Lecture 13
Testing, JUnit and Refacoring

Lecturer: Tommy Yuan

Fall 2008 - Year 2

Object-Oriented Methods

Outline

- Testing
- Junit Testing
- Refactoring

2

UML as Blueprint

- Forward engineering
  - Make it possible to generate code (in Java, C++ and VB) from the models
  - Many fine details of the code are usually filled in by developers while programming
  - Ideally complete executable specification
- Reverse engineering
  - Generation of diagrams from code
- Round-trip engineering
  - Close the loop, synchronise between UML diagram and code, ideally automatically and immediately as either is changed

3

Who tests?

- Ideally independent specialist test teams
- Often the analysts who have carried out the initial requirements gathering and analysis
- In eXtreme Programming (XP) programmers are expected to write test harnesses for classes before they write the code
- Users of the system, who will test against requirements and do user acceptance testing

4

What is tested?

- An approach to testing where the program is considered as a 'black-box'
- The program test cases are based on the system specification
  - Functional requirements
    - Does it do what it's meant to do?
  - Non-functional requirements
    - E.g. Does it do it as fast as it should?
- Test planning can begin early in the software process
- Testers only access the software through the same interfaces that the customer or user would

5

What is tested?

- White box testing or glass box testing
- Objectives are to test the internal of the software, and whether it works as specified.
- Testers need to access the code and then derive test cases according to program structure. Test code links into the libraries and the target software.
- This is typical of *unit tests*, which only test parts of a software system. They ensure that components used in the construction are functional and robust to some degree.
- Knows where goes wrong?

6

Test Plans

- Written before the tests are carried out!
- Written, in fact, before the code is written
- Contain Test Cases
  - test description
  - test data
  - expected outcomes
  - ....

7

Test Plans

Test no. 23			
Purpose: Test correct addition of campaign and adverts			
Step no.	Test description	Test data	Expected result
23.1	Create a new Campaign		Campaign added to database. Campaign estimated Cost is set to $0.00
23.2	Add advert 1 to Campaign	Advert estimated cost=$500	Advert added to the database. Campaign estimated cost is set to $500
23.3	.....		
....	.....		

8

Testing Data

- Testing programs to establish the presence of system defects - try to **break down** the system
- Test data should test the software at its limits and test business rules
  - extreme values (very large numbers, long strings)
  - out of range or close to range limit (0, -1, 0.999)
  - invalid combinations of values (age = 3, marital status = married)
  - nonsensical values (negative values)
  - heavy loads (are performance requirements met?)

9

Levels of Testing

- Bottom up
  - Unit testing (individual classes)
  - Integration testing (classes work correctly together)
  - Subsystem testing (subsystem works correctly and delivers required functionality)
  - System testing (whole system works together with no unwanted interaction between subsystems)
  - Acceptance testing (the system works as required by the users and according to specification)

10

Stages of Testing

- Stage 1
  - Test modules (classes), then programs (use cases) then suites (application)
- Stage 2 (Alpha Testing or Verification)
  - Execute programs in a simulated environment and test inputs and outputs
  - Before release
- Stage 3 (Beta Testing or Validation)
  - Test in a live user environment and test for response times, performance under load and recovery from failure
  - Release to samll group
- Stage 4 Gamma
  - Released to public

11

JUnit Testing

not CppUnit for C++

12

Unit Tests

- First level of testing
- Test individual code segments
- Done by the programmer
- Part of the coding process
- Delivered with the code
- Part of the build process

13

Why Unit Testing?

- Allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e. regression testing).
- Simplifies integration: helps eliminate uncertainty in the pieces themselves and can be used in a bottom-up testing style approach.
- Without unit tests
  - Code integration is a nightmare
  - Changing code is a nightmare

14

When is JUnit appropriate?

- As the name implies…
  - for unit testing of small amounts of code
- It is not intended for complex testing, system testing, etc.
- Typically advocated in the eXtreme Programming methodology,
  - A JUnit test should be written first (before any code), and executed.
  - Once the code is written, re-execute the test and it should pass.

15

What is a JUnit Test?

- Unit tests are written in test classes, that replace 'drivers'
- For example, class Stack
  - Class *Stack* has *push, pop, count, …*
  - Class *TestStack* has *testPush, testPop*
- What is added? Assertions.
  - A package of methods that checks various properties:
    - equality of variables
    - identity of objects
  - The assertions are used to determine the test case verdict.

16

Example class Stack

```
public class Stack  {
   int[] elements;
   int topElement = -1;

   public Stack() {
    this(10);
   }

   public Stack(int size) {
      elements = new int[size];
   }

   public boolean isEmpty() {
      return topElement == -1;
   }
}
```

Note: written after test class

17

Junit Example

Goal: Implement a Stack containing integers.

Tests:

Subclass junit.framework.TestCase

Methods starting with 'test" are run by TestRunner

First tests for the constructors:

```
public class  TestStack extends TestCase {

   public void testDefaultConstructor() {
      Stack test = new Stack();
      assertTrue("Default constructor", test.isEmpty() );
   }

   public void testSizeConstructor() {
      Stack test = new Stack(5);
      assertTrue( test.isEmpty() );
   }
}
```

A JUnit class

A test

Another test

18

Assertions

- Assertions are defined in the special JUnit class Assert
  - If the assertions are true, the method continues executing.
  - If any assertion is false, the method stops executing, and the result for the test case will be "fail".
  - If any other exception is thrown during the method, the result for the test case will be "error".
  - If no assertions were violated for the entire method, the test case will pass.

19

Assert class

```
assertTrue()
assertFalse()
assertEquals()
assertNotEquals()
assertSame()
assertNotSame()
assertNull()
assertNotNull()
fail()
```

For a complete list of the assert methods & arguments see

http://junit.sourceforge.net/javadoc/

20

Running JUnit Tests

- JUnit Integrated into Eclipse
  - Open Source
  - Graphical user interface
  - Easily choose which tests to run
  - Elegantly support for test suites

21

Test Fixtures

Before each test setUp() is run

```
import junit.framework.TestCase;

public class  StackTest extends TestCase {
   Stack test;

   public void setUp() {
      test = new Stack(5);
      for (int k = 1; k <=5;k++)
         test.push( k);
   }

   public void testPushPop() {

      for (int k = 5; k >= 1; k--)
         assertEquals( "Pop fail on element " + k,  test.pop() , k);
   }
}
```

Testing set up

22

Multiple Test Classes

One more example

```
import junit.framework.TestCase;

public class TestQueue extends TestCase{

   public void testConstructor() {
      Queue test = new Queue();
      assertTrue( test.isEmpty());
   }
}
```

```
import java.util.Vector;

public class  Queue{
   Vector elements = new Vector();
   public boolean isEmpty() {
      return elements.isEmpty();
   }
}
```

Multiple test classes can be run at the same time

Add Queue & TestQueue to Stack classes

23

Using a Suite to Run Multiple Test Classes

Running AllTests runs the tests in

StackTest
QueueTest

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

public class AllTests  {
   static public void main(String[] args) {
      TestRunner.run( example.AllTests.suite());
   }

   public static Test suite(){
      TestSuite suite = new TestSuite("Test for example");
      suite.addTestSuite(StackTest.class);
      suite.addTestSuite(QueueTest.class);
      return suite;
   }
}
```

May not needed

Eclipse will generate the above class for you. In the file menu select "New", then in the submenu select "other". In the dialog window select "JUnit Test Suite" which is under Java and then under JUnit.

24

Running Junit without Eclipse

--You can easily run them using Eclipse
--You can also download Junit from http://www.junit.org/ and set in your class path, and run it using command line

```
C:\WINDOWS\system32\cmd.exe
OK (0 tests)

H:\Teaching\OO\Junit-Foo>java org.junit.runner.JUnitCore TestFoo
JUnit version 4.5

Time: 0.031
```

```
H:\Teaching\OO\Junit-Foo>java org.junit.runner.JUnitCore AllTests
JUnit version 4.5

Time: 0.032

OK (1 test)

H:\Teaching\OO\Junit-Foo>_
```

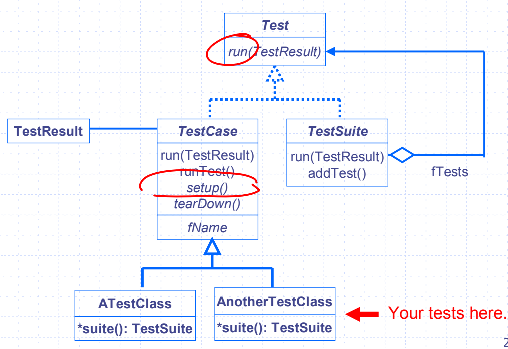
25

Test vs TestCase vs TestSuite?

- "TestCase" is a class
- A single test is a method.
- A "test suite" is then a collection of (possibly related) tests that are run as a group.
- No difference between running a test suite and a test case, but provide for hierarchical test suites.

26

The JUnit Framework



27

Refactoring

28

Refactoring

- Refactoring is the process of improving the design of existing code without changing its observable behaviour.
- Essense - behaviour preserving transformation - each called a 'refactoring'
- Supported by unit tests.
- Martin Fowler's "Refactoring – improving the design of existing code" -a book that has a catalogue of 80 refactorings
- Dedicated website- www.refactoring.com

29

Refactoring

- Replacing code that 'smells'
  - duplicated code
  - big method
  - class with many instance variables
  - class with lots of code
  - high coupling between many objects
  - long parameter lists
  - and so many other ways bad code is written…..

30

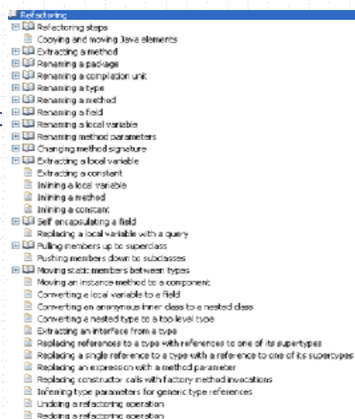
Refactoring Methods

- There are about 100 named refactoring
  - Extract method: Transform a long method into a shorter one by factoring out a portion into a private helper method.
  - Extract a constant: replace a literal constant with a constant variable
  - Extract local variables: introduce explaining temaporary variable
  - ....

31

Refactoring

- IDE Support - Eclipse



32

Resources

- Text: Ch section 19.5.
- Top 12 Reasons to Write Unit Tests
  - http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html
- JUnit
  - http://www.junit.org/
- Refactor
  - http://www.refactoring.com/

33