Testing

John Jannotti

/course/cs0320/www/lectures/

Feb 7, 2017
Course Announcements

- This week is the “webby” lab. See how we’re doing UIs.
- It will cover mechanical details this lecture skips.
Testing Principles

- **Write Tests First (Test Driven Design)**
  - Forces you to think like the caller.
  - Forces you to develop “testable” interfaces.
  - Satisfaction as tests start passing.

- **Write Tests Last**
  - Based on your knowledge of implementation.
  - Can you get someone else to write some?
  - Every time you find a bug, write a test.
A fun story of a subtle multi-decade bug

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];
        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found.
}
```

http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
Consider “surprising” input.

Empty collections, nulls, 0, negatives, MAX_VALUE, MIN_VALUE.

Empty strings, “international” characters, “weird” characters.
  ▶ Quoting can be really tricky, yet critical for security.
  ▶ Avoid doing it “yourself.” Use well-tested library routines.

Duplicate arguments. What does list.addAll(list) do?
  ▶ Remember to think of the object’s state as an argument.

Missing files, empty files, BIG files (2GB & 4GB can be important)

Consider integer overflow and floating point (im)precision.
Let the computer do the work

- Input “fuzz” — tests generated at random.
  - Simple utilities — truly random data.
  - More useful — random perturbations of “good” data.
  - Most useful — structured perturbations of good data.

- Exhaustive tests
  - Computers are fast, and some tests could run “offline” anyway.
  - https://randomascii.wordpress.com/2014/01/27/theres-only-four-billion-floatsso-test-them-all/

- ...perhaps against a “known good” implementation.
  - Getting any ideas for testing your KdTrees?
Last slide’s blog post exhaustively tested a new ceil() function on all 32-bit floats. (You remember floats, right? It’s a keyword.) How long did it take to conduct the exhaustive test, by comparing the result of the standard ceil() function to the new, faster ceil function?

A) 200 milliseconds
B) a minute and a half
C) 15 minutes
D) 2.5 hours
E) just under a day

PS. It was wrong a third of the time!
Black box testing

- Black box tests adherence to a spec.
  - `assertEquals(min(2,3), 2);`
  - `assertEquals(min(3,2), 2);`
  - `assertEquals(min(-1,3), -1);`

- These kind of tests *could* be written by a QA team.
  - On the one hand, nice to have a different brain write tests.
  - On the other, that can be a slow process, filled with finger-pointing.
Whitebox testing considers the implementation (but still tests the spec).

- Considering explicit boundary conditions of the code.
  - Choosing a median value? Try odd and even sized arrays.
  - Selecting a min/max internally? Try $a > b$, $b < a$, ties.
  - Alternate implementations based on size? Transitions?
  - Growing your hash table at known size intervals?

- Take care! You should still be testing the spec, not for a particular implementation.
Coverage

You are usually looking for high “coverage” — a metric for how well your tests exercise your code.

- Consider each branch
- Construct input to exercise code
- Code coverage tools can help.
- Statements (Lines), Branches, Paths
- Coverage Tools exist, we’ll look into Maven integration.
- I’m told EclEmma is nice, for Eclipse. http://www.eclemma.org/
- You can run it, look for uncovered code, then devise a new test.
Coverage

```c
int calc(int a, int b, int c) {
    if (a < b) {
        b += 2;
    } else {
        if (b == c)
            a *= 2;
        return a - b;
    }
    return a + b;
}
```

Which claim is true?

A) calc(1,2,3), calc(2,1,0) provides statement coverage.

B) calc(1,2,3), calc(2,0,0) provides branch coverage.

C) calc(1,2,3), calc(2,0,1), calc(4,4,4) provides path coverage.

D) Path coverage is possible with two calculate() calls.

E) Path coverage requires at least four calculate calls.
Unit, System, Integration Testing

- Scale of the test
- Unit test should be self-contained
  - Are you separating CSV lines into tokens properly?
  - Can your KdTree find members properly?
  - Does your command loop process input properly?
  - At this level, avoid testing multiple modules. This can be tricky.
- Integration tests a few components
  - Is a star file being put into KDTree properly?
  - You can still use the JUnit framework for these.
- System tests are “end-to-end”
  - Does a star file plus query yield the proper output?
  - We’re giving you a start with our system test framework.
JUnit is a de facto standard in the Java world.

Similar libraries for other languages. They’re not rocket science. Build your own if you need.

Helpful framework for testing:
- You write small `testSomething()` methods
- JUnit runs them, compiles a report

Test should be small and self-contained:
- When tests fail, you should be able to pinpoint blame.
- Avoid cross-module calls.
- “Leave no trace” (try...finally, `setUp()`, `tearDown()`)
- How can you avoid infrastructure classes?
- How can you get good coverage of error handling?

It’s great for them to run at compile time.
Sidebar: try/finally

- try is not just for catching exceptions.
- In combination with finally allows for guaranteed cleanup.

```java
void executeWithFile(String filename, Action e) throws IOException {
    InputStream stream = new FileInputStream(filename);
    try {
        e.execute(stream); // might throw an exception
    } finally {
        stream.close();
    }
}
```
Java 7 introduced syntax for a common pattern.

```java
void executeWithFile(String filename, Action e) throws IOException {
    try (InputStream stream = new FileInputStream(filename)) {
        e.execute(stream);  // might throw an exception
    }
}
```
Consider a Thermostat class

```java
public class Thermostat {
    private double intention = 69.0;
    private double slack = 2.0;

    private Thermometer thermometer;
    private Device heater;
    private Device cooler;

    public void act() {
        double current = thermometer.getTemp();
        if (current > intention + slack) {
            heater.turnOff();
            cooler.turnOn();
        } else if (current < intention - slack) {
            heater.turnOn();
            cooler.turnOff();
        }
    }
}
```
How can you test classes in isolation?

```java
public class Thermostat {
  public Thermostat(double intention) {
    this.intention = intention;
    this.thermometer = new Thermometer(); // Speaks USB
  }

  public static void main(String[] args) {
    Thermostat stat = new Thermostat(69.5);
    while (true)
      stat.act();
  }
}
```
Fakes, Stubs, and Mocks, Oh My!

Fake objects may be used to stand in for your real, complex objects when unit-testing another component.

```java
interface Thermometer {
    public double getTemp();
}

class UsbThermometer implements Thermometer {
    public double getTemp() {
        // Complex USB specific device interaction
    }
}

class StubThermometer implements Thermometer {
    public double getTemp() { return 68.5; }
}
“Wider” constructors allow for flexibility

```java
class Thermostat {
    public Thermostat(double intention, Thermometer thermometer) {
        this.intention = intention;
        this.thermometer = thermometer;
    }

    public static void main(String[] args) {
        Thermostat stat =
            new Thermostat(69.5, new USBThermometer());
        while (true)
            stat.act();
    }
}
```

This line of thinking leads to “Dependency Injection” or “Inversion of Control” frameworks.
Mocks allow more comprehensive tests

```java
class MockThermometer implements Thermometer {
    public double temp;
    public boolean throwError;

    public double getTemp() {
        if (throwError)
            throw new IllegalStateException("No device");
        return temp;
    }

    public void setTemp(t) {
        temp = t;
    }
}
```

Now, the unit test can preset the temperature in the mock, or even ask for error conditions.
Develop some simple unit tests

```
Multimap<String, Integer> mm = HashMultimap.create();
assertEquals(mm.get("key1").size(), 0);
assertTrue(mm.isEmpty());

mm.put("key1", 5);
assertFalse(mm.isEmpty());
assertEquals(mm.get("key1").size(), 1);
assertEquals(mm.get("key2").size(), 0);
assertTrue(mm.get("key1").contains(5));

mm.get("key1").add(15);
mm.get("key2").add(100);
assertEquals(mm.get("key1").size(), 2);
assertTrue(mm.get("key1").contains(15));
assertTrue(mm.containsKeyEntry("key2", 100));
assertTrue(mm.containsValue(5));
```
```java
int hash = mm.hashCode();

Collection<Integer> k1 = mm.get("key1");
assertEquals(k1.size(), 2);
mm.get("key1").clear();
assertEquals(mm.size(), 1);
assertEquals(k1.size(), 0);

mm.put("key1", 15);
mm.put("key1", 5);

assertEquals(hash, mm.hashCode()); // Legit?
```
public static <K,V>

void multimapOk(Multimap<K,V> mmap) {
    Collection<K> keys = mmap.keys();
    int size = mmap.size();
    int sum = 0;
    for (K key : keys) {
        assertFalse(mmap.get(key).isEmpty());
        sum += mmap.get(key).size();
    }
    assertEquals(size, sum);
}
Consider adding \texttt{invariant()} to mutable classes.

\begin{verbatim}
private boolean invariant() {
    Collection&lt;K&gt; keys = keys();
    int size = size();
    int sum = 0;
    for (K key : keys) {
        if (get(key).isEmpty()) return false;
        sum += mmap.get(key).size();
    }
    return size == sum;
}
\end{verbatim}

When should \texttt{invariant()} be true?
Use `assert` to test `invariant()`

```java
public void put(K key, V value) {
    assert invariant();  
    /* the real work */
    assert invariant();
}
```

When running your code, pass `-ea` to `java`. 
Sidebar: what about non-void methods?

```java
1 public boolean put(K key, V value) {
2     assert invariant();
3     /* the real work */
4     return existed;
5     assert invariant(); // Dead code!
6 }
```
We will learn about some tools that simplify this process.
Testing larger functionality

- Unit tests only go so far.
- System tests cover end-to-end functionality
  - How do you know your program works?
  - And how do you know it still works after modifying?
  - Regression tests avoid improvement paralysis.
- Integration tests will matter a lot for group projects.
  - Test that code from different people work together.
  - Yet don’t jump right to complete system tests.
  - Stubs and Mocks will be necessary.
  - Advice: once again, code to interfaces.
  - And I can’t stress this enough, use git.