integration Testing

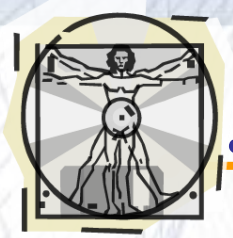
- The definitions are not very clear
- In this course when testing interactions between two objects:
  - Using a mock → unit testing
  - Using actual implementations → integration testing

We will talk about integration testing later



Mock Object Frameworks

- Stubs are often best created by hand/IDE
- Mocks are tedious to create manually
- Mock object frameworks help with the process
- Frameworks provide the following:
  - Auto-generation of mock objects that implement a given interface
  - Logging of what calls are performed on the mock objects
  - Methods/primitives for declaring and asserting your expectations

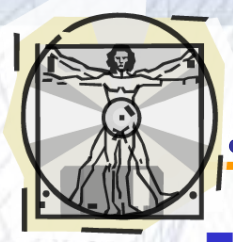


jMock

- How to test things that involve external components with jMock?
- jMock can be used together with JUnit not instead of it

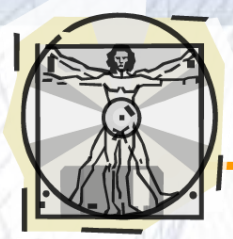
- Mock vs. stub

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



jMock 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 above accept a mock object and return 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();
    - "I expect that method1 will be called on mockB 3 times here"

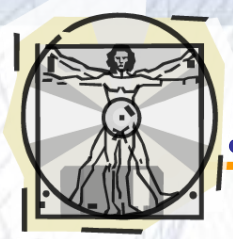


JMock Expected Actions

- will(action)
  - actions: returnValue(**v**), throwException(**e**)
- values:
  - equal(**value**), same(**value**), any(**type**),

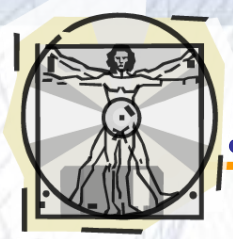
  aNull(**type**), aNonNull(**type**), not(**value**),

  anyOf(**value1, ..,valueN**)
  - – 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?

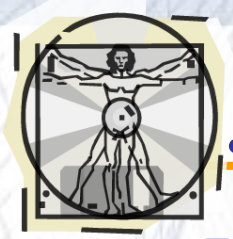
- With the help of jMock we can test a class independently when it depends on another class by creating a mock of dependent class
- What we need?
  - Eclipse and Junit and jMock jars included in the classpath



jMock How to Use it?

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

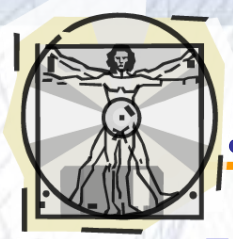
```
package test;
public interface ITestInterface
{
    public int test();
}
```



jMock How to Use it?

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

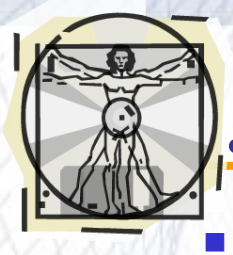
```
package test;

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



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;
     }
}
```



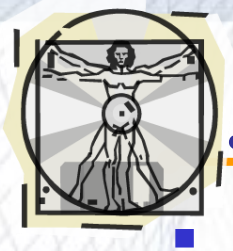
jMock How to Use it?

- 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
  }
}
```

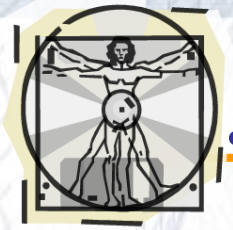
Mock object

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



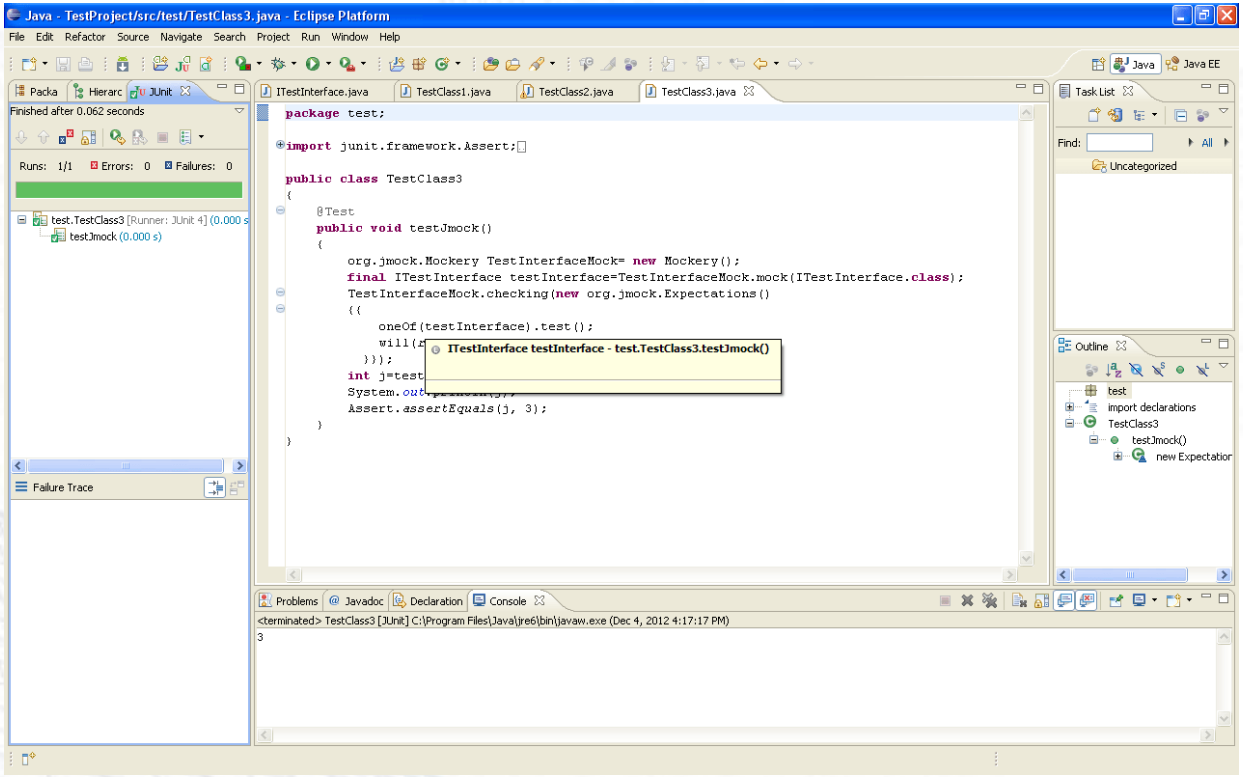
jMock How to Use it?

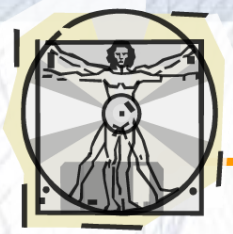
- **`TestClass1`** implements the interface **`ITestInterface`**
- Here **`TestClass2`** is dependent on **`TestClass1`**
- In unit testing we are not doing new of **`TestClass1`** in **`TestClass2`** We use mock object in unit testing it
- In **`TestClass3`** we are making the Mock object of **`ITestInterface`** 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



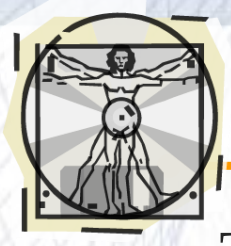


Mock - Sample

- Mockito

```
@BeforeClass
public static void setUpBeforeClass() throws Exception {

    ICalculator iCal = mock(ICalculator.class);
    when(iCal.add(-2.341115,-3.451113)).thenReturn(-5.792228);
    calcService = new calculatorService();
    calcService.setICalc(iCal);
}
```

http://site.mockito.org/



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, not always `assertTrue`
- Choose representative test cases from equivalent input classes
- Avoid complex logic in test methods if possible
- Use helpers, `@Before` to reduce redundancy between tests