CMSC433, Fall 2007
Programming Language Technologies and Paradigms

Testing

Bill Pugh
September 4th, 2007
Readings for today

• JUnit
• How web servers work
• Regular expressions
  – meaning of regular expressions (e.g., [0-9]+xx[0-9]+)
  – package java.util.regex.*
  – capturing groups
    • e.g., Pattern.compile("([a-z]+)([0-9]+)")
• TCP Sockets and server sockets
Things you should have completed

• The survey
• Getting Eclipse set up
  – plugins (course project manager, Clover)
  – linuxlab cvs repository
• Checking out project 0
• If you submitted project 0, perform a release test
  – several students submitted the project, haven’t performed a release test, and have errors
A student needs a note taker

• A student needs someone to share notes for the course
  – Disability support services will pay you to share your notes. Real money ($).

• See me if you are interested (today if at all possible)
  – Let me look at and copy your notes from today.
• Jon Bentley says that he has asked hundreds of professional programmers to implement binary search
  – given them up to two hours to do so
  – only 10% implemented it correctly
• Donald Knuth noted that while the first binary search algorithm was published in 1946
  – the first correct implementation wasn’t published until 1958
• Testing is almost never complete/exhaustive
• But it can be obviously incomplete
  – like to avoid that
• Want to run tests automatically
• Want to identify regressions as soon as possible
  – a regression is when a test that had been working starts failing
• Lots of small tests are good
  – better than one big monster test
  – when the monster test fails, you have lots of debugging to do
Test cases

• Basic functionality
  – simplest possible test cases

• Corner/Edge/Boundary cases
  – take a knob in the specification, and turn it all the way

• Everything mentioned in the spec

• Try to break the code
  – within the limits of the spec
Code coverage

• Tools exist to help you measure code coverage:
  – the portions of code executed

• Any code that isn’t executed by any test case hasn’t been tested
Several kinds of code coverage

- **Statement coverage**
  - is a statement executed?

- **Branch coverage**
  - is each branch in the program explored?

- **Boolean coverage**
  - does each boolean expression evaluate to both true and false?

- **Path coverage**
  - Is each (feasible) path through a method covered?
    - not all paths are feasible

- **Data coverage**
Why isn’t statement coverage enough?

```java
int f(Object x) {
    int result = 5;
    if (x != null)
        result += x.hashCode();
    return result*x.hashCode();
}
```
Boolean coverage

• How many test cases should we have for:
  \[
  \text{if } (a > 0 \ \&\& \ b > 0) \ \text{foo}();
  \]

• How about
  – a = 1, b = 1
  – a = -1, b = -1

• Should we also test the case where b > 0 evaluates to false?
  – a = 1, b = -1
Path coverage

• How many test cases should we have for:

\[
\text{if (a > 0) foo();}
\]
\[
\text{if (b > 0) bar();}
\]

• We need 4 cases to cover all paths
Infeasible paths

- How many test cases should we have for:
  
  \[
  \text{if } (a > 0) \ foo(); \\
  \text{if } (a > 5) \ bar(); \\
  \]

- One of the paths isn’t feasible
Code coverage is nice

• Can be automatically computed
• Can spot coverage regressions
  – did you commit a change that reduced code coverage?
    • perhaps you introduced new code without a test case
    • perhaps you changed how execution flows so that preexisting code is no longer executed
• Managers can measure it
  – Good managers will understand it and use it well
  – Some managers insist on high or perfect code coverage
Code coverage isn’t enough

• Test cases that obtain code coverage but don’t check the results sort of miss the point
  – still useful

• A lot of times you need data coverage
  – did you try duplicated elements?
  – did you try negative values in the array?
• Mutation testing
  – Modifies your code
  – checks to see which code changes don’t result in test cases failing
  – may indicate lack of coverage

• http://jester.sourceforge.net/
Writing tests isn’t easy

- Some experience/evidence to suggest that to really test your code, you will need to write more lines of test code than lines of code being tested

- *We found over the years that the ratio of code to test code very rarely changes: Typically for every line of Java, it takes 3 to 5 lines of JUnit to achieve 80-90 percent code coverage. If you have 250,000 lines of Java code, you probably need about one million lines of JUnit. Writing tests is hard for anybody, and writing tests for legacy code, especially that many tests, is no fun. You are looking at many engineering years of effort.*

  – Alberto Savoia, Agitar software

• Automated generation of JUnit characterization tests
  – doesn’t know what your code is supposed to do
• Tries to generate test cases that achieve full coverage (statement or branch?)
• Asserts that the code does whatever it does at generation time
  – may not be correct
  – check test cases to see if the asserted behavior is correct
  – use tests to check for changes in behavior
    • for good or bad
Writing tests isn’t easy

- You may need to rewrite/refactor your code to make it easy to test your code
  - pull out logic so that it is easier to test
  - create mock objects
    - dummy objects that the code under test will interoperate with
  - allow for failure injection
    - how can you test how your code behaves when the network fails unless you can force the network to fail on demand
An amusing story, and a few lessons

• A standard part of binary search implementations:

  \[
  \text{int \ mid = (low + high) / 2;}
  \]

• What is wrong with this code?
This code is bad

- This had been used in Sun’s JDK binary search implementation for many years
- What could go wrong?
- What is the array is very large?
  - more than $2^{30}$ elements

- What if $(\text{low} + \text{high})$ overflows 32-bit signed arithmetic?
  - then you get a negative number
  - which remains negative when you divide by 2
Several ways to fix it

- Easiest fix: use unsigned right shift

```java
int mid = (low + high) >>> 1;
```

- Shift result one bit to the right, filling in with a zero in the high order bit
  - result is always nonnegative
  - pretty sure it gives us what we want

- How do we test it?
Testing it

• This won’t be tested for project 0, but...
• We could create a byte array containing more than $2^{30}$ elements
  – might even be able to test that on my laptop
  – harder with an int array

• But it would be better to break out the functionality

```java
static int computeMidpoint(int low, int high) {
    return (low+high) >>> 1;
}
```
Advantages

• We can test the midpoint finding functionality without testing binary search
  – or supplying a large array
• Clear to people who read the code
  – many developers may not be familiar with >>> operator
Disadvantages

• None

• No performance hit for pulling code out into a separate method
  – a good JVM will inline the method call, give same performance as original code