Class #11 (2004/02/12)

- Changes/Review
- Exceptions

Lo Fi Prototyping Experience Report
- Assertions
- Expectations: ATP, S.I
Changes & Lecture 11 Takeaway

- **Changes/Notices/Comments**
  - extra credit policy (6 lectures, papers [non-Word] … 1/2 the points)
  - inspections lab Tue 1-2

- **Takeaway lecture 11 - Package Architecture**
  - Measure and Manage Dependency
  - UML is evolving; interaction frames may be a good thing
  - Metrics best-known at intramodule level (McCabe, Halstead); use GQM to select correct metrics

- **RAS - see also new WiKi page**
  - *Reliability* is the chance a system will operate for some period of time, and drives MTBF.
  - *Serviceability* is time needed to correct faults, and drives MTTR.
  - *Availability* is the proportion of time the system will operate to spec, and is a function of MTBF and MTTR.
Modeling Discussion

- Cost of detail modeling
- Stop modeling when no choices left for next developer (coder)
- Sequence diagram activation boxes, callbacks
Exceptions and Assertions

Sometimes it breaks ...
Why Make an Exception?

- Traditional - error codes returned up control stack
  - complex code (if-then tests)
  - control coupling
  - some operations don’t return values (ctor)

- Exception model
  - clean separation of error/normal code blocks
  - speedup normal path

- Risks
  - mem leaks, bad refs … this is “implicit goto”
  - complicated across process boundaries (tiers)
  - complicates APIs (painful to add later)
Java Exception Model

- **Exception** is a class; create an instance and throw() it (percolates up the call stack seeking a handler)

- `try/catch/finally` statement
  ```java
  try{s1; s2; ...}
  catch(ExceptionA eA){ seA1; seA2; ... }
  catch(ExceptionB eB){ seB1; seB2; ... }
  finally {sf1; sf2;... }
  ```

- if an exception thrown by s1, s2, ... then any matching catch block is entered, right away (skip remaining s3...)

- “matching” based on first **isa** (type match)

- whether an exception was thrown or not, the **finally** block *always* executes before method exits (except infinite loop in `try`, or `System.exit()` in catch!)
Exceptions in Exceptions?

- **throw from catch?**
  - Sure!
  - It goes up the call stack, not down the listing

- **throw from finally?**
  - Generates “cannot complete normally” warning
  - It goes up the call stack; doesn’t re-examine the catch clauses
The Throwable Hierarchy

From http://www-csfy.cs.ualberta.ca/~c115/W04/references/Exceptions/classes.html
Exceptions: checked or unchecked?

- Use checked for recoverable conditions
  - if caller can be expected to recover, then force them to consider doing so
- Use run-time for programming errors
  - usually precondition violations
  - you extend RuntimeException, leave Error to the JVM
- Generally, don’t catch Error or RuntimeException except to log … the condition isn’t recoverable
Inside An Exception

- Carry stack frame array (method names, line#; not params);
  - captured at exception ctor
  - exception object creation & stack clone is costly!
- Carries a “message” string for humans `e.getMessage()`
- Access to “what happened?”
  - `e.printStackTrace()` to dump data for humans
  - `e.getStackTrace()` gets array for program use (1.4)
Nested/mapped Exceptions

- Common practice:

```
try {     ... } catch(YourException e) {
throw new MyException();}
```

- used inter-tier (e.g., map all DB faults to StoreException) as an Adapter strategy
- loses cause (YourException e)
- ad hoc responses to nesting problem

- New “Chained” facility (in J2SE 1.4)
  - Throwable getCause(), initCause(Throwable)

- RMI Exceptions now carry server stack to client
Exception Rules!

- Single base class, so often just one catch block
- Create as few Exception types as possible
  - use invariants, assertions, system exceptions
- Every thread has `try/catch` at outermost level
- "catch" block - fix or re-throw (no return unfixed)
  - exception :) at outermost block, might just report to user and return to continue
- Before throw, restore state of the object to valid
- Don’t return from `try` block (confusing, because finally still runs!)
Exception Idioms

- Place try/catch around loops, not inside
  - about 21% slower in bare loop if no JIT
- Throw exceptions from constructors
  - gets around “no return” problem
  - avoid “two stage construction”

```java
class Foo {
    public Foo (String filename)
        throws
            FileNotFoundException,
            IOException {
        FileReader fr =
            new FileReader(fileName);
        }
    public void aMethod() { . . . }
}

class Test {
    public static void main (...){
        Foo myFoo = null;
        try { myFoo = new Foo("temp.txt"); }
        catch(FileNotFoundException fnfe)
        { . . . }
        catch (IOException ioe)
        { . . . }
        myFoo.aMethod(); . . .
    }
}
Exception Idioms -2

- Use for *exceptional* conditions

```java
int data;
MyInputStream in = ...
data = in.getData();
while (data != 0) {
    . . . // use data
data = in.getData();
}
```

VS

```java
int data;
MyInputStream in = ...
while (true) {
    try {data = in.getData();}
    catch (NoMoreDataException e1) {break;}
    . . . // use data
}
```

- Not for control flow!

- Don’t hide Exceptions … *very* confusing!
  - direct suppression
  - nested try? Each thrown exception replaces … only “most recent” goes up to caller
W/E 17 Feb:
A: 3 Astrix for paper proto

W/E 10 Feb (revised, late email from Aaron):
W: 2 Astrix for paper proto photos
W: 3 Caesar’s for Ed’s UML exploration
W: 1 Snake for Greg’s suggestions
A: 5 Snake for Brian’s prototyping effort
A: 2 Snake for pestering re soln posting

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>3</td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>BattleSnake</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>Caesar's</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>CLASS TOT</td>
<td>9</td>
<td>17</td>
<td>30</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
</tbody>
</table>

Current week:
Instructor awarded 0/6
TA awarded 3/9
Averaging 10/wk * 7 weeks = 70 BP … need 80 BP

**NO FREE PIZZA AT THIS RATE**

*but I’m starting to budget for it...*
An Experience with Lo Fi Prototyping

Team Unknown
assert-iveness in Java

- Assertions available in various language: runtime test that generates some kind of error
- ‘assert’ added in J2SE 1.4 (not in J2ME, sorry)
- assert facility uses AssertionError to communicate
- Two forms of assert statement
  - assert expression ; // no detail message
  - assert expression : expression2 ;
    - expression2 must return a value (not void)
    - expression2 value used as detail message in ctor
Assertions (in Java)

- Specifically for debugging (compact, easy to turn off)

If-then-else style:
```java
if (i % 3 != 0) {
    if (i < 0) {
        System.err.println("Error in Variable i");
        return -1;
    }
}
System.out.println("Change $"+i%3);
```

Assertion style:
```java
if (i % 3 != 0) {
    assert i > 0 : "Error in Variable i";
    System.out.println("Change $"+i%3);
}
```
Asserting an Internal Invariant

- Use an assertion whenever you would have written a comment that asserts an invariant

```java
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // We know (i % 3 == 2)
    ...
}
```

```java
switch(suit) {
    case Suit.CLUBS:
        ...
        break;
    case Suit.DIAMONDS:
        ...
        break;
    case Suit.HEARTS:
        ...
        break;
    case Suit.SPADES:
        ...
    default:
        assert false: suit; // or, better
        throw new AssertionError(suit);
}
```

throw new AssertionError better because protection even if assert disabled (but no extra runtime cost) and legal when enclosing method returns a value and every case returns value; then first alternative wouldn’t compile
Asserting Control (-Flow Invariants)

- Place an assertion at any location you assume will not be reached

```java
void foo() {
    for (...) {
        if (...) {
            return;
        }
    }
    assert false; // Execution should never reach this point!
}
```

- If truly “unreachable” then compile-time error if you try to assert that it is not reached. Again, an acceptable alternative is simply to throw an `AssertionError`
Assert the Contract

- Parameter checks on non-public methods
  - Don’t use assert on public method preconditions, use Exceptions … usually IllegalArgument Exception
  - Use assert on private/helper methods, don’t bother callers

```java
private void setRefreshInterval(int interval) {
    // Confirm adherence to precondition in nonpublic method
    assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;
    ... // Set the refresh interval
}
```

- assert lock status (nice trick)

```java
private Object[] a;

public synchronized int find(Object key) {
    return find(key, a, 0, a.length);
}

// Recursive helper method – always called with a lock on this object
private int find(Object key, Object[] arr, int start, int len) {
    assert Thread.holdsLock(this); // lock-status assertion
    ... // Recursive body
}
```
**Assert the Contract - 2**

- You can test postcondition with assertions in both public and nonpublic methods

```java
/**
 * Returns a BigInteger whose value is (this-1 mod m).
 *
 * @param  m the modulus.
 * @return this-1 mod m.
 * @throws ArithmeticException  m <= 0, or this BigInteger
 * has no multiplicative inverse mod m (that is, this BigInteger
 * is not relatively prime to m).
 */

public BigInteger modInverse(BigInteger m) {  
    if (m.signum <= 0) 
        throw new ArithmeticException("Modulus not positive: " + m);

    ... // Do the computation

    assert this.multiply(result).mod(m).equals(ONE) : this;
    return result;
}
```
**assert idioms**

- **To ensure assert enabled:**

  ```java
  static {
  boolean isAssertEnabled = false;
  assert isAssertEnabled = true;
  if (!isAssertEnabled)
      throw new RuntimeException("Assertion must be enabled!");
  }
  ```

- **Don’t use assert with required methods**

  - **NO**  
    
    ```java
    assert names.remove(null);
    ```

  - **YES**  
    
    ```java
    boolean nullsRemoved = names.remove(null);
    assert nullsRemoved;
    ```
assert in small devices

- Not supported by spec in J2ME/CLDC (is implementation-dependent)
Software Inspection Expectations

- Aaron will discuss details of “how”
- 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
  - 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)
ALL DONE FOR TODAY