Testing in Java

- **Time:** 6:30 to 9:00 then bar til 9:30. No rush, cover what we can.
- **Format:** Talk, (D)emo, (E)xercise, (S)olutions & "more"
- **Agenda:**
  - **Backgound** and different kinds of testing
  - **Unit Testing**: JUnit
  - **Simplifying tests**: Annotations, Hamcrest
  - **Test Dummies**: Stubs, Mocks, verifying interaction

Testing

- **Purpose**: Prove that it (still) works
- **Originally**: Nasty afterthought, for somoeone else!
- Software **Engineering**: Cheaper to find faults **early** (cf recall, death!)
- **Kinds**:
    - **White box**: Unit, Regression, Integration
    - **Black box**: System, UAT
    - **Other**: Performance, Security

Unit Testing

- Test each component separately
- "White box"- test can see innards of **C**lass **U**nder **T**est
- "Unit" is a single class, query, or similar chunk
- We test `public` methods, "the interface" of our CUT
    - Several test-methods for each CUT method
    - Might test `package` level features too
    - Might refactor CUT for testability (or subclass, eg car dipstick)
    - Encourages modularity (Test is unit's first "interaction")
    - Aids understanding (Test is a good example use-case)
    - Coverage leads to reliability- and confidence to edit!

Early Unit Tests

- Just write some code that:
  - Calls CUT's methods
  - Checks & prints results
- Typically a `main` & other methods, removed for production build
  - `#if if /* comment */`
- Messy! Pollutes production code with test code
- Who is going to run each test (and look at the output!)

JUnit to the rescue!

- **Automated** Unit Testing framework
- **Self Verifying** So we dont need to read the output, test can only:
  - Pass, Fail, Error, Ignored
- Three basic **steps**:
  - Setup: `create and arrange objects to test`
  - Invoke: `actual= anObject.aMethod(s)…`
  - Verify: `assertEquals(expected, actual)`
- **Demo & ex** Eclipse: CUT> RightMouse> Create Test
  - Same package (white box, can see innards)
  - Different source folder (so as not to pollute main code)

Basis of a good test

- **F**ast
  - People avoid slow things / holdups
- **I**ndependent
  - Shouldn’t depend on other tests `(usually)`
- **R**epeatable
  - Should give same outcome no matter how many runs
- **S**elf validating
  - Defines its own success criteria, automatic
- **T**imely and **T**horough
  - Thorough **coverage**, written around same **time** as code (See **TDD** later)

Simplifying tests

- **Essential!** - eg @Before / @After - `each` test
- @Test(**expected=** AnExeption.class) **Demo** (ex later)

More:

- @BeforeClass , @AfterClass **static** methods (before `all` tests in class)
  - These run **before all** not **before each**
- Combining Test classes into Suites (also Categories & Groups)
  - @RunWith(Suite.class)
  - @Suite.SuiteClasses({…})
- Data-driven tests
  - @RunWith(Parametirized.class)
  - @Parameters method returns values for ctor

Simplifying tests using Hamcrest Matchers

- assertThat(`actual` , `aMatcher()`)
  - static factory method, == `new Matcher(expected)`
  - Matcher `objects` remember state (eg their role), subclass, agregate,…
- State allows descriptive failure message, cf "expected true got false"
  - even better than `assertEquals` … descriptive arg1 string
- No more confusion between `expected, actual` args which have same type
- Can write our own Matcher classes
- Many built-in Matchers for us to combine eg:
  - `is()` , `not()` , `equalto()` , `sameInstance()`
  - `allOf(m1,m2,m3)` , `anyOf()`
  - Much more eg `contains` , `containsString` , `startsWith`
- **Demo & ex**

Timely Tests - Test Driven Development

- When to write tests
    - **Soon** after development of a feature
      While you have your head around it.
- **OR** just **before** writing a feature - **TDD**
    - Part of Agile **XP** (by Kent Beck, developer of JUnit).
    - Write a failing test
    - Code **only** enough to make it pass (or fake it for now!)
    - Refactor! (incremental code can get messy)
    - Keeps focus, Maximise amount of work not done. **KISS** , **YAGNI**
      Aids "top down" approach, write "spec" first
      Eases understanding of problem domain (small steps)
      Seems slow but saves **a lot** of later debugging (catch bugs early!)

Test Doubles

Dummy modules that mimic behaviour of code that our CUT interacts with [ie same `interface`]

- What if our Unit calls code that:
  - Doesnt exist yet, Is unpredictable, Has rare or unpredictable behaviour
  - Just like crash-dummy in car safety tests
- Dummies allow testing of rare behaviour (car crash, network failure)
- eg Stub class, just mimic "aspects" of behaviour relevant to this test. (Crash dummy doesnt need all Person features, maybe even just a torso.)
- Use **MOCKS** to alleviate plethora of stubs. Mocks are dummies created "on-the-fly"
  - Define mock's behaviour from code in the Test class eg using `Mockito` :
    when(mock.method(2)).thenReturn(7)
- `EasyMock` takes this further by verifying interactions that took place between CUT and mock.
  (eg expand airbag test by checking crash dummy's sensors.)

Further Techniques

- Other tools
  - Selenium UI tests
  - Spring Test
  - Jenkins- Continuous Integration
  - Apache JMeter- performance
  - **J**ava **M**icrobenchmark **H**arness
- **B**ehaviour **D**riven **D**evelopment…
  - JBehave
  - Cucumber

Behaviour Driven Development

- Scenario script from business user:
  - Scenario:
    Fat cat eats too much.
    Given a cat named Sid weighing 4 kilos.
    When we feed him 5 cans of tuna.
    Then his weight should become 6.2 kilos.

- Feeds into Unit Test code:
  - @Given("a cat named $name weighing $wt kilos.")
    `public void setup(String name, double wt) { …`
    @When("we feed him $qty cans of tuna.")
    `public void invoke(int qty) { …`
    @Then("his weight should become $wt kilos.")
    `public void verify(double wt) { …`

Thanks for listening!

Any Questions?

mike.burton@mycosystems.co.uk