INB370 / INN370
Software Development
Lecture 3 — Unit Testing

Faculty of Science and Technology

Semester 1, 2010

Aims of the Week 3 lecture and practical session

- To briefly examine the role of *assertions* as a 'defensive programming' technique
  - And how assertions are expressed in Java programs
- To describe the role of *unit testing* in large-scale software development
  - And how unit testing can be performed with JUnit and Eclipse

Part A — Assertions

Assertions

- *Assertions* are used to document facts that we expect to be true at certain points in a program:
  - They define expected *input values* (especially method parameters) in a way more precise than just their types
  - They document intended *loop invariants*
  - They describe expected initial and final *states* of methods
- Assertions *do nothing*, as long as the program is well-behaved
  - But they produce a *fatal* error otherwise
- Assertions can be used to assist program *debugging*
  - They can be used instead of laboriously inserting and removing print statements
  - They allow errors to be trapped at their source, rather than propagating to some other part of the program
- They can be the basis for formal program *proofs*

Assertions in Java

- Two forms:
  - **`assert`** *boolean_expression*`;`
  - **`assert`** *boolean_expression* `:` *displayable_expression*`;`
- In the second form the displayable expression, usually a string, is printed if the assertion is violated
- When the boolean expression is false, an `AssertionError` is generated and the program is terminated
  - This is an `Error` rather than an `Exception` to prevent programmers from trying to recover from assertions
  - A failed assertion is a *fatal* programming error!
- In Java assertions are disabled by default (… is this a good idea?)
  - Enable them in Eclipse by adding a "`-ea`" VM argument via `Run > Run Configurations`

Part B — Unit testing

How can we tell if a program is fit for its intended purpose?

- From worst to best:
  1. Trust the programmer's assurances
  2. Test the program a few times
  3. Conduct rigorous inspections and walkthroughs of the algorithm and code
  4. Test the program *a lot*
  5. Analyse formal models of the algorithm and/or program using a simulator or model checker
  6. Prove properties of the algorithm and/or program mathematically
- In practice, testing (a lot) remains the best compromise for most software development projects

Limitations of testing

- Testing is the most widely-used way of checking a program's correctness, but it has several problems:
  - Comprehensive testing, using all possible input values, is infeasible for non-trivial programs
  - Picking test cases at random is unlikely to achieve good test *coverage*
  - In general, testing can reveal the presence of errors but cannot prove their absence

Why do programs fail?

- Mistakes made by the programmer:
  - Poorly designed *algorithms*
  - *Coding* errors
- Problems outside the programmer's control:
  - The program being used in ways it's not designed for
  - Faults in the program's development or run-time environment
- Mistakes made by the person who commissioned the program
  - Omissions in the program's *requirements specification*

What is unit testing and what are its advantages?

- Testing a program from the bottom up, one ‘unit’ at a time
  - Units are typically *methods* in object-oriented programs
- Well-chosen tests help *document* the code
  - They can be derived from the published API
  - They specify *what* the code must do (in certain situations) but do not constrain *how* the code works
- The tests can be *automated* to allow rapid assessment of progress
  - The number of passed vs failed tests can be displayed and tracked
- The tests can be used to support *regression testing*, i.e., testing a modified program to ensure that the change hasn’t ‘broken’ anything
  - This requires that the tests are maintained in sync with the code

Test-driven development

- Automated unit testing is a cornerstone of *test-driven development*, as used in Agile programming methodologies
  - Tests are developed *before* the code is written
  - Once the code passes the tests, new tests are added
  - The code is then extended to pass the new tests and refactored, if necessary, to improve its structure
- These cycles are kept as short as possible, e.g., just a few days
- The process finishes either when all of the sponsor's tests are passed (unlikely) or when the available time or money run out (more typical)
  - But no matter when the project stops, there is a usable product at the end
- We will examine test-driven development in detail in a later lecture

Reference:
P. Schuh. *Integrating Agile Development in the Real World*. Thomson, 2005.

Testing systematically

- For unit testing of methods there are two basic strategies for designing test suites:
  - *Black box* testing derives test cases from the requirements specification (aka ‘testing to spec’)
  - *Glass box* testing derives test cases from the program code (aka ‘testing to code’)
- *Both* should be used for critical programs!

Reference:
S. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, sixth edition, 2005.

Designing black box test suites

1. Consider both the expected *input and output values*
2. Partition the inputs and outputs into ranges, or *equivalence classes*, within which the same behaviour is expected
3. For each equivalence class produce:
   a. A set of typical tests which cover various cases depending on the data type: zero/one/many; less than/equal to/greater than; member/non-member; equal/non-equal
   b. A set of *boundary value* tests at and around the extreme limits of the equivalence class

Designing glass box test suites

- Glass box tests consider the structure of the program code and aim to exercise each part
- Various forms of glass box test coverage have been proposed, and are supported by debugging tools:
  - *Statement coverage* — every statement is executed at least once
  - *Branch coverage* — each condition is evaluated at least once
  - *Path coverage* — all control-flow paths through the code are executed at least once
- Iterative statements (loops) pose a serious challenge for glass box testing

Java and JUnit

- JUnit is a unit testing framework for Java programs:

  `http://www.junit.org`

- Tests are written in Java source code using special *annotations*
  - Running the tests automatically shows which ones pass and which fail
  - However, devising the tests and interpreting the results still depends on the programmer's skill
- Tests can *pass*, *fail* or produce an *error*
  - An error indicates that the test itself is invalid, usually because it threw an uncaught exception
- Eclipse is configured to act as a JUnit 'runner' so no `main` method is required to run the tests
- This approach is much more elegant and maintainable than writing 'tester' main programs as we did in last week's prac

Creating JUnit test classes

- Tests appear as parameterless **`public void`** methods preceded by an annotation:
  - `@BeforeClass` / `@AfterClass` — this method is executed once before/after any tests are performed
  - `@Before` / `@After` — this method is executed before/after each test is performed
  - `@Test` — this method is a test to be performed
  - `@Test(expected =` *X*`.class)` — this test is expected to throw exception *X*, i.e., an exception is considered a success!
  - `@Test(timeout =` *T*`)` — this test fails if it runs for longer than *T* milliseconds, which allows us to test code with infinite loops

JUnit assertions

- Within a test method, individual tests are written using JUnit's Assert class (not to be confused with Java's **`assert`** statement):

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

- A variety of JUnit assert methods are available:
  - `assertEquals()`
  - `assertArrayEquals()`
  - `assertTrue()`
  - …
- Like Java's built-in assertions, these methods can take an optional string to document the reason for the failure

Running a suite of test classes

- To run multiple test classes as a single test use the following annotations on a class with no body:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({class_name, ...})
```

Part C — Demonstrations

Demonstrations

1. Inserting assertions into a method: `CoinBias`
2. Testing a (stateful) class: `RainfallAverage`
3. Multiple test classes for a (stateless) class: `IncomeTax`

Taxable income	Tax bracket threshold	Tax on threshold amount	Marginal tax rate on amount above threshold
$0 to $6,000	$0	$0	0%
$6,001 to $30,000	$6,000	$0	15%
$30,001 to $75,000	$30,000	$3,600	30%
$75,001 to $150,000	$75,000	$17,100	40%
$150,001 and over	$150,000	$47,100	45%

Homework

- Read Sun's Java Tutorial on assertions:

  `http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`

- Read the JUnit Cookbook (such as it is):

  `http://junit.sourceforge.net/doc/cookbook/cookbook.htm`

- Read the JUnit FAQ:

  `http://junit.sourceforge.net/doc/faq/faq.htm`

- Browse the JUnit API:

  `http://junit.sourceforge.net/javadoc/index.html`