Métodos Formais em Engenharia de Software

Unit-testing with JML

José Carlos Bacelar Almeida

Departamento de Informática
Universidade do Minho

MI/MEI 2008/2009

Talk Outline

- Unit Testing
  - software testing
  - JUnit framework
  - testing coverage
- JML-Unit
  - basic usage
  - tool support

Software Testing

- Goal: detect software failures so that defects may be corrected
- Problem: it is often impossible to test software systems under all possible inputs and admissible states

  "Testing is able to signal the presence of faults, but can't demonstrate their absence"

- Usually performed on different levels:
  - unit testing
  - functional testing
  - integration testing
  - system testing, acceptance testing, ...
- Important to allow an early fault-detection.

		Time Detected				
		Requirements	**Architecture**	**Construction**	**System Test**	**Post-Release**
Time Introduced	**Requirements**	1×	3×	5–10×	10×	10–100×
	Architecture	-	1×	10×	15×	25–100×
	Construction	-	-	1×	10×	10–25×

Unit Testing

- Tests the minimal software component, or module (e.g. a Java class).
- Isolate each part of the program and show that the individual parts are correct.
- Unit test cases embody characteristics that are critical to the success of the unit --- to some extent, they act as specifications of appropriate/inappropriate uses.
- Typically done by software developers to ensure that the code meets software requirements and behaves as intended.
- Good unit test design produces test cases that cover all paths through the unit with attention paid to loop conditions.
- It relies on a sustainable process for ensuring that test case failures are reviewed daily and addressed immediately.
- Dependencies with other parts of the system (e.g. databases) is abstracted by mock-objects.

JUnit framework

- Java framework that supports unit-testing.
- Some terminology:
  - Test Case - set of conditions/variables/invocations that exercises the target code
  - Assertions - act of comparing the outcome of tests with expected results
  - Fixture - appropriate environment for running the test cases
  - Test Suite - collection of test cases
  - Test Runner - program that runs the tests
- Availability:
  - http://www.junit.org/
  - current version: 4.5

  obs.: version 4.X uses Java annotations (a Java5 feature...). Thus, when using JUnit and JML, it is preferable to use version 3.8.2.

JUnit usage

- JUnit is Java framework to assist programmers in writing unit-tests for their code
- Basic usage steps:
  - write a TestCase
  - write a TestSuite
  - run the tests
- Benefits of unit testing greatly depend on:
  - programmers commitment in writing quality test cases
  - organisational procedures for evaluate and monitorise tests (build system; code/tests organization; ...)
- Recommended reading:
  - JUnit Primer - (http://clarkware.com/articles/JUnitPrimer.html)
  - JUnit - A Cook's Tour (http://junit.sourceforge.net/doc/cookstour/cookstour.htm)

TestCase classes

- Collects the tests for the intended "code unit"
- Extends JUnit TestCase abstract class
- Each test is a public method with name "testXXX()"
- Outcome of tests is checked against expected values (assert methods)
  - assertTrue/False, assert(Not)Equals, assert(Not)Same
  - assertArray(Not)Equals
  - fail
- (Test Fixture) initialisation of common objects under test is performed by overriding method "setUp()" ("tearDown()" to release them).
- (optional) static method "suite()" defines the default TestSuite for the test methods.

TestCase example

```
import junit.framework.TestCase;

public class ShoppingCartTest extends TestCase {
    private ShoppingCart cart;
    private Product book1;

    protected void setUp() {
        cart = new ShoppingCart();
        book1 = new Product("Pragmatic Unit Testing", 29.95);
        cart.addItem(book1);
    }

    /**
     * Tests emptying the cart.
     */
    public void testEmpty() {
        cart.empty();
        assertEquals(0, cart.getItemCount());
    }

    ...

    // collects test methods from this class...
    public static Test suite() {
        TestSuite suite= new TestSuite(MoneyTest.class);
        return suite;
    }
}
```

TestSuite classes

- TestSuite is the smallest execution unit in JUnit
- Is a composite of other tests (either TestCases or TestSuites)
- Adopt often an hierarchical structure (mirroring the package structure)

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class EcommerceTestSuite {
    public static Test suite() {
        TestSuite suite = new TestSuite();

        suite.addTestSuite(ShoppingCartTest.class);

        suite.addTest(CreditCardTestSuite.suite());

        return suite;
    }

    /**
     * Runs the test suite using the textual runner.
     */
    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```


Code Coverage

- One of the most widely used approach to measure software testing quality is the adoption of some "coverage measure"
- Different possible measurements:
  - function coverage (functions tested)
  - line coverage (lines of code exercised by tests)
  - branch coverage (coverage of decision points)
  - path coverage (control flow paths)
  - entry/exit coverage, ...
- Like any software metric, results should be taken with caution:
  - can be blind to "that obvious problem"
  - often very easy to trick
- Several (free) tools available (e.g. Emma)

JML-Unit

- A Behaviour Interface Specification Language for Java (Gary T. Leavens et al. [BCC+05])
- It permits to:
  - specify behaviour of Java classes
  - record design & implementation decisions
- ...by adding assertions to Java source code
- JML syntax is well integrated with Java:
  - JML assertions are added as comments in .java files, between /*@ ... @*/, or after //@ ;
  - Properties are specified as Java boolean expressions, extended with some operators (\old, \forall, \result, ...),
  - ...and some keywords (requires, ensures, signals, assignable, pure, invariant, non_null, ...).

JML-Unit usage

- Pre and postconditions for methods are established through the "requires" and "ensures" clauses:

```
/*@ requires amount >= 0;
  @ ensures balance == \old(balance)-amount;
  @ ensures \result == balance;
  @*/
public int debit(int amount) {
...
}
```

- where
  - \old(balance) refers to the value of balance before the execution of the method;
  - the multiple ensures clauses are equivalent to their conjunction;
  - \result refers to the outcome of the method (return value).

JML-Unit (data generators)

- JML properties are boolean Java expressions...
- ...with the proviso that their evaluation is "side-effect free" (i.e. does not change the internal state).
- A method without side-effects is called pure. Programmers might signal methods as pure:

```
public /*@ pure @*/ int getBalance(){...}

Directory /*@ pure non_null @*/ getParent(){...}
```

- The non_null clause signals that the result of getParent() can't be null (can also be used in arguments and instance variables).
- JML property language is extended with binding operators: \forall, \exists, \sum, \product, \max, \min, ...

E.g. (\forall int i ; 0<=i && i<N ; a[i]==null)

JML-Unit

- Unit tests are built around
  - input data
  - code execution
  - result check
- JML runtime assertion check is clearly interesting for checking successful test results: "a method call is successful whenever its post-condition is valid (+class invariant)" (if input data validates precondition...)
- JmlUnit adds to the equation "JUnit+JML" some ingredients:
  - systematically generates (JUnit) TestCases that exercise all public class methods;
  - preconditions are used to filter out irrelevant method calls
  - post-conditions/invariant violations capture test failures
- ...the user only needs to provide "interesting input data"

JMLUnit usage

- 1st step: setup CLASSPATH
  - include "jmlruntime.jar, jmljunitruntime.jar, junit.jar"
- 2nd step: run jmlunit on the target jml-annotated java file

```
$ jmlunit MyJmlClass.java
generates "MyJmlClass_JML_Test.java" and "MyJmlClass_JML_TestData.java"
```

- 3rd step: add test-values to "MyJmlClass_JML_TestData.java"
- 4th step: java-compile test case

```
$ javac MyJmlClass_JML_*.java
```

- 5th step: jml-compile MyJmlClass.java

```
$ jmlc MyJmlClass.java
```

- 6th step: run the tests

```
$ jml-junit MyJmlClass_JML_TestCase
```


Test Data

- Edit the "MyJmlClass_JML_TestData.java" file:
  - Clonable objects (Strings, ints, ...)

```
private org.jmlspecs.jmlunit.strategies.StrategyType
    vjava_lang_StringStrategy
    = new org.jmlspecs.jmlunit.strategies.StringStrategy()
        {protected java.lang.Object[] addData() {
              return new java.lang.String[] {
                  // replace this comment with test data if desired
                   "Jose", "Maria", "Manuel"
                  };
              }
         };
```

- Non-Clonable objects

```
    private org.jmlspecs.jmlunit.strategies.StrategyType
        vUserStrategy
        = new org.jmlspecs.jmlunit.strategies.NewObjectAbstractStrategy()
            {
                protected Object make(int n) {
                    switch (n) {
                      // replace this comment with test data if desired
                      case 0: return new User("user1","pass1");
                      case 1: return new ...
//                    ...
                    }
```