Université Nice SOPHIA ANTIPOLIS

Université Nice SOPHIA ANTIPOLIS

Object Oriented Testing
Junit, Mockito

Philippe Collet

Master 1 IFI International

2012-2013

http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1213

Agenda

- Back to V&V
- JUnit 4
- Differences with JUnit 3
- Alternatives to Junit
- Mockito
- Alternatives to Mockito

V&V Principles (reminder?)

- Two aspects of the concept of quality:
  - Conformance to the specification: VALIDATION
    - Answering: do we build the right product?
    - Checks during realization, most often with the client
    - **Defect** w.r.t to requirements the product must meet
  - Correctness of a stage or of the whole: VERIFICATION
    - Answering: do we build the product correctly?
    - Testing
    - **Errors** w.r.t to precise definitions established during the preceeding developement stages

Test: definition...

- An execution experiment, to emphasize a defect or an error
  - Diagnosis: what is the problem
  - Need of an **oracle**, which indicates whether the experiment result is conformed to the intentions
  - Location (if possible) : where is the cause of the problem?

☞ *Tests should find errors!*

☞ *One should not aim at demonstrating that a program works using tests!*

- Oftent neglected as:
  - Project leaders do not invest in negative results
  - Developers do not consider testing as a destructive process

Elements of a test

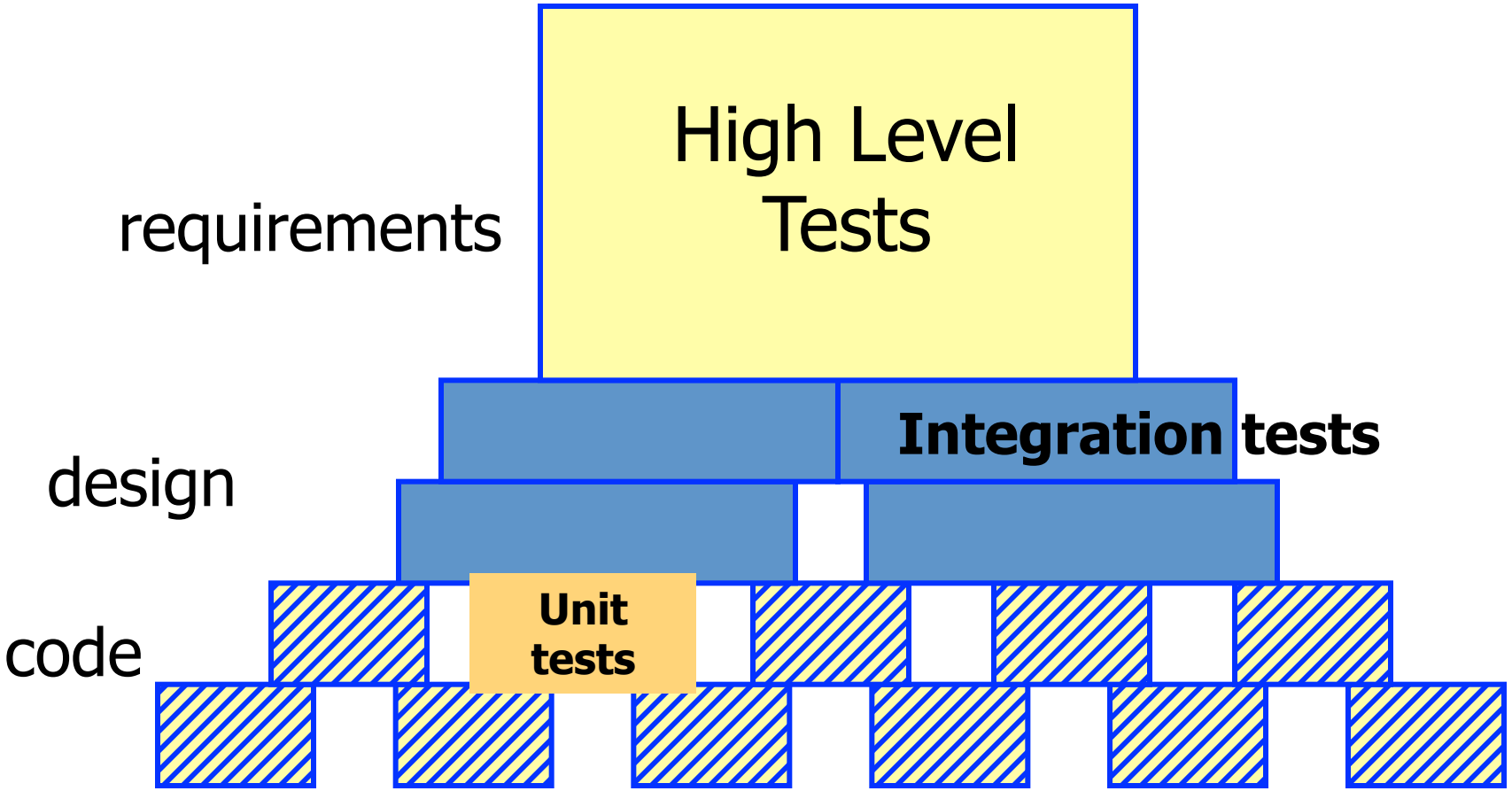
- Name, objective, comments, author
- Data: test set
- Some code that call methods: test case
- Oracles (checking of properties)
- Traces, observable results
- Reporting, a summary…

- Average cost: as much as the tested program

Test vs. Attempt vs. Debugging

- Testing data are kept
  - The cost of testing is recouped
  - Because a test must be **reproductible**
- A test is different from code adjustment or debugging
- Debugging is an investigation
  - Hard to reproduce
  - Which aims at explaining an issue

Testing Strategies



Black box functional testing

- Principles
  - Rely on external specification
  - Partition data to test in **equivalence classes**
    - An expected value in 1..10 gives [1..10], < 1 et > 10
  - Add « relevant » values:
    - Bound testing: on bounds for acceptance, just beyond bounds for denial

Université Nice SOPHIA ANTIPOLIS

JUnit

JUnit v4

www.junit.org

JUnit

- The reference for unit testing in Java
- 3 of the advantages of eXtreme Programming applied to testing:
  - As unit testing use the interface of the class under test, it leads the developer to think of the usage of this interface, early in the development
  - They enable developers to detect early outlier cases
  - **Providing a documented correctness level, they enable developers to modify the code architecture with confidence**

Example

```
class Money {
    private int fAmount;
    private String fCurrency;
    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }
}
```

First test before the implementation of **simpleAdd**

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

public class MoneyTest {
    //…
    @Test public void simpleAdd() {
    Money m12CHF= new Money(12, "CHF");       // (1)
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);         // (2)
    assertTrue(expected.equals(result));      // (3)
}}
```

1. Code to set up the context of the test (*fixture*)
2. Experiment on objects in the context
3. Result checking, oracle…

Test Cases

- Write normal classes
- Define inside some methods annotated with @Test
- To check the expected results (write oracles…) one has to call one of the numerous variants of provided assertXXX() methods:
  - **assertTrue(String message, boolean test), assertFalse(…)**
  - **assertEquals(…) : test with equals**
  - **assertSame(…), assertNotSame(…) : test using object references**
  - **assertNull(…), assertNotNull(…)**
  - **Fail(…) : to directly raise an AssertionFailedError**
  - ***Overloading of some methods for the different base types (int…)***

  - **Add « import static org.junit.Assert.* » to make everything available**

Application to equals in Money

```
@Test public void testEquals() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");

    assertTrue(!m12CHF.equals(null));
    assertEquals(m12CHF, m12CHF);
    assertEquals(m12CHF, new Money(12, "CHF"));
    assertTrue(!m12CHF.equals(m14CHF));
}
```

```
public boolean equals(Object anObject) {
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```

Fixture: common context

- Duplicated setup code:

```
Money m12CHF= new Money(12, "CHF");
Money m14CHF= new Money(14, "CHF");
```

- Classes that have several test methods can use annotations @Before and @After on methods to initializer, resp. clean, the context common to all tests (= *fixture*)
  - Each test is executed in its own context, by calling the @Before method before and the @After method… after (for each test method):
  - For 2 methods, the execution is equivalent to:
    - @Before-method ; @Test1-method(); @After-method();
    - @Before-method ; @Test2-method(); @After-method();
  - This should ensure that no side effect occurs between tests' execution
  - The context is defined by attributed in the testing class

Fixture : application

```
public class MoneyTest {
    private Money f12CHF;
    private Money f14CHF;

@Before public void setUp() {
    f12CHF= new Money(12, "CHF");
    f14CHF= new Money(14, "CHF");
}

@Test public void testEquals() {
    assertTrue(!f12CHF.equals(null));
    assertEquals(f12CHF, f12CHF);
    assertEquals(f12CHF, new Money(12, "CHF"));
    assertTrue(!f12CHF.equals(f14CHF));
}

@Test public void testSimpleAdd() {
    Money expected= new Money(26, "CHF");
    Money result= f12CHF.add(f14CHF);
    assertTrue(expected.equals(result));
}
```

Test Execution

- By introspection of classes
  - Class as method parameter

```
org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);
```

  - Runtime introspection of the class
  - Analyzing annotations @Before, @After, @Test
  - Test execution following the defined semantics (cf. previous slides)
  - Gereration of an object representing the result
    - NOK: detail of the error (Stack Trace, etc.)
    - OK: only counting what's OK
- Further details on the result of a test execution
  - Failure = error of the test (detection of an error in the code under test)
  - Error = error/exception in the testing environment (detection of an error in the testing code)

Test execution with the command line

- Using the class
  - org.junit.runner.JUnitCore

```
java org.junit.runner.JUnitCore com.acme.LoadTester
com.acme.PushTester
```

- Installing JUnit
  - Put junit-4.5.jar in the CLASSPATH (compilation and execution)
  - That's all…

Other features

- Testing exception raising
  - @Test(expected= *ExceptionClass*.class)

```
@Test(expected = ArithmeticException.class)
 public void divideByZero() {
   calculator.divide(0);
 }
```

- Testing an execution with a limited time
  - Spécified in milliseconds

```
@Test(timeout=100)
…
```

- *No equivalent in JUnit 3*

Other features

- Ignore (temporarily) some tests
  - Additional Annotation @Ignore

```
@Ignore("not ready yet")
  @Test
  public void multiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(calculator.getResult(), 100);
  }
}
```

- *No equivalent in JUnit 3*

Test Parametrization

```
@RunWith(value=Parameterized.class)
public class FactorialTest {

    private long expected;
    private int value;

    @Parameters
    public static Collection data() {
        return Arrays.asList( new Object[][] {
                                { 1, 0 },   // expected, value
                                { 1, 1 },
                                { 2, 2 },
                                { 24, 4 },
                                { 5040, 7 },
                                });
    }

    public FactorialTest(long expected, int value) {
        this.expected = expected;
        this.value = value;
    }

    @Test
    public void factorial() {
        Calculator calculator = new Calculator();
        assertEquals(expected, calculator.factorial(value));
    }
}
```

Test Parametrization

- @RunWith(value=Parameterized.class)
  - Executes all tests of the class with data provided in the method annotated with @Parameters
- @Parameters
  - 5 elements in the example list
  - Each element is an array used as arguments for the testing class constructor
  - In the example, data are used in assertEquals
- Equivalent to:

```
factorial#0:  assertEquals( 1, calculator.factorial( 0 ) );
factorial#1:  assertEquals( 1, calculator.factorial( 1 ) );
factorial#2:  assertEquals( 2, calculator.factorial( 2 ) );
factorial#3:  assertEquals( 24, calculator.factorial( 4 ) );
factorial#4:  assertEquals( 5040, calculator.factorial( 7 ) );
```

- *No Equivalent in JUnit 3*

Fixture at the class level

- @BeforeClass
  - A single annotation per class
  - Evaluated once for the testing class, before any other initialization @Before
  - Looks like a constructor…
- @AfterClass
  - A single annotation per class too
  - Evaluated once after all the executed tests, after the last @After
  - Useful to really clean the testing environment (file closing, side effect…)

Suite: test organisation

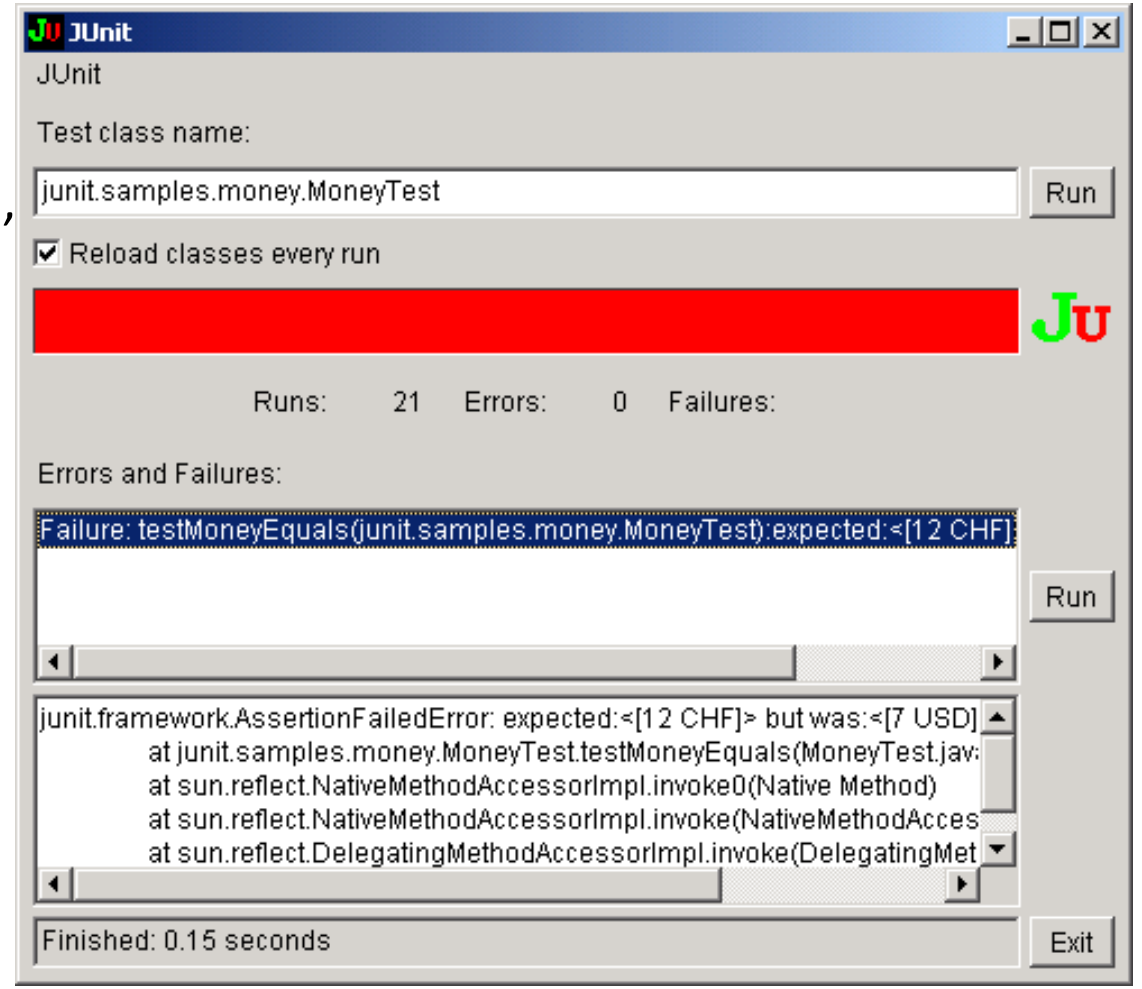
- Some test classes can be organized in hierarchies of « suite »
  - They automatically call all @Test methods in each testing class
  - A « suite » is made of testing classes or suites
  - Tests can be assembled in a hierarchy at any level, all tests are always automatically executed in one pass

```
@RunWith(value=Suite.class)
@SuiteClasses(value={CalculatorTest.class,
AnotherTest.class})
public class AllTests {
   ...
}
```

  - **A suite can have several @BeforeClass and @AfterClass methods, which will be called one before and after the suite execution**

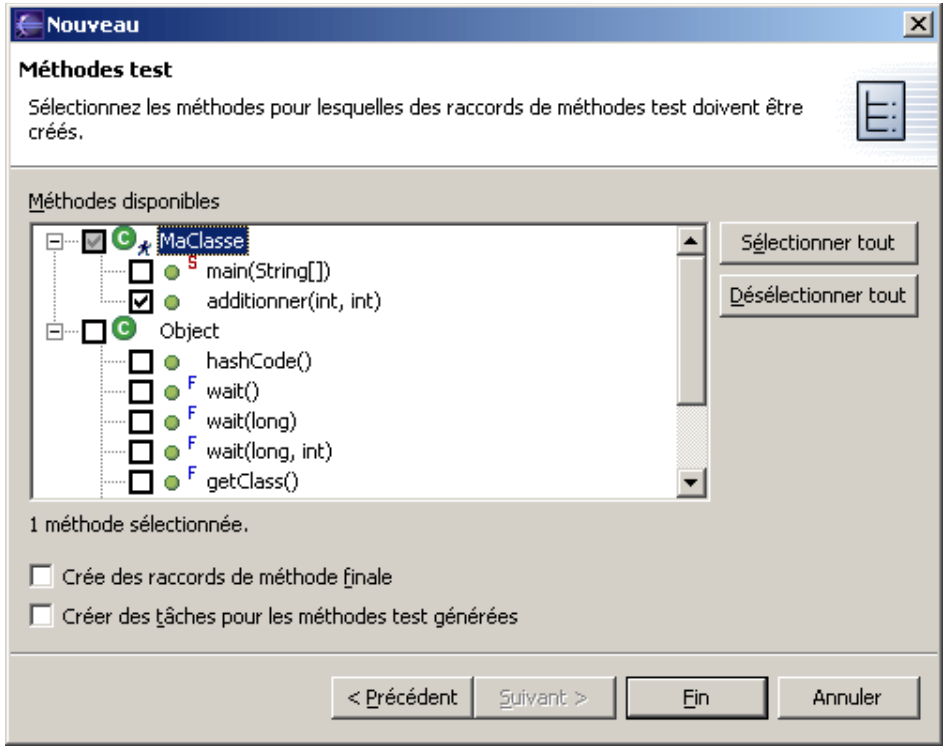
JUnit : TestRunner

- Executes and displays results
- Two versions in JUnit (textual, graphical)
- Integration in IDE

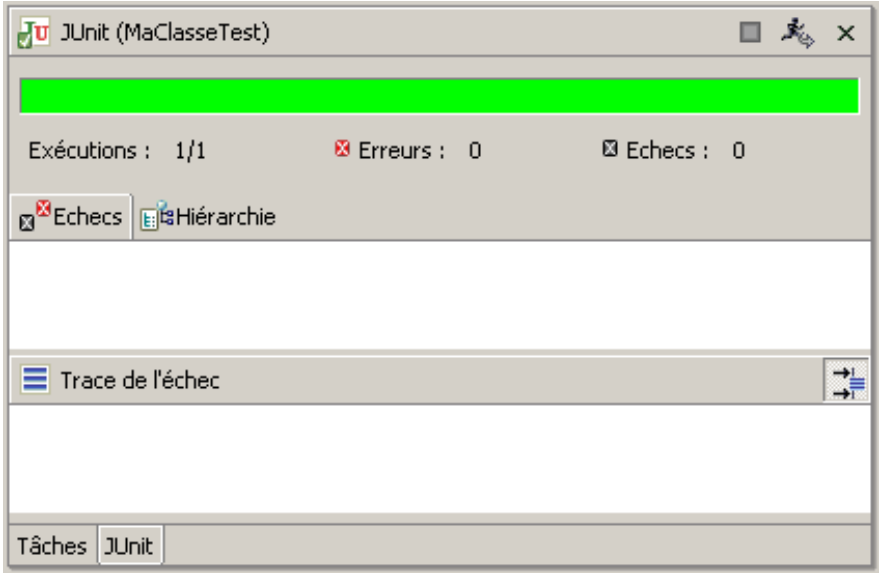


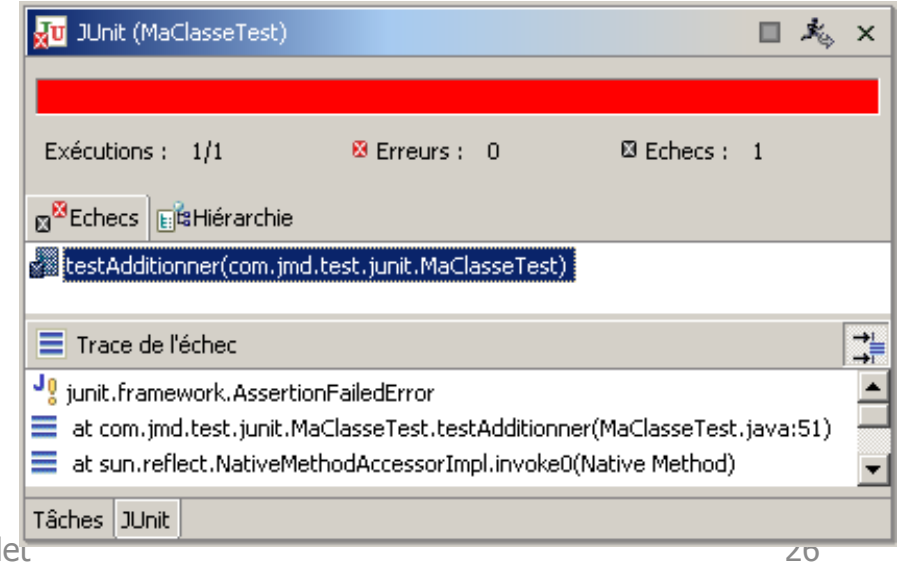
JUnit dans Eclipse

- Wizards for:
  - Creating test cases



- TestRunner integrated in the IDE





What to test? Some principles

- **Principe Right-BICEP**
  - Right : Are the results right?
  - B (Boundary) : Are all the boundary conditions CORRECT ?
  - I (Inverse) : Can you check inverse relationships ?
  - C (Cross-check) : Can you cross-check results using other means?
  - E (Error condition) : Can you force error conditions to happen?
  - P (Performance) : Are performance characteristics within bounds?

Right-BICEP : right

- **Right**
  - Validation of results wrt what's defined in the spec
  - One should be able to answer to « how do we know that the program executed correctly? »
    - If no answer => specifications certainly vague, incomplete
  - Tests = translation of specifications

Right-BICEP : boundary

- **B : Boundary conditions**
  - Identify conditions at the boundaries of the specification
  - What happens if data are
    - Anachronic e.g. : !*W@\/"
    - Not correctly formated e.g. : fred@foobar.
    - Empty or null e.g. : 0, 0.0, "", null
    - Extraordinary e.g. : 10000 for a person age
    - Duplicate e.g. : duplicate in a Set
    - Non compliant e.g. : ordered list that are not
    - Disordered e.g. : print before connect

Right-BICEP : boundary

- **To correctly establish « boundaries »**
- **« CORRECT » principle =**
  - C - Conformance - does the value conform to an expected format?
  - O - Ordering - is the set of values ordered or unordered as appropriate?
  - R - Range - is the value within reasonable minimum and maximum values?
  - R - Reference - does the code reference anything external that isn't under direct control of the code itself?
  - E - Existence - does the value exist (e.g. is not null, non-zero, present in a set)?
  - C - Cardinality - are there exactly enough values?
  - T - Time (absolute and relative) - is everything happening in order? At the right time? In time?

Right-BICEP

- **Inverse – Cross check**
  - Identify
    - Inverse relationships
    - Equivalent algorithms (cross-check)
  - That allow for behavior checking
  - Example: testing square root using the power of 2 function

Right-BICEP

- **Error condition – Performance**
  - Identify what happens if
    - Disk, memory, etc. Are full
    - Network connection is lost
  - E.g., check that an element is not in the list
    - Check that the execution time is linear with the size of the list
  - Watch out! This part is the whole domain of non-functional testing (load, performance…)

Methodological Aspects

- **Coding/testing, coding/testing…**
- **Running tests as often as possible**
  - As often as compiling!
- **Start by writing tests on most critical parts**
  - Write tests that have better ROI!
  - *Extreme Programming* Approach
- **When adding a functionality, write tests first**
  - *Test-Driven Development*…
- **If you ends up debugging with System.out.println(), you'd better write a test instead**
- **When a bug is found, a test that characterizes it must be written**

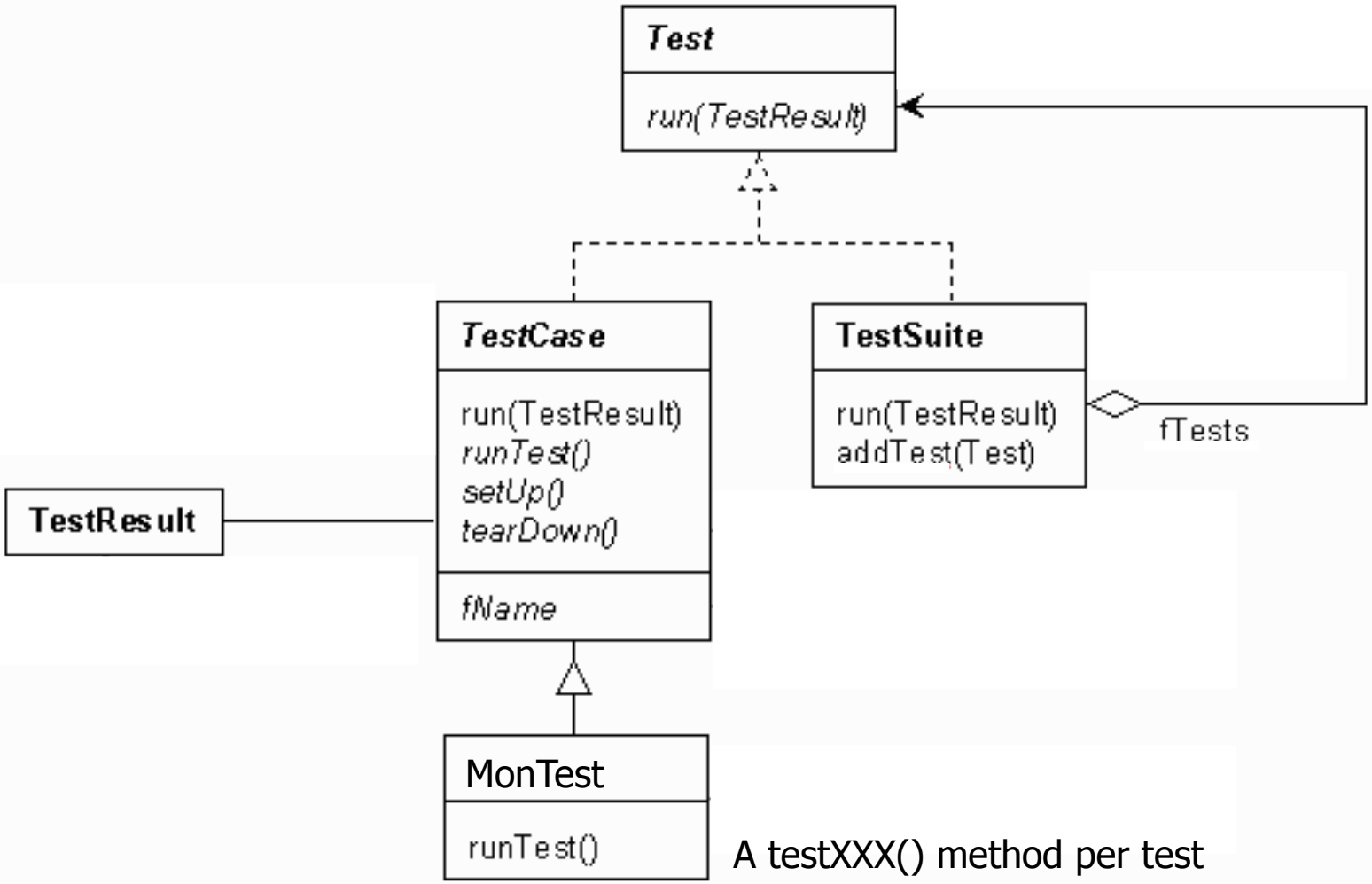
From eXtreme Programming to Test-Driven Development

- **What this is all about?**
  - Deliver only functionalities that the software needs, not those the programmer believes it must provide
  - An obviousness *a priori*
- **How?**
  - Write client code as if the code to develop already existed and was completly to make your life easier!
  - Tests are the client code!
  - Ecrire du code client comme si le code à développer existait déjà et avait été conçu en tout
  - Change the code to compile tests
  - Implement code incrementally
  - Refactor everything to make things always easier for you
  - Write a test (red), write the code (green), refactor
- **+ Principle of continuous integration**
  - During developemnt, the program always works, maybe without some requirements, but what it does, it does it well!

En JUnit 3, les TestCases…

- Write subclasses of TestCase
- A TestCase can define any number of methods testXXX()
- To check expected results (oracle), assertXXX() are provided inside the TestCase
- Setup and teardown methods handle fixture

JUnit 3: class diagram



A testXXX() method per test

JUnit 3.x JUnit 4

```
package junit3;

import calc.Calculator;
import junit.framework.TestCase;



public class CalculatorTest extends
TestCase {

private static Calculator calculator =
            new Calculator();

protected void setUp() {
   calculator.clear();
}

public void testAdd() {
    calculator.add(1);
    calculator.add(1);
    assertEquals(calculator.getResult(),
2);
  }
```

```
package junit4;

import calc.Calculator;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

private static Calculator calculator =
new Calculator();

@Before
public void clearCalculator() {
calculator.clear();
}

@Test
public void add() {
calculator.add(1);
calculator.add(1);
assertEquals(calculator.getResult(), 2);
}
```

JUnit 3.x JUnit 4

```
public void testDivideByZero()  {
    try {
      calculator.divide(0);
      fail();
    } catch (ArithmeticException e) {
    }
  }


 public void notReadyYetTestMultiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(calculator.getResult(),
100);
    }
}
```

```
 @Test(expected =
ArithmeticException.class)
  public void divideByZero() {
    calculator.divide(0);
  }


  @Ignore("not ready yet")
  @Test
  public void multiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(calculator.getResult(),
100);
  }
}
```

Alternative to JUnit

- TestNG (www.testng.org)
  - Annotations Java 5: @Test, @Configuration…
  - Batch execution of test, data injection, distribution on slave machines…
  - Integration in IDE, in tools like Maven
  - Designed to cover several test categories: unit, functional, integration, client/server from end to end

Université Nice SOPHIA ANTIPOLIS

Testing & Mock object

Definition

- Mock = Dummy/artificial object
- Mocks are simulated objects that reproduce the behavior of real objects in a controled way
- One then tests behavior of other, real, objects, but linked to inaccessible or non implemented objects
- This object is replaced by a mock

Definition(s)

- Dummy: empty objects with no functionality implemented
- Stub: classes that returns, hard coded, a value for an invoked method
- Fake: partial implementation that always returns the same results wrt provided parameters
- Spy: class that verifies its own usage after execution
- Mock: class that acts both as a stub and a spy

Example

- Non deterministic behavior (hour, date, sensor)
- Long initialization (DB)
- Class not yet implemented or with an evolving ongoing implementation
- Complex states hard to reproduce inside tests (network error, exceptions on files)
- In order to test, on need to add dedicated attributes or methods

Principle

- A mock has the same interface as the object it simulates
- The client object ignores it interacts with a real object or a simulated one
- Most of mock frameworks allow for
  - Specifying which methods are going to be called, with which parameters and in which order
  - Specifying values returned by the mock

Université Nice SOPHIA ANTIPOLIS

Mockito

http://code.google.com/p/mockito/

With elements from a lecture of M. Nebut lifl.fr

Mockito

- Automatic generator of mock objects
- Lightweight
  - Focus on expected behavior and on checking after execution
- Simple
  - A single type of mock
  - A single way to create them

Principles

- The framework works in spy mode:
  - Mocks creation
    - *mock* method or @mock annotation
  - Behavior description
    - *When* method
  - Runtime memorization of interactions
    - Use of the mock in code testing a specific behavior
  - Queries, at the end of the test, on mocks to determine how thay have been used
    - *Verify* method

import static org.mockito.Mockito.*

Creation

- Through an interface or a class (using .class)
  - AnInterface mockUnamed = mock(AnInterface.class);
  - AnInterface mockNamed =
    mock(AnInterface.class, "thisMock");
  - @Mock AnInterface thisMock;
- Default Behavior
  - assertEquals("thisMock", thisMock.toString());
  - assertEquals("numeric type: 0 ", 0,
    myMock.fctReturningAnInt());
  - assertEquals("boolean type: false",
    false, myMock.fctReturningABoolean());

Stubbing

- To replace the default behavior of methods
- Two possibilities
  - Method with a return type
    - when + thenReturn ;
    - when + thenThrow ;
  - Method of type void :
    - doThrow + when ;

Stubbing
returning an unique value

```
// stubbing
when(myMock.fctReturningAnInt()).thenReturn(3);

// description with JUnit
assertEquals("a first time 3", 3, myMock.fctReturningAnInt());
assertEquals("a second time 3", 3, myMock.fctReturningAnInt());
```

Stubbing
consecutive return values

```
// stubbing
when(myMock.fctReturningAnInt()).thenReturn(3, 4, 5);

// description with JUnit
assertEquals("a first time 3", 3, myMock.fctReturningAnInt());
assertEquals("a second time 4", 4, myMock.fctReturningAnInt());
assertEquals("a second time 5", 5, myMock.fctReturningAnInt());



when(myMock.fctReturningAnInt()).thenReturn(3, 4);
// shortcut for    .thenReturn(3).thenReturn(4);
```

Stubbing
Raising exceptions

```
public int returnAnIntorRaiseAnExc() throws WhateverException;
// stubbing
when(myMock.returnAnIntorRaiseAnExc()).thenReturn(3)
                    .thenThrow(new WhateverException());

// description with JUnit
assertEquals("1st call: returns 3",
                3, myMock.returnAnIntorRaiseAnExc());

try {
     myMock.returnAnIntorRaiseAnExc(); fail();
} catch (WhateverException e) {
     assertTrue("2nd call: exception", true);
}
```

Exception raising + void method = doThrow

Some remarks

- Methods equals() and hashcode() cannot be *stubbed*
- A mock behavior not executed does not lead to an error
- One must use *verify*
  - Which methods have been called on a mock
  - How many times, with which parameters, in which order
- An exception is raised if checking fails, the test will fail as well

Verify

- Method called only once:
  - verify(myMock).somefunction();
  - verify(myMock, times(1)). somefunction();
- Method calles at least/at most once:
  - verify(myMock, atLeastOnce()). somefunction();
  - verify(myMock, atMost(1)) somefunction();
- Method never called:
  - verify(myMock, never()). somefunction();
- With specific parameters:
  - verify(myMock). someotherfunction(4, 2);

Verify

- import org.mockito.InOrder;
- To check that the call (4,2) is done before the call (5,3):
  - InOrder ord = inOrder(myMock);
  - ord.verify(myMock).somefunction(4, 2);
  - ord.verify(myMock).somefunction(5, 3);
- With several mocks :
  - InOrder ord = inOrder(mock1, mock2);
  - ord.verify(mock1).foo();
  - ord.verify(mock2).bar();

Alternatives to Mockito

- EasyMock, Jmock
- All based on expect-run-verify