UCL

COMP1008
Unit Testing Classes

UCL

Testing in practice

- Test always and often.
  - Re-run all your tests every time you edit and compile any code.
- This implies that testing is a core activity of the programming process.

UCL

Repeatable

- Tests must be repeatable.
- Test data should be the same each time a test is run.
- New tests should be added and existing tests retained.
- “Ad Hoc” testing is no good.

UCL

Automated

- Testing should be automated.
  - A *test framework* runs the tests and checks the results.
- Manual testing is error prone and boring.
  - It won’t be done properly. Ever.

UCL

Thinking about a Test

- Purpose – what is being tested and why?
  - What is the specification of the method/class being tested.
- Design – how does a test advance the design.
- Test data – data used for testing.
- Test procedure – how the test is carried out.
- Expected results – what you expect to happen.
- Likely errors – is the test doing something likely to find an error?
- Confidence – does the test give you confidence your code is correct?

UCL

Test First

- Write your test first.
  - Work in small steps.
- Then the program code you need to be tested.
  - If the test is hard or impossible to write your program design is wrong.
- Use testing to find errors as early as possible.
- Use testing to guide the design your program.

Testing Class Based Programs

- All classes must be tested, individually (*unit testing*) and in collaboration (functional testing).
- The program as a whole is also tested (acceptance testing).
- Primarily concerned with unit testing here.

JUnit – www.junit.org

- JUnit is a very widely used unit test tool.
- Lightweight and straightforward to use.
  - You will be using it lots next year in 2007.
- Visit the web site and see what you make of it.
  - Use JUnit to test your mini-project.
- v3.8.1 has been in use for several years
- v3.8.2 recently released (minor updates)
- v4 also now released.
  - Rewritten for Java 5.
  - Not using it here.

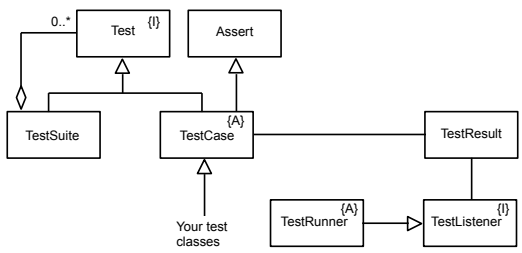
JUnit (1)

- A testing framework.
- Provided as a jar (library) file, junit.jar.
  - Must be on your classpath (see web)
- Integrated into tools like BlueJ and Eclipse.
  - For this course BlueJ is recommended.
  - See www.bluej.org.

Unit?

- A “unit” is a specific piece of functionality.
  - A class.
  - A method.
  - A set of related methods.
  - A behaviour.
- Fine grained.
  - Basic principle is to work one small step at a time.

JUnit (2) Basic JUnit Framework



TestCase

- A *test class* you write is a subclass of TestCase
  - Inherits the ability to run tests.
- A TestCase contains one or more *test methods*.
  - Literally methods whose name starts with *test*:
    - public void testGetName()
    - public void testResult()
- A TestCase creates and initialises one or more *fixture* objects.
  - A fixture is an object used for testing by calling its methods.

Assertions

- A test asserts something is true:
  - assertTrue(value == 3)
  - assertEquals("UCL",obj.getName())
  - assertNull(aRef)
- If an assertion fails, the test containing the assertion fails.
  - Or really, the test succeeded in finding an error.

Testing Process

- Create and initialise fixture object(s).
- Call public methods and check results.
  - Either those returned directly,
  - Or by calling another public method to check state of object.
- Private methods/variables are tested indirectly via public methods.
  - If you lack confidence that this is good enough, change your design or scrap the code.

Example – Book class

```
public class Book {
  private String title;
  private String author;
  public Book(String title, String author)  {
    this.title = title;
    this.author = author;
  }
   public String getTitle()   {
      return title;
   }
   public String getAuthor()  {
      return author;
   }
}
```

In this case we need to test that objects are properly initialised.

Note that instance objects are *immutable*.

class BookTest

```
public class BookTest extends junit.framework.TestCase
{
   private Book b;
   protected void setUp()  {
      b = new Book("a","b");
   }
   public void testGetTitle()  {
      assertEquals("a",b.getTitle());
   }
   public void testGetAuthor()  {
      assertEquals("b",b.getAuthor());
   }
}
```

Fixture object.

setUp Method to initialise fixture.

Test Methods

Running the tests (1)

- From command line:
  java junit.textui.TestRunner BookTest
- Displays

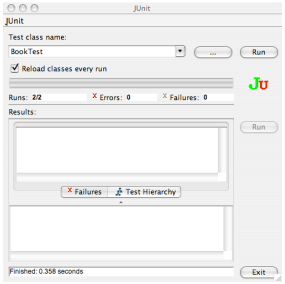
  ..

  Time: 0.012

  OK (2 tests)

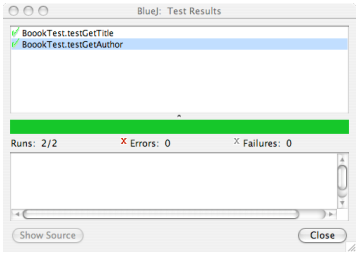
Running the tests (2)

- Using the GUI TestRunner
  java junit.swingui.TestRunner BookTest



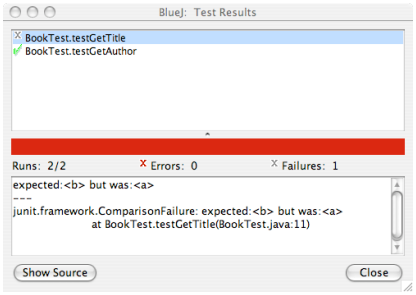
Running the Test (3)

- From BlueJ (use right button menu on BookTest icon)



Green Bar — GREEN

If a test fails...



Red Bar — RED

Running tests

- This happens automatically:
  - call setUp to initialise fixture(s).
  - run test.
  - call optional tearDown method to remove fixtures.
- A test always runs with a new copy of the fixture(s).
  - Running one test must not affect running another.

Class Library (1)

```
public class Library
{
    private String name;
    private ArrayList<Book> books;
    public Library(String name) {
        this.name = name;
        books =
          new ArrayList<Book>();
    }
    public String getName() {
      return name;
    }
    public void addBook(Book b) {
        books.add(b);
    }
    public Book searchByTitle(String title)  {
        for (Book b : books) {
            if (b.getTitle().equals(title)) {
                return b;
            }
        }
        return null;
    }
}
```

Void method - result of calling it will be tested by using search method.

Class LibraryTest (1)

```
public class LibraryTest extends junit.framework.TestCase
{
    private Library library;

    protected void setUp()  {
        library = new Library("name");
        library.addBook(new Book("a","b"));
    }

    public void testGetName() {
        assertEquals("name",library.getName());
    }
// Continues on next slide
```

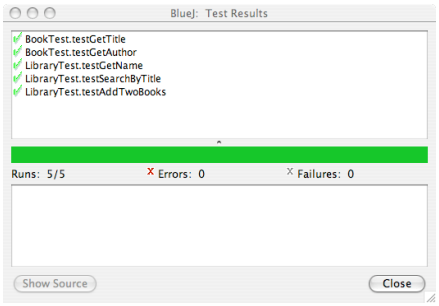
Initialise fixture.

Class LibraryTest (2)

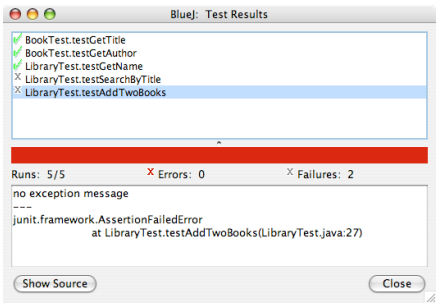
```
    public void testSearchByTitle() {
        Book b = library.searchByTitle("a");
        assertNotNull(b);
        assertEquals("a",b.getTitle());
    }
    public void testAddTwoBooks() {
        library.addBook(new Book("c","d"));
        Book b1 = library.searchByTitle("a");
        assertNotNull(b1);
        assertEquals("b",b1.getAuthor());
        Book b2 = library.searchByTitle("c");
        assertNotNull(b2);
        assertEquals("d",b2.getAuthor());
    }
}
```

Tests depends on addBook in setUp.

Run the tests...



If there are errors...



Reminder: Basic Strategy

- Test the public methods.
- Methods that return a value: call the method and check value returned.
- Void method: call method and then call another non-void method to check right thing happened (e.g., addBook then search).
- Work one test at a time.
- Take small steps.
- Keep it simple (YAGNI - You Ain't Gonna Need It).
- DRY - Don't Repeat Yourself.

Can't test this method...

- Modify or get rid of it!
- Or add a non-void method to return a value that can be checked.
- Having testable code is more important than "perfect" design.
  - But no excuse for being sloppy.
  - Rethink if getting messy.

Can I test...

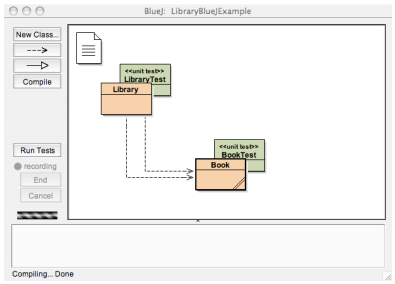
- GUIs, database access, file handling, networking...
- Yes! But beyond the scope of this introduction.
  - Wait for next year.

What do I need to know now?

- The basic ideas, as shown in the example:
  - Test class as subclass of TestCase.
    - Collection of tests.
  - Using fixture(s) - objects to test.
  - setUp method to initialise fixture(s).
    - New fixtures created for every test.
  - Test methods - call method and check result.
    - Use assertions.

Try it out!

- Use BlueJ and try out some examples.



Summary

- Testing is a core part of the design and programming process.
- Testing is used to find bugs and errors, so they can be fixed at the earliest opportunity.
- Test early, often and always.
- Testing relies on establishing an acceptable degree of confidence, not on “proof”.
- Testing is essential!