Class #13 (2004/02/17)

- Changes/Review
- Progressing to Code
- Test Types, level & visibility
- HW2 Q&A (also in S.I. Lab session)
Changes & Lecture 12 Takeaway

- Changes/Notices/Comments
  - inspections lab today, 1-2
  - HW2 and ATP due Thursday
  - Sliding “white box” discussion to Thursday, after reading
  - Some notes re HW 2 on DForge

- Takeaway lecture 12 - Exceptions and Assertions
  - Exceptions should have first-class representation in design
  - Exceptions used for contract precondition check and exceptional conditions
  - Assertions used to support debugging; invariants, postconditions, private methods
Progressing to Code

- Establish tools
  - easiest if all use same toolset (freeze; synchronize)
  - recommended: Ant, JUnit (www.junit.org)
  - set up an Ant build process; include targets for clean, checkin, checkout, compile, jar, junit, release, test, doc, ...
  - Ant script(s) go in svn, too!

- Establish build policy
  - “buildmaster” (mostly automated)
  - build from Repository (checkout, build, test)
  - release on working incr of function, and tag in svn

- Practice now, before you have substantial code
Repository/Package Structure

**Design now!** (complicated to refactor later) I recommend:

- build.xml
- Makefile
- ...

```
src/
  <name>/
  server/ <= public APIs
  impl/ <= code, unit tests
  test/ <= integration tests

client/
  common/ <= interfaces, utils
lib/ <= 3rd party libs
conf/ <= conf files
tests/ <= intg, accept scripts, support
```

```
built/ <= output dir, recreated each build
classes/ <= compiler output
docs/ <= release notes
test/ <= JUnit results
api/ <= optional, JavaDoc output
dist/ <= JARs packaged for installation
```
Naming Conventions

- Have one!
  - “Names” applies to variables, methods, classes, files, properties, ...
  - all ‘public’ names should be reviewed by another team member
  - names should help readability
    - phrase as message to target
    - where OO, beware of set/get (exposes state)
    - use action message names (desired outcome)
    - use isFoo() message to get boolean returns
  - use patterns: “can I grep this name?”
Division of Labor

- Suggest owners for build/tools, client, server, integration test
  - everybody handles all code, but owner reviews
  - coder writes unit tests
  - *very* frequent build ensures nobody falling behind
- Alt: functional area owner (crosses client/server)
- Set up task-TODO list
  - keep in repository; edit in new tasks, done, change
  - simple bug track
Start clean, stay clean

- Provide API, use from GUI and tests
- Client log to server, unified logfile
- JavaDoc, particularly for public API
- public constants on interfaces, not classes
- Everything private by default
- Small, simple methods (“single responsibility”) 
- Tags convention in code
  - // TODO something here runs, but incomplete
  - //FIXME runs but returns wrong or stubbed result
  - // FUTURE improvement possibility here
Tool Recommendations
JUnit Sample Unit Test

class ServerTest extends junit.framework.TestCase{

    // some like TestServer, for alphabetizing; I like
    // ServerTest for easily seeing that Server has a Test.

    /**
     * Unit test ensure nulls handled
     */

    public void testNullsInName() {
        Server s = new Server(testServerUrl, "my_server");
        assertEquals("my server", s.getFullName);
    }
}

>java junit.textui.TestRunner ServerTest
1) testNullsInName

AssertionFailedError: expected:<my server> but was:<my >

... Use junit.swingui.TestRunner for GUI
JUnit assertXXX

- `assert()` // deprecated, conflicts with J2SE 1.4 assert
- `assertEquals(x, y)` // .equals, or values
- `assertFalse(boolean expr)`
- `assertNotNull(obj ref)`
- `assertNotSame(x, y)` // mem ref check using ==
- `assertNull()`
- `assertSame()`
- `assertTrue()`
- `fail()` // commonly used in exception handler testing
JUnit Sequence

- New instance of test case for every test method
  - $5 \text{test<foo>}$ means five objects
  - no sharing of instance variables between tests
- For each test method object
  - setUp()
  - test<foo>
  - tearDown()
JUnit suite()

- Multiple TestCase group into suite()
- if no suite(), Junit uses reflection to add all testXXX to default suite
- Can use public static Test suite(){…} to incorporate subset of tests, to control order of tests, or to wrap with decorators
- Repeated test:

```java
public static Test suite(){
    return new RepeatedTest(new TestSuite(ServerTest.class), 10);
}
```
JUnit One-time setUp()

- `junit.extensions.TestSetup()` allows wrapping one-time setup and tear-down around a set of tests, but you have to create your own `suite()`. Not hard:

```java
public class ServerTest extends TestCase{
    public void testNullInName(){…}
    public void testServerShutsDown(){…}

    public static Test suite(){
        TestSetup setup = new TestSetup(new TestSuite(ServerTest.class)){
            protected void setUp throws Exception {
                // do one-time set up here
            }
            protected void tearDown() throws Exception {
                // do one-time teardown here
            }
        };
        return setup;
    }
}
```
JUnit Practices

- One function check per test method
  - failure aborts entire method
  - usually one assert per test but combine asserts when they are ANDing against same functionality
- **Use** `setUp()` and `tearDown()` **for common init/cleanup code**

- Exception test:

  ```java
  public void testPassNullsToConstructor(){
      try{ Server s = new Server(null, null);
          fail("Expected IllegalArgumentException");
      } catch (IllegalArgumentException expected){}
  }
  ```
Ant - “Make Without Wrinkles”

- Java application
- Runs build.xml by default
- Handles file dependencies - sort of … uses timestamps, not logical info. (Change base class method, won’t always recompile derived class.) So, do ant clean compile frequently
- Has default target, but selectable
  - ant clean
  - ant compile
  - ant test
- Explicit depends between targets, Ant sorts out a target order
- Sample build.xml in DForge Docs area
Ant Practices

- Top 15 Practices paper in DForge docs area
- Subversion? Not clear yet
  - Ant has CVS task, and I’ve seen mention of a Subversion task
  - Eclipse “subclipse” plugin supposed to handle this
  - Checking now
  - Fallback: separate process to check in/out, build from local sandbox
Ant & JUnit Together
Installation

- Some people like to:
  - put ant.jar, optional.jar and junit.jar in your svn/lib
  - write custom ant.bat script, in dir with buildfile
  - in ant.bat, put the jar files on the classpath; do not incluse user’s env, invoke Ant
    - no need to install Ant, JUnit on each dev machine
    - limits impact of user variability
    - synchronizes versions of Ant/JUnit across team
Testing

- With consistent naming, use Ant’s `junit` and `batcuestest` targets to locate and execute tests

- Run one test with nested test element in junit target

```xml
<target name="junit2" depends="compile">
  <!-- ant -Dtestcase=server/TestServer junit2 -->
  <property name="testcase" value="TestServer" />
  <junit fork="false">
    <test name=${testcase}”/>
  </junit>
</target>
```
Reporting

- **In `junit` target:**
  - `<formatter type="xml"/>` get one xml file/test
  - `<batchtest todir="${dir.build}"/>` put XML

- **In `junitreport` target**
  - `<fileset dir="${dir.build}"/> <include name="TEST-*.xml"/></fileset>`
  - `<report format="frames" todir="${dir.build}"/>"
Tool references

- Ant - http://ant.apache.org
- JUnit - http://www.junit.org
- Antenna Ant tasks for J2ME MIDP
  http://antenna.sourceforge.net/
- Clover (test coverage) free use arranged
  - needs Ant/Junit
- IDEs? Eclipse, SunOne, NetBeans, Jbuilder, ...
W/E 17 Feb:
A: 3 Astrix for paper proto
A: 2 Cae good org meeting
A: 4 Cae exceptional design doc
W: 1 Unk for Niraj slide-checking
W: 2 Unk for real-time prototyping demo
W: 1 Cae for Ed asking “HW2P1sig digits”
W: 1 Astrix for 1st Lecture Review (Nesan)

<table>
<thead>
<tr>
<th></th>
<th>27-Jan</th>
<th>3-Feb</th>
<th>10-Feb</th>
<th>17-Feb</th>
<th>24-Feb</th>
<th>2-Mar</th>
<th>9-Mar</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrix</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>BattleSnake</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>Caesar's</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>CLASS TOT</td>
<td>9</td>
<td>17</td>
<td>30</td>
<td>43</td>
<td>43</td>
<td>43</td>
<td>43</td>
<td>43</td>
</tr>
</tbody>
</table>

Current week:
Instructor awarded 5/6
TA awarded 9/9
Averaging 4.3/wk * 7 weeks = 75 BP … need 80 BP
Need 13/wk rate now

**NO FREE PIZZA AT THIS RATE**

… but I ain’t planning to be out of town
Prototyping a Snakepit ...
Test Types

- Visibility: Black vs. White
- Scope
  - Unit
  - Integration
  - System
  - Acceptance, alpha/beta
- Intent
  - Functional (original, regression)
  - Performance
  - Usability
Black and White

- **Black-Box**
  - Don’t “see inside”
  - Specification/requirements driven
  - Equivalence class & boundary tests

- **White-Box (sometimes “glass box”)**
  - See “inside the box”
  - Source code driven
  - Statement, branch, path coverage
Scope and Focus

- **Unit (individual module, component or class)**
  - focus on correctness of module
  - use both white- and black-box

- **Integration (interaction of units)**
  - focus on correctness of interaction
  - mostly black-box, some white-box

- **System (entire application)**
  - overall correctness, performance, robustness
  - black-box

- **Acceptance (purchase/deployment trigger)**
  - fitness for use by single customer
  - black-box and negotiation

- **Alpha/Beta (market drivers to quality)**
Test Planning Exercise

- Two rectangles are identified by 8 integer values, giving the Cartesian co-ordinates of the lower left hand corner and upper right hand corner of each rectangle.
- A program is required to calculate whether these two rectangles: overlap, just touch, do not intersect or the input data is invalid.

- 1. Draw 2 or 3 interesting cases for this problem
- 2. Compare your cases with those around you
- 3. When would you consider two different cases for the rectangle problem to be “the same” or “significantly different”?
Equivalence Class Testing

- **Method:**
  - Identify the entire input data space of the unit under test
  - Partition this input space into different classes.
  - Select a data element from each class and execute the unit using this input
  - Check that the output is as expected

- The **goal of equivalence class testing** is to select classes so that the behaviour of the SUT can be **assumed** to be correct for all data in the class if it is **known** to be correct for one data element of the class.
Equivalence Class Test Examples

- Input space: integers from 1 to 100
- Partitions:
  » normal input: [1,100]
  » error input: [-100,0] [100,1000]
- Test cases one value from each class: -1, 42, 200

- Input space: calendar dates
- Partition: several dimensions needed here - valid and invalid dates, 30/31 day months, Februaries in leap years
- Test cases: one from each class
Boundary Value Analysis

- *Special case of equivalence class testing*
- Choose equivalence classes as above
- Test data is taken from the **extremes of partitions** of input and output ranges
- Tests normally performed for values at and either side of each boundary
- Example (see previous page)
  - Input space: integers from 1 to 100
  - Boundary values 0, 1, 2 and 99, 100, 101
Unit Test: Equivalence Classes

- **Test Purpose:**
  - to detect output errors for given inputs using representatives of the entire input space

- **Evidence:**
  - a set of inputs offered and outputs observed

- **Assumptions:**
  - if behaviour of SUT is correct for one element in an input class then it is correct for all elements

- **Deduction:**
  - oracle: is output correct for a given input?

- **Verdict:**
  - Code correct for all inputs OR code revision required
Test Planning Exercise Revisited

- Reconsider the classes you identified for the rectangle problem at the beginning of the lecture. Can you now translate these into test data for the rectangle problem using equivalence class testing?
- Invent a *faulty* implementation to solve this problem.
- Would your test data detect your fault(s)? Swap test data with the people around you - who has the best set of test cases?
Output-driven testing

- Repeat equivalence/boundary test development given output target
- “what inputs should yield error output?”
- “what inputs *should* give me boundary result of...
Software Inspection Expectations

- Aaron will discuss details of “how” today 1-2
- Inspection will be done in a lab
  - 2 hr max
  - 200-400 SLOC (pick the most important!)
- Line-numbered listing of code to review
- Coding standard check printouts
ATP Expectations

- Key points
  - due Thur
  - traceability: each test includes *trace back* to requirements being validated
  - 1 hr long; prioritize so we can cut off test at 1 hr
    - state what requirements not in ATP due to time
    - practice this for real timing (“dry run”)
  - step-by-step (legal thinking)
  - fail test? Revise program or ATP; if ATP get TA approval; re-run
    - pass in 2 tries or “significant adverse effect”
- Formal, written (so meeting notes and time logs)
Test slide references