

# **Software Design and Construction**

**159.251**

## **Software Testing**

Amjed Tahir

[a.tahir@massey.ac.nz](mailto:a.tahir@massey.ac.nz)

Original author: Jens Dietrich

# References

**[CC]** Robert Martin:

[Clean Code: A Handbook of Agile Software Craftsmanship.](#)

Prentice Hall 2009.

**[EJ]** Joshua Bloch:

[Effective Java Second Edition.](#)

Sun Micro 2008.

# Additional Readings

Kent Beck, Erich Gamma: The JUnit Cookbook

<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>

Robert Martin: The Three Rules of TDD

<http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>

# Summary

- the problem of program semantics
- assertions
- unit testing
- test-driven development
- testability as design goal
- black-box vs white-box testing
- mock testing
- boundary testing
- testing non-functional requirements
- user interface testing
- assessing test coverage
- end user testing

# The Problem of Semantics

- ***semantics*** - *the meaning of a word, phrase, sentence, or text* (Oxford Dictionary)
- mainly applies to methods
- objects - entity with state, state is changed by methods
- **How can be describe what methods do?**
  - describe state change
  - describe state before, and state after method invocations
  - use pre- and postconditions

# Example

```
public class Account {  
    private int amount = 0;  
    public void deposit(int sum)  
        ...  
}  
    public void withdraw(int sum)  
        ...  
}
```

deposit sum, the amount should increase by sum, sum must be >0, otherwise an `IllegalArgumentException` is thrown

withdraw sum, the amount should decrease by sum, and sum must be less than the amount, otherwise an `IllegalArgumentException` is thrown

# Semantics and Compiled Languages

- New programmers sometimes have misconceptions about programs

**"if it compiles, it works!"**

- based on the observation that compilers in languages with static types (like Java) will find and report many errors
- however, there are obvious limits to this

# Compiler Limitations

```
public class Math {  
    // add two numbers  
    public String add(int i,int j){  
        return 42;  
    }  
}
```

```
public class Math {  
    // add two numbers  
    public int add(int i,int j) {  
        return 42;  
    }  
}
```

```
public class Math {  
    // add two numbers  
    public int add(int i,int j) {  
        return i+j;  
    }  
}
```



# Compiler Limitations

```
public class Math {  
    // add two numbers  
    public String add(int i,int j){  
        return 42;  
    }  
}
```

the (Java) compiler will find this bug: the return type should be int, not string

```
public class Math {  
    // add two numbers  
    public int add(int i,int j) {  
        return 42;  
    }  
}
```

for the compiler, this is **correct!**  
**BUT** wrong semantic!

```
public class Math {  
    // add two numbers  
    public int add(int i,int j) {  
        return i+j;  
    }  
}
```

this is the intended version that has the correct **meaning** (=semantics)

# Embedding Semantics in Code

- design by contract (DbC) proposed by Bertrand Meyer:  
add pre- and postconditions ("**expects**" and "**ensures**")  
directly to method declarations
- part of the Eiffel programming language
- allows verification: prove that a system is correct with  
respect to these constraints
- but:
  - not available in mainstream Java (although some  
extensions like JML and contract4j exist)
  - verification difficult and may impose runtime  
overhead
  - writing precise contracts can be difficult
- We covered this fully in 159272!

# Java Assertions

- simple mechanism to add constraints to code
- syntax:
  - `assert condition`
  - `assert condition:message`
- condition must evaluate to a boolean (True or False)
- can be used to specify pre- and post conditions depending on where assertions are used within the method body
- but note that assertions might not always be on – for example, in case that precondition checks might be missed !!

# Java Assertions ctd

- by default, assertions are **not** checked at runtime
- they must be explicitly enabled by starting the JVM with the parameter `-ea`
- fine-grained switching is also supported (per package)
- if an assertion is violated (i.e., the condition evaluates to false), an `AssertionError` is thrown
- for more info, see:
  - <http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html>
  - [http://www.deitel.com/articles/java\\_tutorials/20060106/Assertions.html](http://www.deitel.com/articles/java_tutorials/20060106/Assertions.html)

# Example: Accounts with Asserts

```
public class Account {  
    private int TotalAmount = 0;  
    public void deposit(int amountToAdd) {  
        assert amountToAdd > 0;  
        int oldAmount = this.amountToAdd;  
        ...  
        assert amount > amountToAdd;  
  
        assert amount == oldAmount + amountToAdd;  
    }  
    public void withdraw(int sum) {  
        ...  
    }  
}
```

precondition

postcondition 1  
(coarse)

postcondition 2  
(better)

# Contract Patterns and APIs

- conditional runtime exceptions to check preconditions

```
if (sum<0) throw new IllegalArgumentException(...)
```

- contract APIs wrapping conditional preconditions:

- com.google.common.base.Preconditions
- org.apache.commons.lang3.Validate

- contract annotations (JSR303/349):

- javax.validation.constraints @Max, @Min, NotNull, ..

- checkerframework (annotations with static checks)

# Tests as Interpolation Semantics

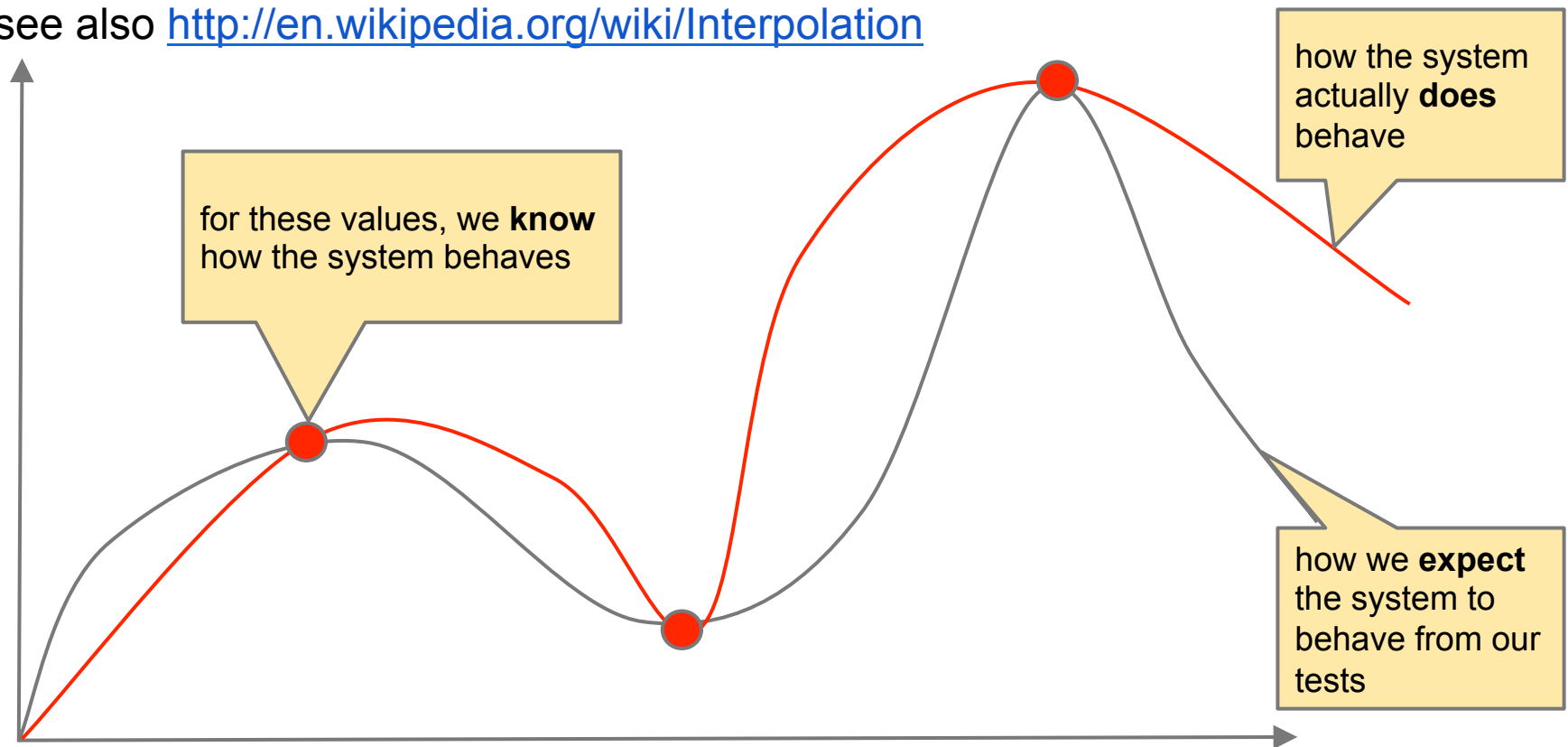
- it is hard to specify and prove correctness for all possible objects and values
- but what about if we can prove correctness for selected objects and values
- example:
  - amount of account is 100, deposit 10, results should be 110
  - amount of account is 0, deposit 42, results should be 42
  - amount of account is 100, deposit -10, result should still be 100, and an IllegalArgumentException is thrown
- then **infer** that this will work for other values as well!
- our test cases **do not prove** that the system behaves correctly, but they can provide a (potentially sufficient) approximation of correctness

# The Interpolation Metaphor

**Interpolation** (Mathematics): insert (an intermediate value or term) into a series by estimating or calculating it from surrounding known values.

Oxford Dictionary.

see also <http://en.wikipedia.org/wiki/Interpolation>





# Challenges

- is there a mechanism to write (a lot of) **inexpensive** tests to get many points for interpolation?
- are some test cases **better** than others to interpolate?
- how can we **recalibrate** tests if we find that our approximation is not sufficient?

# Consequences

- **test cases cannot prove correctness (the absence of bugs, compliance to specification)**
- the code that is being tested may not contain a bug, or the parameters used in the tests do not show the bug
- but test cases can prove **incorrectness**: the **presence of bugs**
- therefore, **a good test is a test that fails**

# Sourcing Tests

- tests from **requirements**: describe what user wants (e.g., example calculations)
- tests from **bug reports**: describe why/how the system fails
- tests serve two purposes:
  - **validation** - did we built the right system?
  - **verification** - did we built the system right?
- tests from requirements contribute more to validation
- tests from bug reports contribute more to verification

# Testing Level

- **Unit testing:** test the smallest units in the software ( classes, functions etc.)
- **Integration testing:** test multiple units together (multiple classes or separate functionalities)
- **System testing:** test the whole system together

# Writing Tests: JUnit

- JUnit is a popular library to write (unit) tests for Java
- developed by Kent Beck and Erich Gamma
- derived from SUnit for Smalltalk (also by Beck/Gamma) in the late 90ties
- integrations: IDEs(Eclipse, IntelliJ etc..), ANT, command line
- current version (annotation based): 5.\* - major differences to 3.\* !
- clones: various "X"Units exist for other languages (NUnit, PUnit, ...)
- Alternatives for Java: TestNG

# JUnit 4 Basics

- tests are public methods in a test class annotated with `@org.junit.Test`
- Test classes are the same as normal classes! test classes are not special in any other way: they don't have to subclass/implement a particular superclass/interface
- this makes it possible to add test methods to any class
- tests check conditions that express expectations
- this is done through static methods `org.junit.Assert.assert*` that are usually imported using static imports (i.e., that they can be used like functions within the test classes)

# JUnit Asserts

- asserts accept a message parameter - this will be used when assertion fails
- basic asserts:
  - `assertTrue`, `assertFalse`
  - `assertNull`, `assertNotNull`
- comparisons:
  - `assertEquals` - compares expected with computed value using `equals()`
  - `assertEquals` also works for primitives, for numerical datatypes a delta can also be used (difference between expected and actual is  $< \text{delta}$ )
  - `assertSame` - compares expected with computed value using `==`

# Test Oracles

- assertEquals/ assertEquals compare **expected** with **actual** values
- the expected values are also called **test oracles**
- oracle refers to the expected state of the system at the end of a test

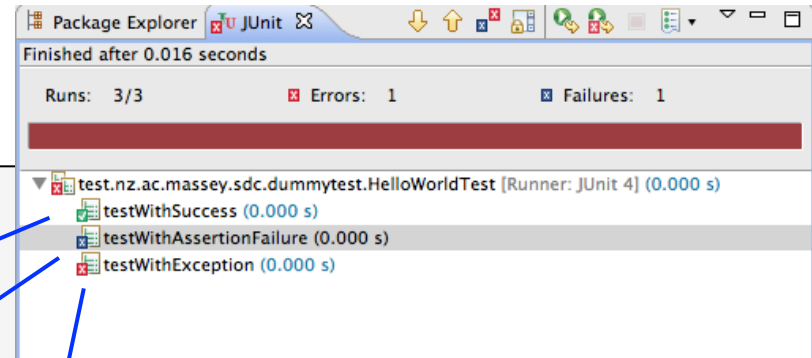


# JUnit Failing Tests

- if an assertion fails, an assertion error is generated
- JUnit catches this error, and marks the test as **failed**
- tests can also fail for another reason: if the execution of tests throws an exception
- JUnit runners are front ends for JUnits - user interfaces used to control and execute test
- example: graphical JUnit runners such as the Eclipse JUnit runner (an Eclipse view)
- graphical JUnit runners will visualise this differently, usually using colour codes:
  - **green** - test succeeds
  - **gray** - test fails due to `AssertionError`
  - **red** - tests fails due to `exception`

# Example: Possible Test Outcomes

```
public class HelloWorldTest {  
    @Test  
    public void testWithSuccess() {  
        assertTrue(true);  
    }  
    @Test  
    public void testWithAssertionFailure() {  
        assertTrue(false);  
    }  
    @Test  
    public void testWithException() throws Exception {  
        throw new java.io.IOException();  
    }  
}
```



JUnit runner in Eclipse

test methods  
can declare  
exceptions!

# Test Fixtures



photo from [http://theinspirationroom.com/daily/print/2008/10/mercedes\\_crash\\_test\\_mammogram.jpg](http://theinspirationroom.com/daily/print/2008/10/mercedes_crash_test_mammogram.jpg)

- Know as “Test setup”
- prepare: standardised environment: make tests repeatable and predictable
- tests should run in separation (should not interfere with each other)
- clean up after the test to restore environment

# Test Fixtures

- run a method before each test to prepare test:
  - annotated with `@org.junit.Before`
  - this is used to initialise the objects to be tested
- run a method after each test to clean up:
  - annotated with `@org.junit.After`
  - this is used to release/reset objects to be tested
- advanced:
  - static methods annotated with `@org.junit.BeforeClass` and `@org.junit.AfterClass`
  - these methods are executed once before/after the entire set of tests execute
  - these methods are mainly used for optimisation, and should be used with caution

# Case Study: Testing Tax Calculator

- algorithm: NZ uses a progressive tax system with tax brackets
- the parameters used are from 2012, see <http://www.ird.govt.nz/how-to/taxrates-codes/itaxsalaryandwage-incometaxrates.html> for details and examples
- assume we have to develop a tax calculator
- requirements: should implement the algorithm described on the IRD web site
- source code: <https://bitbucket.org/jensdietrich/oop-examples/src/1.0/junit-primer/>

# Tax Calculator Implementation

```
public class IncomeTaxCalculator {
    double[] brackets = {0,14000,48000,70000};
    double[] taxRates = {10.5,17.5,30,33};
    public double calculateIncomeTax(double income) {
        if (income<0) throw new IllegalArgumentException();
        double tax = 0.0;
        for (int i=brackets.length-1;i>=0;i--) {
            double bracket = brackets[i];
            double taxRate = taxRates[i];
            if (income>bracket) {
                double taxableInThisBracket = income-bracket;
                income = income - taxableInThisBracket;
                tax = tax + taxableInThisBracket*taxRate/100;
            }
        }
        return tax;
    }
}
```

# Tests

```
import org.junit.*;
```

import annotations: **@Before**, **@After**, **@Test** etc

```
public class IncomeTaxCalculatorTests {
```

```
    private IncomeTaxCalculator calculator = null;
```

**@Before**

```
    public void setup() {
```

```
        calculator = new IncomeTaxCalculator();
```

```
    }
```

**@After**

```
    public void tearDown() {
```

```
        calculator = null;
```

```
    }
```

set up test environment **before** each test is executed

clean up test environment **after** each test has been executed

# Tests (ctd)

this will import the assert\* methods used below

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

annotation from org.junit to mark this as a test

**@Test**

```
public void test1() {  
    double tax = calculator.calculateIncomeTax(65238);  
    assertEquals(12591.40, tax, 0.01);  
}
```

acceptable delta between expected and computed value is 0.01 (1c)

**@Test**

```
public void test2() {  
    double tax = calculator.calculateIncomeTax(45000);  
    assertEquals(6895.0, tax, 0.01);  
}
```

expected  
value

computed  
value

the actual test(s): calculate income for a given salary, and compare it with the expected outcome (data from IRD web site)



# Questions

There is no method to compare two doubles without a delta..

@Deprecated

```
public static void assertEquals(double expected, double actual)
```

delta - the maximum delta between expected and actual for which both numbers are still considered equal.

Do you know why?

# Questions

For delta, you may use really small number

```
private static final double DELTA = 1e-15;

@Test
public void testDelta(){
    assertEquals(123.4, 123.4, DELTA);
}
```

# What is a good Test Case?

- test cases should be **simple** and **readable**
- test cases should replace semantic comments
- if test cases are complex, then it becomes more likely that the test cases themselves are incorrect!
- **avoid procedural logic** (conditionals, loops) in tests as much as possible
- test **one concept at a time**, write many small "atomic" test cases
- this will make it much easier to **diagnose** software - to spot what causes a test to fail
- prefer to have just **one assertion per test** (although this can be unproductive, e.g., when testing larger data structures) – avoid Assertion Roulette!

# The One Assertion Rule

- can conflict with other principle such as DRY
- example: assume you have a Person class with lastName and firstName properties, and a method setFullName
- the idea is that you set a full name string set is parsed, and then firstName and lastName are set
- to test this, it would be practical to have two assertions (for the firstName and the lastName), otherwise you need to copy the code before the assertion

# Boundary Testing

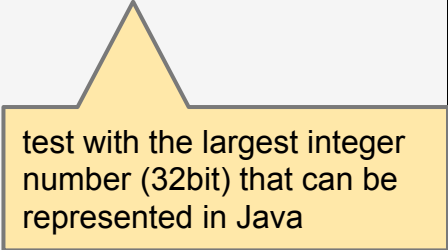
- test extreme/boundary values are more likely to fail
- problems tend to be "at the edges"
- therefore, they are more valuable tests
- tax calculator:
  - test with very small income
  - test with zero income
  - test with very large income
  - test with negative value (should throw exception)

# Boundary Testing Example

```
@Test
public void testZero() {
    double tax = calculator.calculateIncomeTax(0);
    assertEquals(0,tax,0.01);
}

@Test
public void testSmallIncome() {
    double tax = calculator.calculateIncomeTax(10);
    assertEquals(1.05,tax,0.01);
}

@Test
public void testLargeIncome() {
    double tax = calculator.calculateIncomeTax(Integer.MAX_VALUE);
    // should be very close to the max tax rate
    assertEquals(0.33*Integer.MAX_VALUE,tax,Integer.MAX_VALUE*0.001);
}
```



test with the largest integer number (32bit) that can be represented in Java

# Testing Exceptions (naive)

```
@Test
public void testNegativeIncome1() {
    try {
        double tax = calculator.calculateIncomeTax(-42);
        assertTrue(false);
    }
    catch (IllegalArgumentException x) {
        assertTrue(true);
    }
}
```

test fails if exception is **not** thrown

test succeeds if exception is thrown

# Testing Exceptions (better)

```
@Test(expected=IllegalArgumentException.class)  
    public void testNegativeIncome2() {  
        calculator.calculateIncomeTax(-42);  
    }
```

**declare** that the expected outcome is the exception

- JUnit 4+ has built-in support for testing expected exceptions
- this is a good example how annotations are used for a declarative programming style (declare **what** to do, **not how** to do it)



# Testing Non-Functional Properties

- aka "testing ilities"
- in general this is not straightforward, and relies on APIs to build assertions
- however, support for measuring runtime is built into JUnit
- use case: requirement that the tax calculator should calculate tax in <100ms
- use timeout property in test annotation!
- if tests timeout, they will **fail!**
- related:
  - **Stress tests** - test software with large data or transaction volumes to assess its **scalability** and **robustness** (fault tolerance)
  - **Smoke tests** - ensuring that the most important functions work correctly .

# Tests with Timeouts

```
@Test(timeout=100)
public void test1InclPerformance() {
    double tax = calculator.calculateIncomeTa
    assertEquals(12591.40,tax,0.01);
}
```

**declare** that the expected outcome is the exception

```
@Test(timeout=100)
public void test2InclPerformance() {
    double tax = calculator.calculateIncomeTax(45000);
    assertEquals(6895.0,tax,0.01);
}
```

- to “test the tests”, the calculation must be delayed
- this can be done by adding `Thread.sleep(1000)` to the `calculateIncome` method - this will cause the method execution to pause for 1s

# User Interface Testing

- (junit) testing can be extended to test user interaction
- frameworks like [abbot](#) can script and replay user interactions, and define asserts based on the state of user interface components
- popular package to test web user interfaces: [selenium](#)
- low level support for UI testing: [java.awt.Robot](#)
- this class allows to programmatically simulate a user interaction with IO hardware (keyboard, mouse)

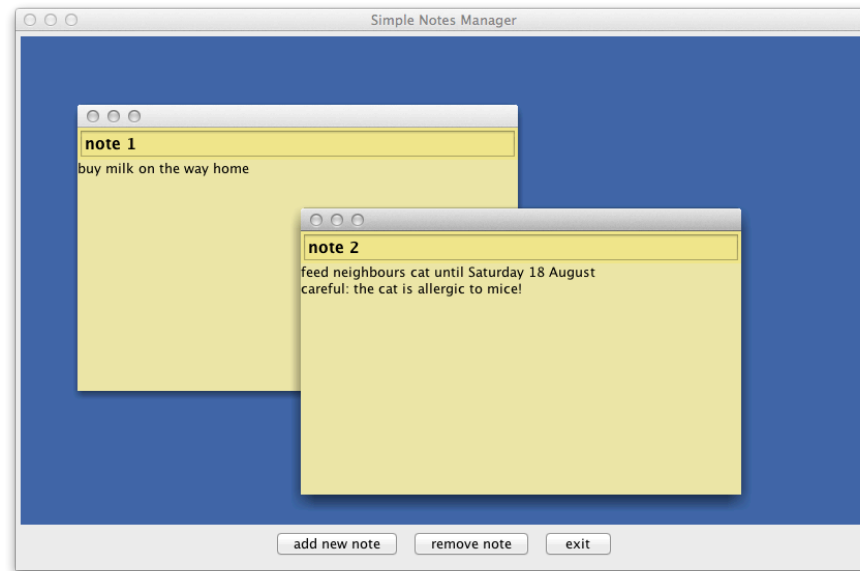
<http://abbot.sourceforge.net/doc/overview.shtml>

<http://seleniumhq.org/>

# User Interface Testing (ctd)

- while UI testing is possible, it is often better to focus more on testing other parts of the application
- UI testing is sometimes useful for **acceptance testing**:  
UI tests replay end user (client) interaction
- acceptance testing is part of validation (did we build the right system)

# Case Study: The Note Manager



- simple ui, designed with UI builder
- uses multi-document interface MDI - (windows within window)
- users can add and remove notes
- users can edit title and content (body) of each note
- source: <https://bitbucket.org/jensdietrich/oop-examples/src/1.0/gui-test/>

# UI Test: Fixture

```
public class TestNoteManager {  
    private NotesManager ui = null;  
    private Robot robot = null;  
    @Before  
    public void openNoteManager() throws Exception {  
        ui = new NotesManager();  
        ui.setVisible(true);  
        robot = new Robot();  
    }  
    @After  
    public void waitABit() throws Exception {  
        robot = null;  
        Thread.sleep(1000);  
    }  
}
```

this will open the application (the window will pop up)

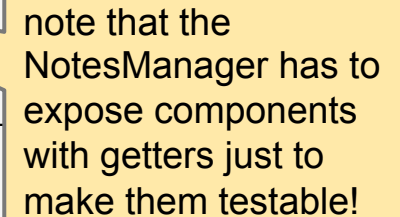
create a new robot to simulate user input

wait 1 s - this is for us to better see what is going on, otherwise windows close too quickly

# UI Test: Test Adding Notes

```
@Test
public void testAddOne() throws Exception {
    JButton btnAdd = ui.getBtnAddNote();
    btnAdd.doClick();
    assertEquals(1,ui.getNoteEditors().length);
}
```

```
@Test
public void testAddTwo() throws Exception {
    JButton btnAdd = ui.getBtnAddNote();
    btnAdd.doClick();
    btnAdd.doClick();
    assertEquals(2,ui.getNoteEditors().length);
}
```



note that the NotesManager has to expose components with getters just to make them testable!

# UI Test: Editing Notes

```
@Test
public void testAddAndEditTitle() throws Exception {
    JButton btnAdd = ui.getBtnAddNote();
    btnAdd.doClick();
    Thread.sleep(1000);
    NoteEditor editor = ui.getNoteEditors()[0];
    JTextField titleEditor = editor.getTitleEditor();
    clickOn(titleEditor);
    enterString("Hello world");

    // wait to give event delivery some time!
    Thread.sleep(1000);
    Note note = editor.getModel();
    assertEquals("Hello world", note.getTitle());
}
```

uses some utilities  
simulate input

wait to give event  
delivery some time!


the assert checks the  
model edited by the UI  
application




# UI Test: Using Robot

```
private Robot robot = null;
..
private void moveMouseTo(Component component) {
    Point p = component.getLocationOnScreen();
    robot.mouseMove(p.x+10, p.y+10);
}

private void clickOn(Component component) {
    component.requestFocus();
    robot.mousePress(InputEvent.BUTTON1_MASK);
}
```



move mouse  
programmatically

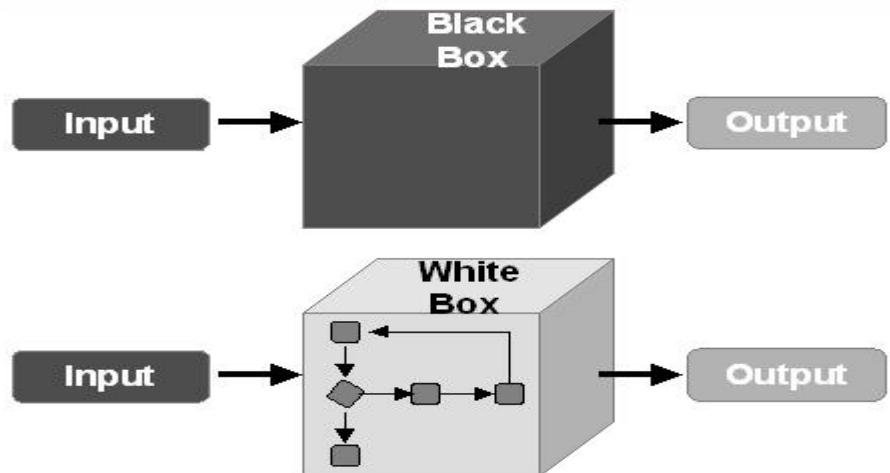


click mouse  
programmatically

# WhiteBox vs BlackBox Testing

- **Blackbox testing**
  - testing at the interfaces of software, internal structure is not revealed or unknown
- **Whitebox testing**
  - testing using knowledge about the internals of the software system

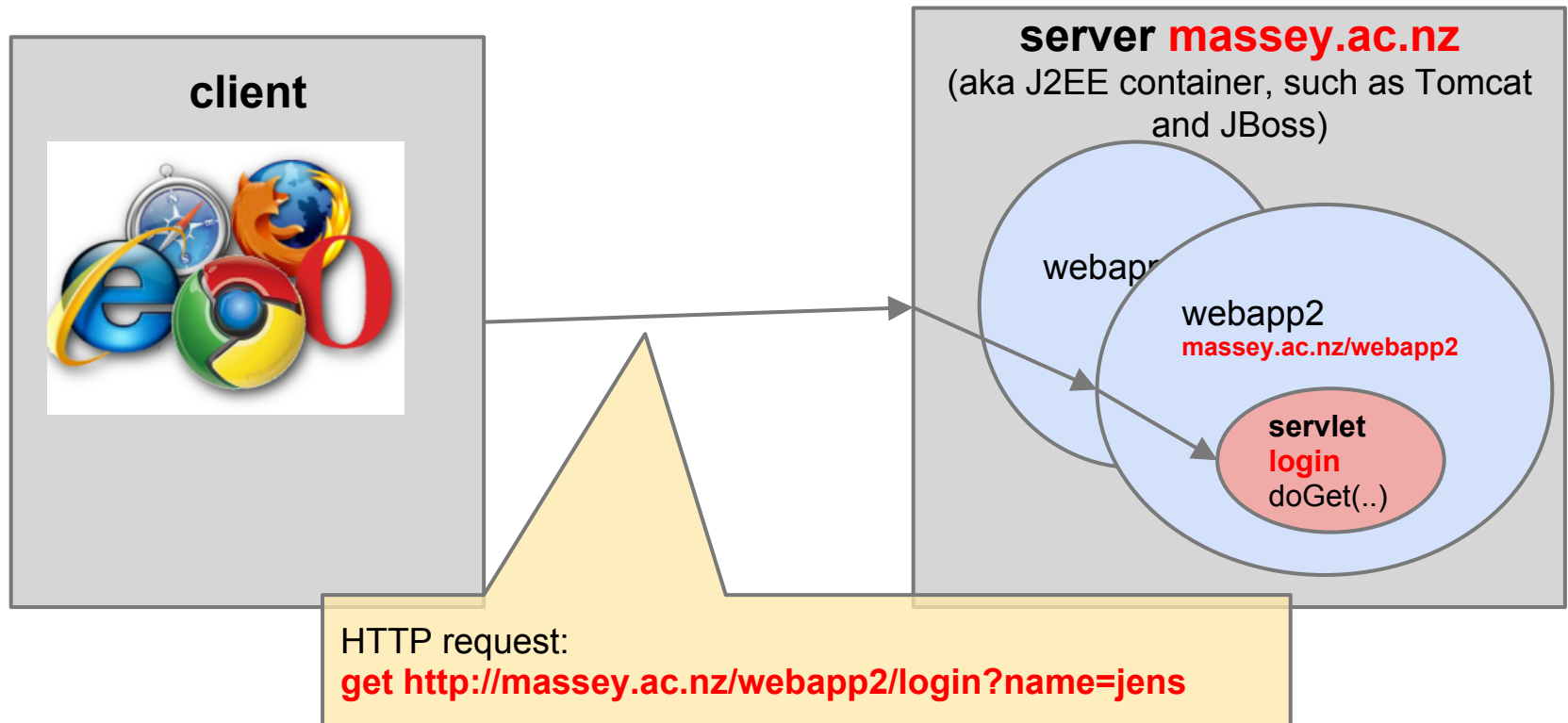
Comparison among Black-Box & White-Box Tests



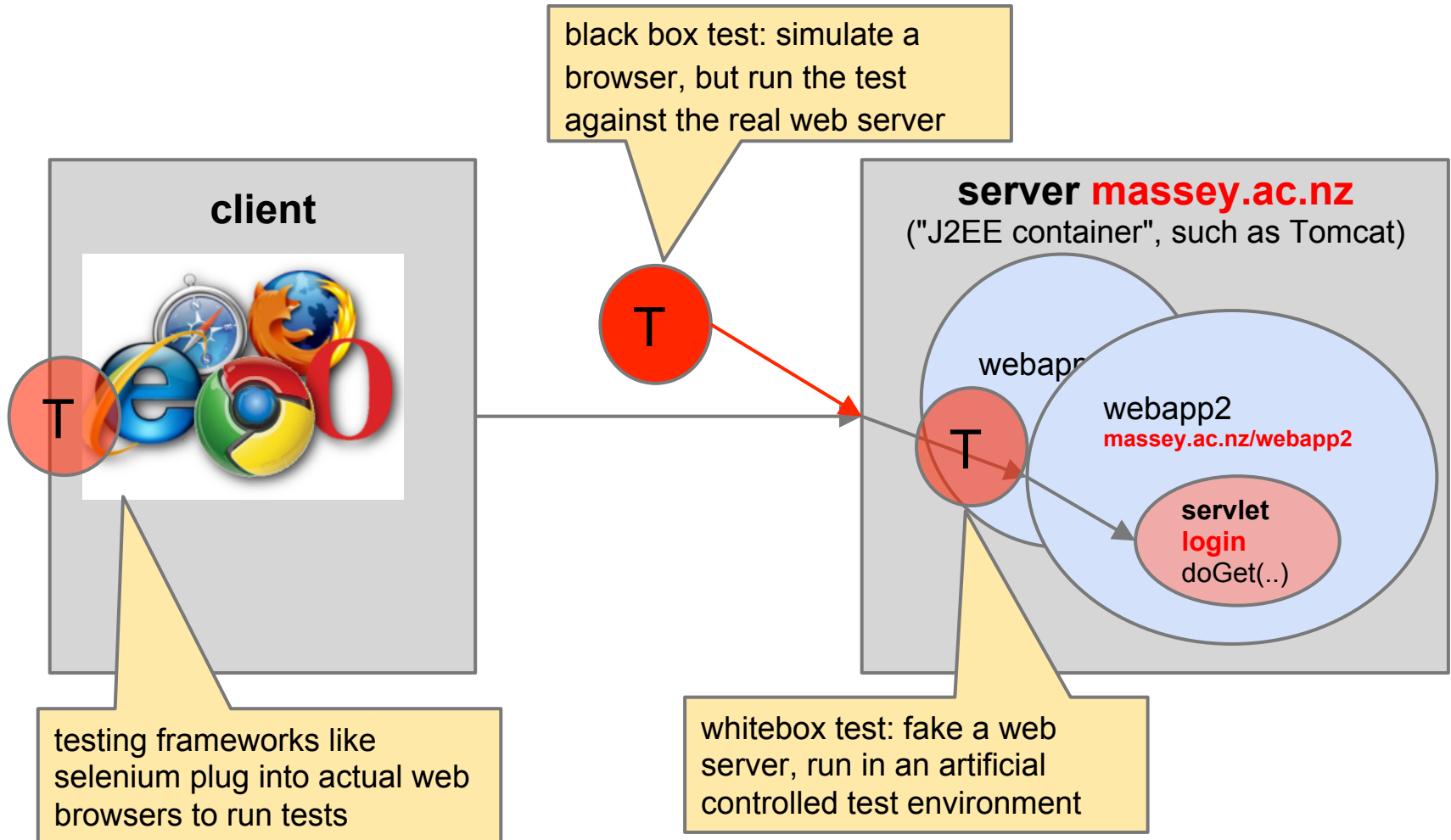
# Case Study: Testing Web Applications

- a Java web application is written using J2EE - the Java enterprise API
- a simple way to do this is to write a servlet: a simple java class that produces responses (usually web pages) for (http) requests coming from web browser sessions
- Java web apps are packaged as .. webapps
- web apps are then deployed within a J2EE compliant server
- there are several servers available, including Tomcat, Jetty, Glassfish and WebSphere
- the servers will route network requests to the servlets registered to handle them based on the address (URL) of the request

# A J2EE Web Application



# Where to Test



# Does your search engine know the answer?

- set up a **black box test** to figure out whether your search knows "[the answer to life the universe and everything](#)"<sup>1</sup>
- we expect to find "The answer to life the universe and everything = 42" in the response page
- we need a networking API to run the test
- basic networking API: java.net.URL
- better: [Apache HTTP client](#)

# Does your search engine know the answer?

- challenge: testing from behind a proxy
- simple solution: set flag `USE_PROXY` in test class
- source:  
<https://bitbucket.org/jensdietrich/oop-examples/src/1.0/blackbox-test/>
- this is a low level solution, libraries like [JWebUnit](#) should be used for production
- The example above uses Google, you can change the search into another browser such as Baidu or Bing

# Does your search engine know the answer? (ctd)

```
@Test(timeout=5000)
public void testFormSubmission() throws Exception {
    URIBuilder builder = new URIBuilder();
    builder.setScheme("http").setHost("www.google.com").setPath("/search")
        .setParameter("q", "the answer to life the universe and everything");
    URI uri = builder.build();

    HttpGet request = new HttpGet(uri);
    HttpResponse response = httpClient.execute(request);

    String content = EntityUtils.toString(response.getEntity());
    String expected = "The answer to life the universe and everything = 42";
    assertTrue(content.indexOf(expected) > -1);
}
```

use utilities to build a google query URL

execute query

result web site as one string

check whether the expected answer is returned



# Blackbox Tests

- advantage: test against a real server
- disadvantage:
  - expensive to setup tests to run against different servers –
  - requires server setup and maintenance
  - difficult to test part of the application in isolation
  - web site as string is too coarse for good tests, real-world web testing frameworks parse the page into a document object model (DOM), and provide a rich DOM-API to build assertions

# Whitebox Test

- test server "from the inside"
- instead of running the web app deployed inside a real server, fake a server
- the contract between servlets and server mainly consists of abstract types defined in the J2EE specification
- a server has to implement these types, and multiple servers do this
- by definition, everything that implements those abstract types correctly is a server
- idea: build a fake server just for testing
- this approach is called **Mock Testing**, and is easy and inexpensive

# Assumptions

- **assertions** represent **postconditions** - they are used to check the state of objects after some methods were invoked
- what about **preconditions?** - conditions that must be true for the test to make sense
  - JUnit uses **assumptions** for this purpose, in particular the method **Assume.assumeTrue(expression)**, where the (boolean) expression expresses the prerequisite

# Assumptions (ctd)

[assumptions are] a set of methods useful for stating assumptions about the conditions in which a test is meaningful. A failed assumption does not mean the code is broken, but that the test provides no useful information.

The default JUnit runner treats tests with failing assumptions as **ignored**.

from <http://junit.sourceforge.net/javadoc/org/junit/Assume.html>

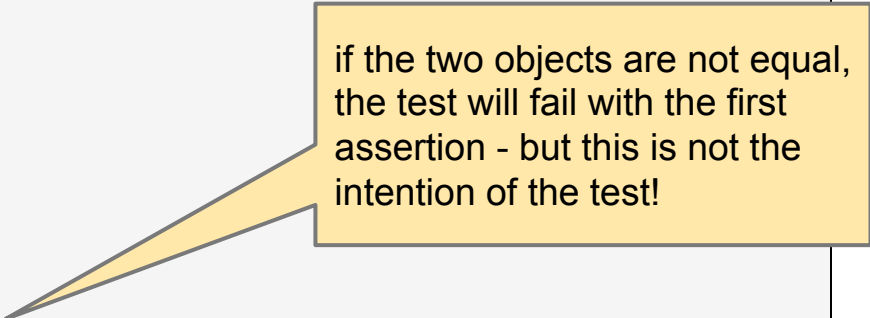
Note: the Eclipse test runner will mark tests with failed assumptions as **passed**!

# Assumptions Example

- scenario: you have a class `Foo` with implementations of `equals` and `hashCode` that are inconsistent
- write a test that shows that two objects are equal but have different hash codes
- i.e., this test case **should fail**
- remember: a good test is a test that fails
- such a test **proves** that `equals` and `hashCode` are inconsistent

# Example ctd: Two Assertions

```
public class TestFoo {  
    private Foo foo1 = null;  
    private Foo foo2 = null;  
    @Before  
    public void setUp() {  
        foo1 = ..;  
        foo2 = ..;  
    }  
    @Test  
    public void test() {  
        assertEquals(foo1, foo2);  
        assertEquals(foo1.hashCode(), foo2.hashCode());  
    }  
}
```

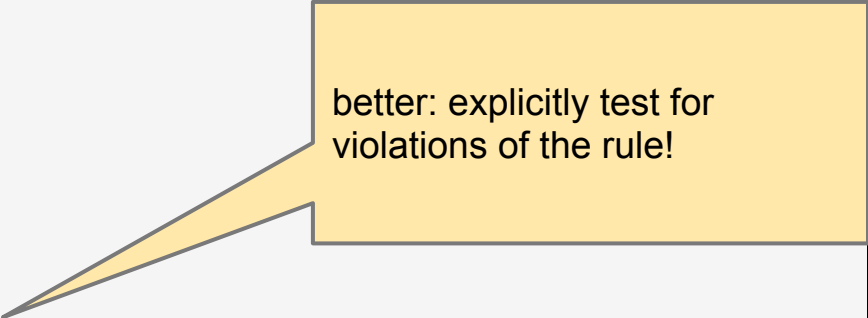


if the two objects are not equal,  
the test will fail with the first  
assertion - but this is not the  
intention of the test!

# Example ctd:

## One (Complex) Assertion

```
public class TestFoo {  
    private Foo foo1 = null;  
    private Foo foo2 = null;  
    @Before  
    public void setUp() {  
        foo1 = ..;  
        foo2 = ..;  
    }  
    @Test  
    public void test() {  
        assertTrue(  
            foo1.equals(foo2) &&  
            foo1.hashCode() !=foo2.hashCode() ;  
        )  
    }  
}
```



better: explicitly test for violations of the rule!

# Example ctd:

## One Assertion + one Assumption

```
public class TestFoo {  
    private Foo foo1 = null;  
    private Foo foo2 = null;  
    @Before  
    public void setUp() {  
        foo1 = ..;  
        foo2 = ..;  
    }  
    @Test  
    public void test() {  
        assumeThat(foo1.equals(foo2)) ;  
        assertEquals(foo1.hashCode(), foo2.hashCode());  
    }  
}
```

if the two objects are not equal,  
the assumption will fail and the  
test **will "be ignored"** - in  
Eclipse this means that the test  
will actually pass



# Dealing with Failed Assumptions

- the Eclipse test runner marks tests with failed assumptions as passed (success)
- the logic behind this is as follows: a test can be seen as rule:  
"if precondition(s), then postcondition(s)", or:  
"if assumptions [are true], then assertions [must be true]"
- the definition of the implication is (classical propositional logic) is:  
 $A \text{ implies } B := \text{not } A \text{ or } B$  (see also [https://en.wikipedia.org/wiki/Material\\_conditional](https://en.wikipedia.org/wiki/Material_conditional))
- i.e. such a rule is always true if the prerequisite (=the assumption) **is false**

# Coverage

- how can we assess which code is tested?
- we need a tool to figure out which parts of the code are visited by tests
- solution: measure **test coverage**
  - which classes are tested?
  - which methods are tested?
  - which blocks / lines of code are tested?

# Example

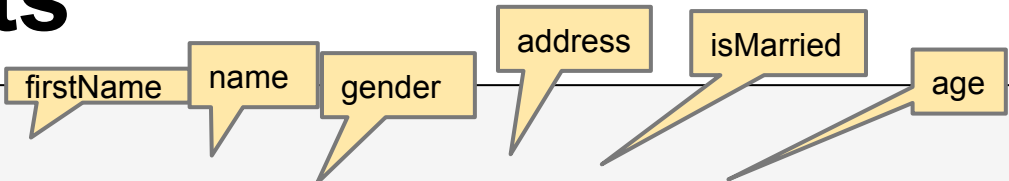
- POJO classes `Person`, `Address`
- `Person` has properties `firstName`, `name`, `age`, `gender`, `address`, and `isMarried`
- getters/setters, custom constructors, `equals` and `hashCode`
- custom method `getFullName()` in `Person`, prints salutation, first name and (last) name
- source:  
<https://bitbucket.org/jensdietrich/oop-examples/src/1.0/coverage/>

# Example: Logic

```
public String getFullName () {  
    String namePart = this.getFirstName() + " " + this.getName();  
    String salutationPart = null;  
    if (gender==Gender.MALE) {  
        salutationPart = "Mr.";  
    }  
    else if (gender==Gender.FEMALE) {  
        if (this.isMarried) {  
            salutationPart = "Mrs.";  
        }  
        else if (age<19){  
            salutationPart = "Miss";  
        }  
        else {  
            salutationPart = "Ms.";  
        }  
    }  
    return salutationPart + " " + namePart;  
}
```

fairly complex logic to  
implement english salutation  
rules based on gender,  
marital status and age

# Example: Tests



```
@Test
public void test1() {
    Person p = new Person("Tim", "Smith", Gender.MALE, null, true, 22);
    assertEquals("Mr. Tim Smith", p.getFullName());
}

@Test
public void test2() {
    Person p = new Person("Kate", "Smith", Gender.FEMALE, null, true, 22);
    assertEquals("Mrs. Kate Smith", p.getFullName());
}

@Test
public void test3() {
    Person p = new Person("Kate", "Smith", Gender.FEMALE, null, false, 17);
    assertEquals("Miss Kate Smith", p.getFullName());
}
```

Problem: no test case for salutation **Ms**

# Tooling: Emma

- there are several **test coverage tools**
- examples (Java): [Clover](#), [EclEmma](#), [CodeCover](#),
- emma is good open source coverage tool:  
<http://emma.sourceforge.net/>
- integrations: Eclipse, ANT
- Eclipse & IntelliJ integration: EclEmma:  
<http://www.eclemma.org/>

# Tooling: Emma (ctd)

- Emma supports the following coverage metrics: class, method, block, code
- using Emma in Eclipse: **right click on Project > Coverage As**
- Emma will then run all tests, register the executes methods, and highlight code that has been executed
- Emma can generate (html) reports
- generating reports in Eclipse:  
File > Export > Java > Coverage Report

# Emma Annotations (in Eclipse)

```
public String getFullName () {  
    String namePart = this.getFirstName() + " " + this.getName();  
    String salutationPart = null;  
    if (gender==Gender.MALE) {  
        salutationPart = "Mr.";  
    }  
    else if (gender==Gender.FEMALE) {  
        if (this.isMarried) {  
            salutationPart = "Mrs.";  
        }  
        else if (age<19){  
            salutationPart = "Miss";  
        }  
        else {  
            salutationPart = "Ms.";  
        }  
    }  
    return salutationPart + " " + namePart;  
}
```

green: code was  
executed  
("visited") by tests

red: code was  
**not** executed  
("visited") by tests

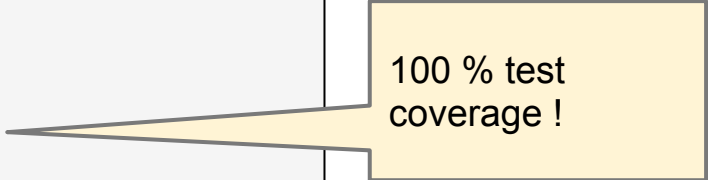


# How Coverage Tools Work

- coverage tools manipulate (**instrument**) the program code (source or byte code)
- in each block or line, a method invocation (something like **emma.visited (aUniqueLabel)**) is added
- a database of all labels is created
- when the program runs (e.g., test cases are executed), some of the visited(label) methods are called - this "ticks off" labels in the database
- metrics are computed by comparing ticked off (visited) labels with unvisited labels
- Emma instruments byte code
- Java libraries exist that facilitate byte code instrumentation , e.g. Apache BCEL

# High Coverage = Good Tests ?

```
public boolean isEven(int i) {  
    if (i%2==0)  
        return true;  
    else  
        return false;  
}  
...  
@Test public void test1() {  
    isEven(2);  
    assertTrue(true);  
}  
@Test public void test2() {  
    !isEven(1);  
    assertTrue(true);  
}
```



100 % test  
coverage !

# High Coverage = Good Tests ? ctd

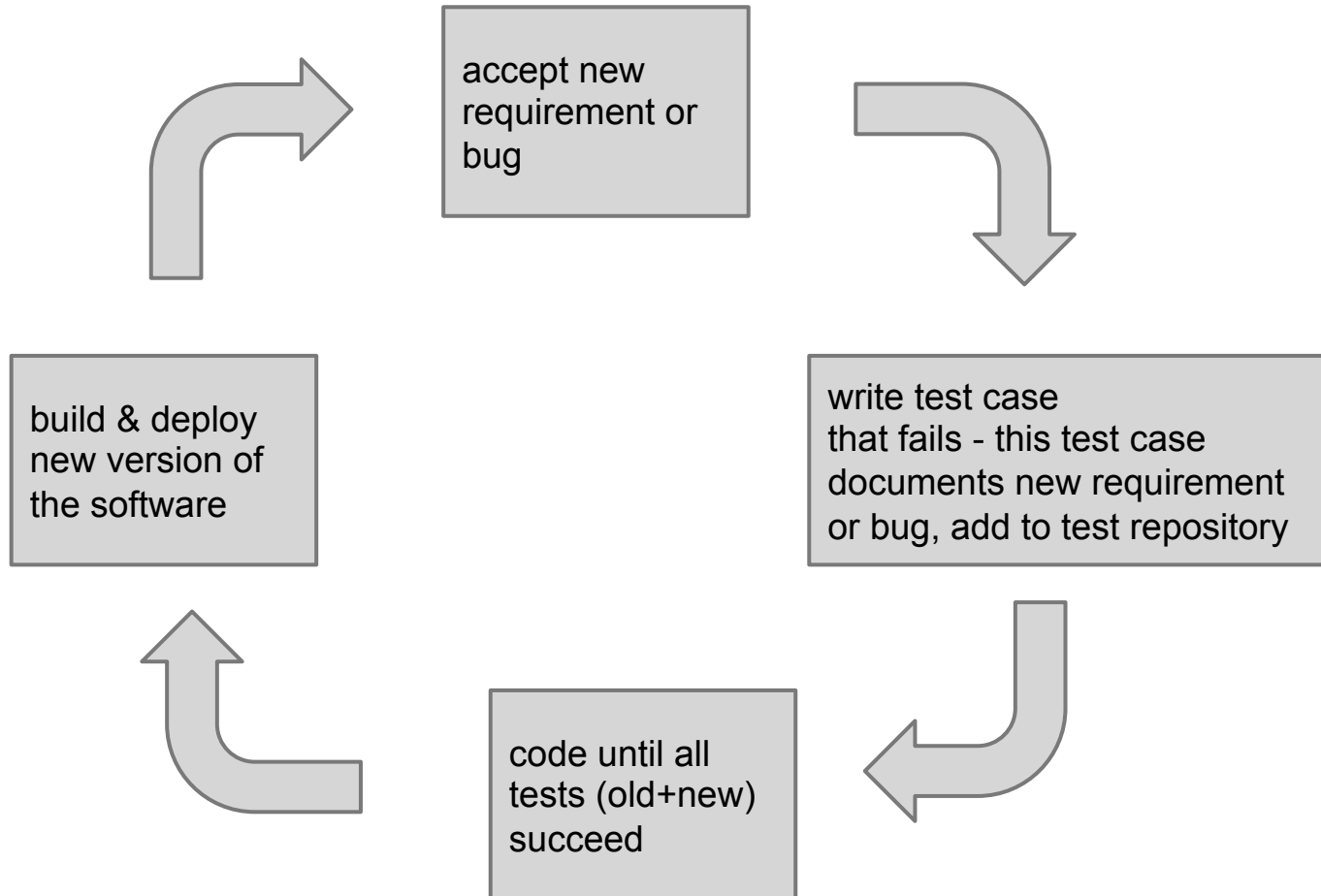
```
public boolean isEven(int i) {  
    if (i%2==0)  
        return true;  
    else  
        return false;  
}  
...  
@Test public void test1() {  
    isEven(2);  
    assertTrue(true);  
}  
@Test public void test2() {  
    !isEven(1);  
    assertTrue(true);  
}
```

but **poor assertions** -  
tests are  
almost  
meaningless  
*(they still test  
that no  
runtime  
exceptions  
occur .. )*

# From Testing to Test-Driven Development (TDD)

- This will be in part 2 of the course (Agile thinking)
- Important questions: **when to test?**
- in many traditional software (SDLC) , testing is done **after** the software has been written – this is called “test last”
- however, if test cases are seen as part of the specification, it makes sense to write them first
- this is called test-driven development (TDD)
- TDD is part of many modern "agile" methodologies, such as extreme programming

# TDD



# How to Write Tests First

- write abstract types, and write tests against the abstract types
- use dummies
- write tests against classes and methods that do not yet exist (the tests will fail because they cannot be compiled!)

# Example: Tests First - Step 1

```
public class IncomeTaxCalculator {  
    public double calculateIncomeTax(double income) {  
        return 42;  
    }  
}
```

- write minimal code, enough to support tests that can be compiled
- this code is not correct!
- this is the "use dummies" approach

# Example: Tests First - Step 2

```
public class IncomeTaxCalculatorTests {  
    private IncomeTaxCalculator calculator = null;  
    @Before  
    public void setup() {  
        calculator = new IncomeTaxCalculator();  
    }  
    @Test  
    public void test1() {  
        double tax =  
            calculator.calculateIncomeTax(65238);  
        assertEquals(12591.40, tax, 0.01);  
    }  
    ...  
}
```

- write tests against the code from step 1
- tests can be compiled, but most will fail!



# Example: Tests First - Step 3

```
public class IncomeTaxCalculator {  
    double[] brackets = {0,14000,48000,70000};  
    double[] taxRates = {10.5,17.5,30,33};  
    public double calculateIncomeTax(double income) {  
        if (income<0) throw new IllegalArgumentException();  
        double tax = 0.0;  
        for (int i=brackets.length-1;i>=0;i--) {  
            double bracket = brackets[i];  
            ...  
        }  
    }  
}
```

- implement the tax calculator **until all tests succeed**
- then build and deploy

# The Rules of TDD

by Robert Martin

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

<http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>

# The Importance of Tests

- test cases are major assets produced in projects - they are **not second class citizens**
- they could be considered more valuable than the actual system - given the tests, a system could be easily re-implemented
- there is an ever-growing repository of tests - these tests re-calibrate our model of the software

# The Importance of Tests (ctd)

- this is used for **regression testing**: uncover bugs caused by changes
- with automated tests, it is **inexpensive to test** - testing can be done often, and can even run in the background
- this facilitates **change**: it is now safe to change because **tests safeguard change**
- this makes testing a core principle of agile methodologies

# Manual Tests

- there could still be some manual user tests, e.g. to assess the ergonomics of user interfaces
- these tests are expensive
- test protocols can be used to make tests **repeatable**
- in these test protocols, preconditions, the test fixture and the expected outcome are described
- manual tests are often used for acceptance testing by clients (validation testing)
- this is when clients decide whether to accept (and pay for) the system