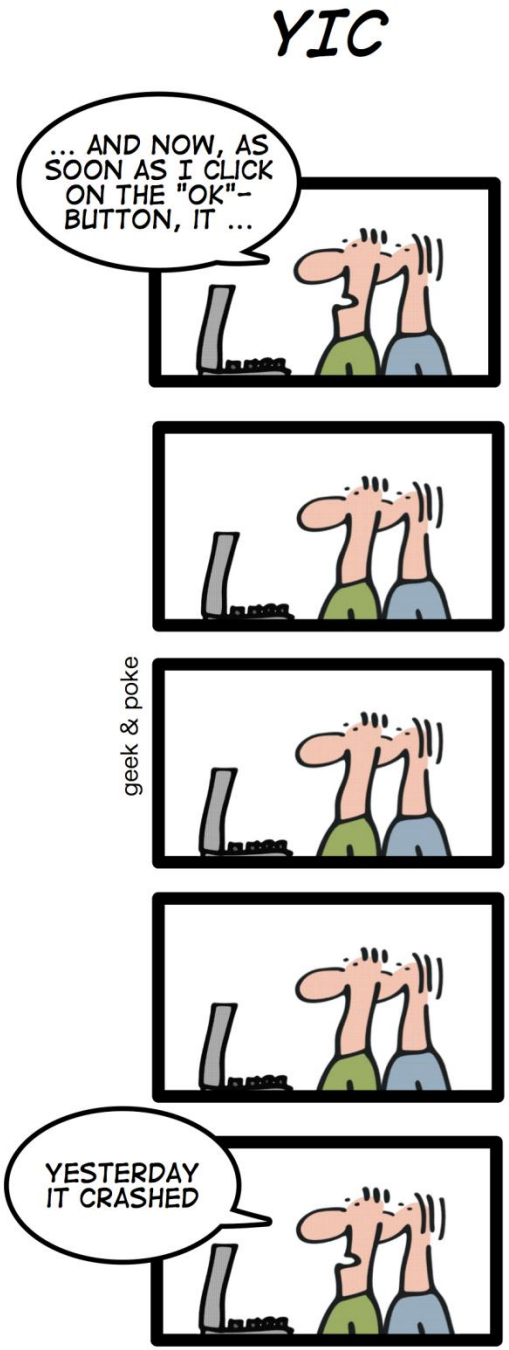

Unit Testing,

JUnit

& Cobertura

geek & poke/Oliver Widder

Unit Testing (1/3)

- Unit testing is sometimes referred to as component testing
- Performed by the/a development team
- Aimed at ensuring that source code units work as expected
- A unit is the smallest testable part of a software (e.g. a function, a class)
- Unit testing requires to apply the white box technique

Unit Testing (2/3)

- For each unit:
  - A test case is defined
  - A test case consists in observing the results based on a specific input
  - Test cases must run isolated from other test cases
  - It's possible to use helpers for that:
    - A stub is a piece of very simple code replacing a real one
    - A mock simulates the execution of real pieces of code
    - A harness is a set of test data

Unit Testing (3/3)

- Some of the benefits of unit testing:
  - Introduces a bit of regression testing
  - Eases change by ensuring that an updated code doesn't break the existing behavior
  - Unit testing provides some kind of documentation

Unit Testing Frameworks (1/2)

- Jtest
  - Goes far beyond unit testing (code analysis, code review, etc.)
  - Closed source
  - http://www.parasoft.com
- JUnit
  - Is part of the big xUnit family
  - Open Source (CPL 1.0)
  - http://junit.sourceforge.net

Unit Testing Frameworks (2/2)

- Cactus
  - Is an extension to Junit which allows unit testing server-side Java code
  - Open Source (APL 2.0)
  - http://jakarta.apache.org/cactus/
- TestNG
  - Is inspired by JUnit but provides more features (such as data-driven testing)
  - Open Source (APL 2.0)
  - http://testng.org

JUnit – Example (1/3)

```
public abstract class MathUtils {
  public static long roundDouble(double d) {
    String dAsString = Double.toString(d);
    int commaPos = dAsString.indexOf('.');
    char digitAfterComma = dAsString.charAt(commaPos + 1);
    if(digitAfterComma >= '5') {
      return Long.parseLong(dAsString.substring(0, commaPos))+1;
    }
    else {
      return Long.parseLong(dAsString.substring(0, commaPos));
    }
  }
}
```

JUnit – Example (2/3)

```
import junit.framework.TestCase;
public class MathUtilsTest extends TestCase {
    public void testRoundDouble() {
        assertEquals(
            MathUtils.roundDouble(1.234), 1);
        assertEquals(
            MathUtils.roundDouble(1.567), 2);
    }
}
```

JUnit – Example (3/3)

```
import static org.junit.Assert.*;
import org.junit.Test;
public class MathUtilsTest {
    @Test
    public void roundDouble() {
        assertEquals(
            MathUtils.roundDouble(1.234), 1);
        assertEquals(
            MathUtils.roundDouble(1.567), 2);
    }
}
```

JUnit (1/2)

- JUnit features:
  - Clear separation between tested code and testing code
  - Console- or Swing-based UI
  - Integration with many IDEs (Eclipse, NetBeans, etc.)
  - Integration with Ant (with nice reporting features for use by other tools)

JUnit (2/2)

- JUnit comes in two flavors:
  - JUnit 3.x (currently, 3.8.2) can be used in environments running on Java 1.4
  - Cf. example #1
  - JUnit 4.x (currently, 4.5.1) is intended to be used for environments running Java 5.0 and greater
  - Cf. example #2

JUnit Concepts (1/3)

- `TestCase`
  - Is a `Test`
  - Class aimed at containing unit tests
  - Usually, one `TestCase` per tested class
- `TestSuite`
  - Is also a `Test`
  - Composite of `TestCase`s or other `TestSuite`s
  - Used to run multiple tests

JUnit Concepts (2/3)

```
import junit.framework.Test;
import junit.framework.TestSuite;
public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite(
            "Test for JUnitSamples project");
        suite.addTestSuite(MathUtilsTest.class);
        return suite;
    }
}
```

JUnit Concepts (3/3)

- `TestRunner`
  - Used to run `TestCase`s or `TestSuite`s
  - Can be console- or GUI-based (`junit.textui.TestRunner.run(Test)` or `junit.swingui.TestRunner.run(Test)`)
  - Usually provided by IDEs, etc.
- `TestResult`
  - Data structure used to collect results of tests

JUnit 3.x – Writing Test Cases (1/2)

- A test class must extend `junit.framework.TestCase`
- Each test has to be implemented as a method of the test class
- Test methods must start with `test` (`e.g. testRoundDouble`) and return `void`

JUnit 3.x – Writing Test Cases (2/2)

- Fixtures are performed using the `setUp` method
  - `setUp` is called before each test is run
- Cleanup is done using the `tearDown` method
  - `tearDown` is called after each test is run
- Asserts are performed using the `assert*` methods:
  - `assertEquals`, `assertFalse`, `assertNull`, `assertNotNull`, `assertNotSame`, `assertSame`, `assertTrue`
  - `fail`

JUnit 4.x – Writing Test Cases (1/4)

- Every class can be a test class
- Each test has to be implemented as a method
- Test methods must:
  - Be annotated with `@org.junit.Test`
  - Return `void`
- Tests can be skipped if they are annotated with `@org.junit.Ignore`

JUnit 4.x – Writing Test Cases (2/4)

- Fixture methods are annotated with `@org.junit.Before`
- Mutiple fixture methods can be defined
- For a fixture method to be run once for all the tests in a class, it has to be annotated with `@org.junit.BeforeClass`

JUnit 4.x – Writing Test Cases (3/4)

- Cleanup methods are annotated with `@org.junit.After`
- Mutiple cleanup methods can be defined
- For a cleanup method to be run once for all the tests in a class, it has to be annotated with `@org.junit.AfterClass`

JUnit 4.x – Writing Test Cases (4/4)

- Assertions are provided by the `org.junit.Assert` class as static methods

- Example: `static import org.junit.Assert.assertEquals;`

- New methods compared to JUnit 3.8: `assertArrayEquals`, `assertThat`

JUnit & Ant (1/5)

- Ant provides two tasks to work with JUnit
- `<junit>` is used to:
  - Run `TestCase`s and `TestSuite`s
  - Capture and export results (as text or XML)
- Noticeable attributes of `<junit>`:
  - `haltonerror`
  - `haltonfailure`

JUnit & Ant (2/5)

- Noticeable nested elements of `<junit>`:
  - `<test>` (main attribute: `name`)
  - `<batchtest>` (main element: `<fileset>`)
  - `<formatter>`

JUnit & Ant (3/5)

```
<project …>
  …
  <target …>
    <junit printsummary="yes">
      <formatter type="xml"/>
      <test name="MathUtilsTest"/>
    </junit>
  </target>
  …
</project>
```

JUnit & Ant (4/5)

- `<junitreport>` is used to merge a set of `<junit>` XML results
- It is also used to apply XSL stylesheets
- Noticeable attributes of `<junitreport>`:
  - `tofile`
  - `todir`
- Noticeable nested elements of `<junitreport>`:
  - `<fileset>`

JUnit & Ant (5/5)

```
<junitreport printsummary="yes">
  <fileset dir="${base}/test/report">
    <include name="TEST-*.xml"/>
  </fileset>
  <report
    format="frames"
    todir="${base}/test/report"/>
</junitreport>
```

Code Coverage

- Is a complement to software testing
- Way of measuring software testing accuracy
- Provides statistics on the way software has been tested

Cobertura

- Cobertura is a code coverage tool for Java
- Cobertura runs either:
  - From the command line
  - From Ant
  - From Maven (not official)
- Uses bytecode injection (instrumentation) to analyze the code
- Produces XML or HTML reports

Using Cobertura with Ant (1/5)

- Three steps to get code coverage using Cobertura:
  1. Instrument the Java classes
  2. Run the unit tests
  3. Generate the Cobertura report

Using Cobertura with Ant (2/5)

```
<path id="cobertura.classpath">
  <fileset dir="lib">
    <include name="*.jar" />
  </fileset>
</path>
<taskdef
  classpathref="cobertura.classpath"
  resource="tasks.properties"/>
```

Using Cobertura with Ant (3/5)

```
<cobertura-instrument
  todir="instrumented-bin">
  <fileset dir="bin">
    <include name="**/*.class"/>
  </fileset>
</cobertura-instrument>
```

Using Cobertura with Ant (4/5)

```
<junit fork="yes" forkmode="once">
  <classpath location="instrumented-bin"/>
  <classpath location="bin"/>
  <classpath refid="cobertura.classpath"/>
  <formatter type="xml"/>
  <test name="MathUtilsTest.class"/>
</junit>
```

Using Cobertura with Ant (5/5)

```
<cobertura-report
  destdir="report-cobertura">
  <fileset dir="src">
    <include name="**/*.java" />
  </fileset>
</cobertura-report>
```

Cactus (1/3)

- Cactus is used to test server-side Java code
- Cactus extends JUnit
- Cactus runs in-container
- Cactus provides the following `TestCase`s:
  - `org.apache.cactus.JmsTestCase`
  - `org.apache.cactus.JspTestCase`
  - `org.apache.cactus.FilterTestCase`
  - `org.apache.cactus.ServletTestCase`

Cactus (2/3)

- Each `TestCase` type provides a set of attributes to be worked with (e.g. `session`, `request`, `response` and `servletConfig` for `ServletTestCase`)
- `setUp` and `tearDown` are performed on the server side
- `begin` and `end` methods are equivalent to `setUp` and `tearDown` but on the client side

Cactus (3/3)

```
public class TestSampleServlet
    extends ServletTestCase {
  public void testMyServletCompute() {
    MyServlet servlet = new MyServlet();
    servlet.init(config);
    session.setAttribute(
        "username", "John Doe");
    assertEquals(
        "JDOE", servlet.compute(request));
  }
}
```