CMPS 115 Winter 04

Class #14 (2004/02/19)

- Changes/Review
- White Box Testing
- OO testing & Mock Objects
- <break>
- Performance by Design
- Presentation Expectations
Changes & Lecture 13 Takeaway

- Changes/Notices/Comments
  - Guest Lecturers confirmed:
    - 3/4 Glen Martin of Sun on “Programming in a Box”
    - 3/9 Wes Isberg of Aspect/J on AOP

- Takeaway lecture 12
  - Coding stage practices
    - design your package structure and repository now
    - stabilize your tool chain across the team; I recommend Ant, Junit
    - start a build policy and process now, before you have lots of code
  - Black Box Testing
    - Equivalence classes help you define good test inputs
    - Boundary testing is choosing your test points at edge of equivalence class
    - Verify fails-as-expected as well as works-as-expected
Test Types

- Visibility: Black vs. White

- Scope
  - Unit
  - Integration
  - System
  - Acceptance, alpha/beta

- Intent
  - Functional (original, regression)
  - Performance
  - Usability
Black and White (Reminder)

- **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 (Reminder)

- **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; usually analysis robustness, demo function, maybe demo reliability, test some performance

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

- **Alpha/Beta** (market drivers to quality)
Fault Models

- “Intuition” founded on strong understanding of the environment and the software
  - common faults in particular environment
    - technointerfaces (OS, UI, API, FS, and …?)
    - sociological (management, process, context)
  - common faults in particular tier
  - common faults in particular style/language
  - common faults in particular design school/method/tooling

- Develop through repeated ...
  - fault analysis
  - conscious categorization
  - trend analysis (bug tracking systems)
White Box Testing

- Subject of most of the reading; by now you understand DD-Paths (decision-to-decision)
- Data flow (DU, Def-Use paths)
- Theoretical basis fit “structured programming” rules and patterns; fits implementation inside methods but less useful checking OO code
- White-box metrics used to guide test suite creation
  - $C_0$ every statement; $C_1$ every DD-Path; $C_2$ is $C_1$ plus loops; $C_d$ is $C_1$ plus dependent-path checks, etc
  - sometimes used to indicate possible trouble spots
OO Testing

– OO emphasizes very small methods (so small most WB metrics are unstable or low-value)
– Main OO characteristics from test viewpoint:
  * information hiding
  * state-dependent behavior
  * polymorphism and dynamic binding
  * inheritance
– OO is about message and state maintenance, so testing is more an integration-level (black-box)
OO Testing Approaches - Internals

- Information hiding
  - state not supposed to be accessible nor specified
    - ‘oracles’ possible but complex and test breaks with evolution
    - brute force (break encapsulation) may create problems
  - Use equivalence scenarios
- State
  - Behavior depends on state - needs setup
  - Use transition-coverage from statechart specification
State-based Testing

- Use state diagram to determine a sequence of events.
- Tests should cover all states.
- All transitions should be tested at least once.
- Test also invalid and unrelated events.
Coverage Metrics

- **State Coverage**
  - Ratio of number of states covered and total number of states in the given state model

- **Event Coverage**
  - Ratio of number of events covered and total number of events in the given state model

- **Transition Coverage**
  - Ratio of number transitions exercised and total number of transitions in the given state model

- **State-Event Coverage**
  - Ratio of state-event pairs exercised and number of states multiply number of events in the given state model
OO Testing Approaches - Externals

- Polymorphism & dynamic binding
  - many test cases
  - generally, don’t depend on superclass testing
  - use transition-coverage from statechart specification
    if you’re lucky enough to have one :)

- Inheritance
  - should we retest inherited methods? (yes)
  - can we reuse a test of a superclass? (sometimes)
OO Testing - Mock Objects

- Stub objects for OO unit test support
  - useful for “integration” problems (like network code)
  - no internal behavior (not a ‘simplified’ version)
  - controlled by test code ("expect this" … "return that")

- Static vs Dynamic
  - static: compile time codegen or hand-generated
  - dynamic: some dynamic Proxy generator
public boolean canUserLogIn( String username,
        String password ) {
    UserManager um = UserManagerFactory.getInstance();
    User user = um.getUser( username );
    if (user != null) {
        if (user.validatePassword( password ))
            return true;
    }
    return false;
}
public void setUserManager( UserManager um ) {
    this.um = um;
}

public boolean canUserLogIn( String username, 
    String password ) {
    User user = um.getUser( username );
    return canUserLogIn( user, password );
}

public boolean canUserLogIn( User user, 
    String password ) {
    if (user == null)
        return false;
    return user.validatePassword( password );
}

Test with mock User
public class MockUser implements User {

    ... 

    // Setup what to return when validatePassword is called
    public void setValidatePasswordResult( boolean result ) {
        expectedCalls++;
        this.returnResult = result;
    }

    // Mock implementation of validatePassword
    public boolean validatePassword( String password ) {
        actualCalls++;
        return returnResult;
    }

    public boolean verify() {
        return expectedCalls == actualCalls;
    }

    ... 
}
Using *Static* Mock User

```java
public void testCanUserLogin() {
    MockUser user = new MockUser();
    user.setValidatePasswordResult( true );

    boolean result = um.canUserLogin( user, "foobar" );

    assertTrue("Expected to validate using " + 
                "password \"foobar\"", result );
    assertTrue("Mock User not used as expected", user.verify());
}
```
Static Mock Issues

- Simple, common
  - easy to generate (use tools, else tedious)
  - type safety is nice
- But, ...
  - slows compile/build
  - brittle, because test sees “inside” code
  - risk of incomplete emulation
public void testCanUserLogin() {
    com.mockobjects.dynamic.Mock mockUser =
            new Mock( User.class );
    mock.expectAndReturn( "validatePassword",
            C.args( C.eq( "foobar" ) ), true );

    boolean result = canUserLogin( (User) mockUser.proxy(), "foobar" );

    assertTrue("Expected to validate using " +
            "password \"foobbar\"",
            result );
    assertTrue("Mock User not used as expected",
            MockUser.verify());
}
Dynamic Mock Issues

- Faster
  - no preprocessing, no static files
  - no hand generation
  - not re-creation each build

- But, …
  - only works to mock interfaces (in Java) and can be clumsy with return values
  - tests less readable, some setup longer
  - String-based dynamic mocks can lead to trouble when method names are refactored
Week ends Tuesday at class start

<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>1</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>BattleSnake</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td>15</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>45</td>
<td>45</td>
<td>45</td>
<td>45</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 the trend requires I plan to be in town
Performance

- Performance Metrics
- Performance Testing
- Performance Tuning (Strategy)
- Design-for-Performance
Performance Metrics

- Important characteristic varies by tier
  - User/realtime “presentation”: latency (1/10th sec)
  - Compute engine “business tier”: throughput, utilization (queuing analysis)
  - Data tier: ‘ility, throughput (concurrency analysis)
  - Network: latency, reliability
- Have a benchmark suite (“95% of user actions effective within 1 second”)
- Measurement is key and often surprising
- Measure in every important environment
Performance Measurement

- Can be data sensitive (do multiple tests)
- Context sensitive
  - HotSpot warm-up (needs >2 executions)
  - cache warm-up
  - equipment variation
  - other activities
- Is observation-sensitive
- With Java, new JVM Profiler Interface and JPDA interfaces help (many tools) ... timing, heap
Performance Tuning (Strategy)

- Check simple things first (different compilers, different VM, turn on optimizations)
- Expect order-of-magnitude speedups on first round with large app
- Mindset: expect to encounter resource limits ...  
  - CPU (speed, availability)  
  - memory  
  - I/O (disk, network)  
  - external resource (DB, lock contention, etc)
Two Laws of Optimization
(Michael Jackson)

- The First Rule of Program Optimization
  Don’t do it.

- The Second Rule of Program Optimization
  --- FOR EXPERTS ONLY ---
  Don’t do it yet.
Performance Tuning Strategy

- Measure (profilers, benchmarks, instrumenting)
- Locate worst 3-5 bottlenecks
- Hypothesize causes; try to refute hypothesis
- Devise a test for factors identified by hypothesis
- Test hypothesis
- Alter app to reduce bottlenecks
- Test for improved performance
Performance by Design - Distributed

- Architecture & Topology often dominate

- Make it parallel? Amdahl’s Law ...
  - Speedup, Num proc, B is % serial \( S = \frac{N}{(B*N) + (1-B)} \)

- Tactics
  - get data and code together (partitioning)
  - reduce data sync costs by minimize duplication
    \( <- balance -> \) duplicate data to reduce data-transfer costs
  - caching to change distr req to local
  - msg reduction (latency); count usually matters more than size
    - compression (for large volumes)
    - batching (split into needed-now, anticipated data)
  - be asynchronous wherever possible; use queues to decouple processes
Performance by Design - Local

- Think about shared resources
  - garbage collection issues get huge on multiprocessor!
- 80/20 rule => profile, speed up the loops
  Think about the hardware support … JVM has fast compare for -1 .. 5
- Interface-based (allows plugin replacements)
  - Factory FooFactory#getFooInstance(), not new Foo()
  - object pooling
- Java: first 4 params fastest; use System.arraycopy for >10 elements; avoid and merge sync locks
- Java strings: interning; temp Strings; StringBuffer
Performance For Game

- Control your protocol- make simple delimiter, fixed format, simply encoded data; create your own tokenizer
  - historically, `getBytes()` from String most compute-intensive method on the class; lots of Unicode processing. If using `getBytes()`, contact me about ASCII assumptions for 7x speedup :)

- Queue-connect GUI, engine, network (three threads); various reader-writer queues avail - I like Doug Lea’s Concurrent utils

What’s Your Protocol Design?

- Astrix

- BattleSnake

- Caesar’s
Presentation Expectations

- 2/26 (one week)
- 15 minutes each
- Status report, not demo
  - General functional condition of app
  - Physical modularization (package structure, dependencies, deployment units)
  - How are you doing against your time estimates?
  - How are you doing against your risk estimates?
  - Prognosis?
- Start at eight, don’t be late - it’s as rude for us to enter late as for next class to enter early :)