Unit Testing
& JUnit
Lecture Outline

- **Communicating your Classes**
- Introduction to JUnit4
- Selecting test cases
UML Class Diagrams

- Give an external view of a class
- But there are lots of questions they don’t answer
- If you want to use this class, you will need more information
- For example, what does the resize method do? What do its parameters mean?

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>height : int</td>
</tr>
<tr>
<td>width : int</td>
</tr>
<tr>
<td>resize(double,double)</td>
</tr>
<tr>
<td>getArea(): int</td>
</tr>
<tr>
<td>getPerimeter(): int</td>
</tr>
</tbody>
</table>
More information

- Method signature
  
  ```java
  public void resize( double hscale, double wscale)
  ```

- JavaDoc
  
  ```java
  /**
   * Grow or shrink a rectangle in two dimensions.
   *
   * @param hscale double multiplier for height
   * @param wscale double multiplier for width
   */
  ```
(Even) More information

- More JavaDoc

/**
* Grow or shrink a rectangle in both dimensions
* scaling dimensions are floats (can be less than 1)
* but final dimensions are recast as integers.
*
* @param hscale the float multiplier for height.
* @param wscale the float multiplier for width.
* @throws exceptions for what cases ??
*/
## Examples of Method Behaviour

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>height=3, width=4</td>
<td>height=6, width=2</td>
</tr>
<tr>
<td>hscale=2.0, wscale=0.5</td>
<td></td>
</tr>
<tr>
<td>height=3, width=4</td>
<td>height=3, width=4</td>
</tr>
<tr>
<td>hscale=1.0, wscale=1.0</td>
<td></td>
</tr>
<tr>
<td>height=3, width=4</td>
<td>height=1??, width=0 ??</td>
</tr>
<tr>
<td>hscale=0.5, wscale=0.0</td>
<td></td>
</tr>
</tbody>
</table>
How much info do you need?

- Method signatures don’t tell you *what* the method does
- JavaDoc *can* specify anything you need to know, but we don’t have a way of checking that the code meets the specification
- **Examples** are helpful for programmers
- Most importantly, we can *check* whether code behaves as the examples claim
- When such checks are themselves Java code, then we can *recheck* that code still works after any change, by *running* the test code
Lecture Outline

- Communicating Java methods
- Introduction to JUnit4
- Selecting test cases
JUnit

- JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is used by the developer who implements unit tests in Java.
- JUnit is Open Source Software, released under the Common Public License Version 1.0 and hosted on SourceForge.
- There is an excellent Eclipse plug-in to support the latest JUnit version 4.1
JUnit

- de facto standard unit testing framework
- bundled in most IDEs
- http://www.junit.org/
- useful to have one complete test suite the programmers can run with a pass fail result
- automatic quality assurance: when the code changes, test cases are rerun to check that nothing breaks
Automated Testing

- Tests must run automatically without any human participation beyond launching the test suite (and maybe not even that)
- Results should be pass/fail
- Test failures need to be blindingly obvious
- Test must be reproducible and consistent
What’s in a test case?

Unit test cases have the following form:

1. describe the method’s inputs
2. describe the expected outputs
3. call the method and observe actual output
4. compare expected with actual and report
Coding a test case in JUnit4

@Test
public void testGetAreaNormal() {  
  //1. describe input
  Rectangle r1 = new Rectangle(3,4);
  //2. describe expected output(s)
  int expected = 12;
  //3. call the method
  int actual = r1.getArea();
  // 4. compare actual output with the expected output
  assertEquals("normal getArea",expected,actual);
}
Before and After every test case

@Before
public void setUp() throws Exception {
    //TODO Create objects and common variables
    //for your tests here
    //setUp is called before EACH test case is run
}

@After
public void tearDown() throws Exception {
    //TODO put code here to reset or
    //release objects
    //eg. p1=null;
    //to garbage collect old MobilePhones
    //tearDown is called after EACH test case is run
}
How a test suite is executed

1. @Before code
2. @Test case 1
3. @After code
4. @Before code
5. @Test case 2
6. @After code

etc. until all test cases are executed

IMPORTANT each test is independent of the previous ones
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class BankAccountTest {

    BankAccount b0;

    @Before
    public void setUp() throws Exception {
        b0 = new BankAccount("rachel", 0); //create an empty BA
    }

    @After
    public void tearDown() throws Exception {
        b0 = null; //garbage collection for old BankAccount
    }

    @Test
    public void testDepositNormal() {
        b0.deposit(100);
        assertEquals(b0.getBalance(), 100);
    }

    @Test
    public void testDepositSpecial() {
        b0.deposit(0);
        assertEquals(b0.getBalance(), 0);
    }

    @Test
    public void testDepositIllegal() {
        //TODO test illegal deposit amounts
        fail("Not yet implemented");
    }
}

Does expected = actual?

To compare any variables of the 8 Java base types (boolean, byte, short, int, long, char, float, double) use

- `assertEquals(expected, actual)`

First argument (optional) is a string that's printed if the assertion fails:

- `assertEquals(message, expected, actual)`
Straight-forward comparisons for basic types

public static void assertEquals (int expected, int actual)

assertEquals(boolean expected, boolean actual)

assertEquals(byte expected, byte actual)

assertEquals(char expected, char actual)

assertEquals(short expected, short actual)

assertEquals(long expected, long actual)
Floating point arithmetic

- As always with floating point arithmetic (double or float), you want to avoid direct equality comparisons.

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

- For example,
  assertEquals(0.552, 0.510, 0.1) is true but
  assertEquals(0.552, 0.510, 0.01) is false

  since the values are equivalent to within a tolerance of
  plus or minus 0.1 but not to 2 DP
How to Test Objects

- What is the expected result of this test?

```java
@Test
public void testObjEquality() {
    Rectangle r1 = new Rectangle(2, 4);
    Rectangle r2 = new Rectangle(2, 4);
    assertEquals(r1, r2);
}
```
Testing Objects Problem

JUnit responds with

junit.framework.AssertionFailedError: resize OK
expected:<Rectangle@913fe2> but
was:<Rectangle@1f934ad>

Problem: we have tested for equivalence of the
references to the 2 rectangle objects, not the
contents of those rectangle objects
Solution

Write a local equivalence tester for the class you are testing

private void testRectangleEquality( Rectangle r1, Rectangle r2) {
    assertEquals("Not same height", r1.getHeight(), r2.getHeight());
    assertEquals("Not same width", r1.getWidth(), r2.getWidth());
}

Aside: This test method exits after the first failure, so if both tests fail only the first is reported
More assertions

- `assertFalse(condition)`
- `assertFalse(message, condition)`
- `assertNotNull(object)`
- `assertNotNull(message, object)`
- `assertNotSame(expected, actual)`
- `assertNotSame(message, expected, actual)`
- `assertNull(object)`
- `assertNull(message, object)`
- `assertSame(expected, actual)`
- `assertSame(message, expected, actual)`
- `assertTrue(condition)`
- `assertTrue(message, condition)`
- `fail()`
- `fail(message)`
- `failNotEquals(message, expected, actual)`
- `failNotSame(message, expected, actual)`
- `failSame(message)`
JUnit4 in Eclipse (Demo)

- Start Eclipse and check that JUnit is on the project build path Java Settings > Libraries
- Select the class to be tested in the Navigator window e.g. BankAccount.java
- Right click and select New->JUnit Test Case
- An outline class BankAccountTest.java will be generated automatically for you
- Just fill in the values to test and go!
Lecture Outline

- Communicating Java methods
- Introduction to JUnit
- Selecting test cases
Choosing Test Cases

- Typical cases
  - Sanity check that method code works as expected, but try to choose tricky cases anyway!

- Special cases and Boundary cases
  - Most errors occur for boundary values, e.g., empty or full array, -1, 0, 1 etc.

- Exceptional cases
  - Illegal input, divide by 0, un-initialized parameters
Tables are a good way of planning your test cases

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPICAL</td>
<td></td>
</tr>
<tr>
<td>height=3, width=4</td>
<td>height=6, width=2</td>
</tr>
<tr>
<td>hscale=2.0, wscale=0.5</td>
<td></td>
</tr>
<tr>
<td>BOUNDARY</td>
<td></td>
</tr>
<tr>
<td>height=3, width=4</td>
<td>height=3, width=4</td>
</tr>
<tr>
<td>hscale=1.0, wscale=1.0</td>
<td></td>
</tr>
<tr>
<td>EXCEPTIONAL</td>
<td></td>
</tr>
<tr>
<td>height=3, width=4</td>
<td>height=1??, width=0 ??</td>
</tr>
<tr>
<td>hscale=0.5, wscale=0.0</td>
<td></td>
</tr>
</tbody>
</table>
Also, ask questions

- Works well when testing someone else’s class eg. a library class
- Ask: What happens if you do XXX?
  - Is an exception thrown?
  - If so which one?
  - Is null treated as zero?
  - Is null returned?
  - Something else?
- Write a test that determines the answer to your question(s)
Future Work – beyond JUnit

- **Test Log Files**
  - Goal: Generate a human-readable text file, detailing the tests run and their results
  - Implementation: add (guarded) print statements to your JUnit Test file

- **Acceptance Tests**
  - Use a main method that executes a sequence of steps (method calls) corresponding to client use cases for the system

- **Coverage Testing**
  - Check that your test cases execute all branches of the code

- **Refs:** JNuke Virginia test environment etc.