CSE 373: Data Structures & Algorithms
Software Interlude -- Testing and JUnit

Riley Porter
Winter 2017

based on work from Michael Ernst, Hal Perkins, Dan Grossman, and Zack Tatlock
Course Logistics

• HW5 out → more graphs!

• Nearing the end! The last main course topic is next week: sorting. HW6 out next Wednesday and due March 10th
Software Quality (QA or QE)

It’s a CS research area and can be a full time job! Some activities include:
• Static analysis (assessing code without executing it)
• Correctness proofs (theorems about program properties)
• Code reviews (people reading each others’ code)
• Software process (methodology for code development)
• Testing (of course)

**Testing is NOT just debugging!**

We’ll cover lots of testing principles and strategies:
- Heuristics for good test suites
- Black-box testing
- Clear-box testing and coverage metrics
- Regression testing
- Integration/System tests
- Test Driven Development
Testing is so important the field has terminology for different kinds of tests

- Won’t discuss all the kinds and terms

Here are three different dimensions:

- **Unit** testing versus **system/integration** testing
  - One module’s functionality versus pieces fitting together

- **Black-box** testing versus **clear-box** testing
  - Does implementation influence test creation?
  - “Do you look at the code when choosing test data?”

- **Specification** testing versus **implementation** testing
  - Test only behavior guaranteed by specification or other behavior expected for the implementation?
Unit Testing

• A unit test focuses on one method, class, interface, or module

• Test a single unit in isolation from all others

• Typically done earlier in software life-cycle
  – Integrate (and test the integration) after successful unit testing

• Common Java unit testing framework: JUnit
Square Root Example

// throws: IllegalArgumentException if x<0
// returns: approximation to square root of x
public double sqrt(double x){...}

What are some values or ranges of x that might be worth probing?

- $x < 0$ (exception thrown)
- $x \geq 0$ (returns normally)
- around $x = 0$ (boundary condition)
- perfect squares ($\sqrt{x}$ an integer), non-perfect squares $x<\sqrt{x}$ and $x>\sqrt{x}$ – that's $x<1$ and $x>1$ (and $x=1$)

Specific tests: say $x = -1, 0, 0.5, 1, 4$
General Approach: Partition the Input Space

Ideal test suite in theory:
(1) Identify sets of input where all the members have the same behavior.
(2) Try one input from each set.

Two problems with execution:

1. Notion of same behavior is subtle
   - Naive approach: execution equivalence
   - Better approach: revealing subdomains

2. Discovering the sets requires perfect domain knowledge
   - If we had it, we wouldn’t need to test
   - Use heuristics to approximate cheaply
Test Suite Example #1

```c
// returns: x < 0 ⇒ returns -x
// otherwise ⇒ returns x

int abs(int x):
    if (x < 0):
        return -x;
    else:
        return x;
```

All $x < 0$ are execution equivalent
   – Program takes same sequence of steps for any $x < 0$

All $x \geq 0$ are execution equivalent

So $\{-3, 3\}$ is probably a good test suite (one element from each subset)
Test Suite Example #2

// returns:  x < 0 ⇒ returns -x
// otherwise ⇒ returns x

int abs(int x):
    if (x < -2):
        return -x;
    else:
        return x;

For this (buggy) implementation of the method, three possible outcomes:
- x < -2 PASS
- x = -2 or x = -1 FAIL
- x ≥ 0 PASS

{-3, 3} as a test suite does not reveal the error!
Determining Actual Subsets

• A subdomain is a subset of possible inputs

• A subdomain is revealing for error E if either:
  – Every input in that subdomain triggers error E, or
  – No input in that subdomain triggers error E

• Need test only one input from a given subdomain
  – If subdomains cover the entire input space, we are guaranteed to detect the error if it is present

• The trick is to guess these revealing subdomains
Heuristic: Boundary Testing

Create tests at the edges of subdomains

Why?

- Off-by-one bugs
- “Empty” cases (0 elements, null, …)
- Overflow errors in arithmetic
- Object aliasing

Small subdomains at the edges of the “main” subdomains have a high probability of revealing many common errors

- Also, you might have misdrawn the boundaries
Boundary Testing

To define the boundary, need a notion of adjacent inputs

One approach:
  – Identify basic operations on input points
  – Two points are adjacent if one basic operation apart

Point is on a boundary if either:
  – There exists an adjacent point in a different subdomain
  – Some basic operation cannot be applied to the point

Example: list of integers
  – Basic operations: create, append, remove
  – Adjacent points: <[2,3],[2,3,3]>, <[2,3],[2]>
  – Boundary point: [ ] (can’t apply remove)
Some Boundary Cases

**Arithmetic**
- Smallest/largest values (edge case and overflow)
- Zero

**Objects**
- null
- Circular list
- Same object passed as multiple arguments (aliasing)
Boundary: Arithmetic Overflow

// returns: |x|
public int abs(int x) {...}

What are some values or ranges of x that might be worth probing?
- x < 0 (flips sign) or x ≥ 0 (returns unchanged)
- Around x = 0 (boundary condition)
- Specific tests: say x = -1, 0, 1

How about...

```java
int x = Integer.MIN_VALUE; // x=-2147483648
System.out.println(x<0); // true
System.out.println(Math.abs(x)<0); // also true!
```

From Javadoc for Math.abs:
- Note that if the argument is equal to the value of
  `Integer.MIN_VALUE`, the most negative representable int value, the
  result is that same value, which is negative
Boundary: Duplicates and Aliases

// modifies: src, dest
// effects: removes all elements of src and
//          appends them in reverse order to
//          the end of dest
<E> void appendList(List<E> src, List<E> dest) {
    while (src.size()>0) {
        E elt = src.remove(src.size()-1);
        dest.add(elt);
    }
}

What happens if src and dest refer to the same object?
– This is aliasing
– It’s easy to forget!
– Watch out for shared references in inputs
Black-Box Testing

Heuristic: Explore alternate cases in the specification, plus potentially some boundary conditions around those cases

Procedure is a black box: interface visible, internals hidden

Example

```c
// returns: a > b ⇒ returns a
// a < b ⇒ returns b
// a = b ⇒ returns a

int max(int a, int b) {...}
```

3 cases the client knows about leads to 3 tests:

- (4, 3) ⇒ 4  (i.e. any input in the subdomain \(a > b\))
- (3, 4) ⇒ 4  (i.e. any input in the subdomain \(a < b\))
- (3, 3) ⇒ 3  (i.e. any input in the subdomain \(a = b\))
Black-Box Testing: Advantages

Process is not influenced by component being tested
- Assumptions embodied in code not propagated to test data
- (Avoids “group-think” of making the same mistake)

Robust with respect to changes in implementation
- Test data need not be changed when code is changed

Allows for independent testers
- Testers need not be familiar with code
- Tests can be developed before the code
Clear (or white or class) Box Testing

Heuristic: Test the actual implementation (look at the code)

**Focus:** features not described by specification
- Control-flow details
- Performance optimizations
- Alternate algorithms for different cases

Common **goal:**
- Ensure test suite covers (executes) all of the program
- Measure quality of test suite with % *coverage*

**Assumption** implicit in goal:
- If high coverage, then most mistakes discovered
Clear-Box Testing: Motivation

What are some subdomains that black-box testing won't catch:

```java
boolean[] primeTable = new boolean[CACHE_SIZE];

boolean isPrime(int x) {
    if (x > CACHE_SIZE) {
        for (int i = 2; i < x / 2; i++) {
            if (x % i == 0)
                return false;
        }
        return true;
    } else {
        return primeTable[x];
    }
}
```
Clear-Box Testing

• Finds an important class of boundaries -- ones not necessarily easy to guess given the specification
  – Yields useful test cases

• Consider CACHE_SIZE in isPrime example
  – Important tests \texttt{CACHE\_SIZE-1}, \texttt{CACHE\_SIZE}, \texttt{CACHE\_SIZE+1}
  – If \texttt{CACHE\_SIZE} is mutable, may need to test with different \texttt{CACHE\_SIZE} values

Disadvantage:
  – Tests may have same bugs as implementation
  – Buggy code tricks you into complacency once you look at it
Code Coverage Example #1

What is enough testing? What cases? Does this code have a bug?

```c
int min(int a, int b) {
    int r = a;  // should be r = b
    if (a <= b) {
        r = a;
    }
    return r;
}
```

• Consider any test with \(a \leq b\) (e.g., \(\text{min}(1, 2)\))
  – Executes every instruction
  – Misses the bug

• *Statement coverage* is not enough
What is enough testing? What cases? Does this code have a bug?

```c
int num_pos(int[] a) {
  int ans = 0;
  for (int x : a) {
    if (x > 0)
      ans = 1; // should be ans += 1
  }
  return ans;
}
```

- Consider two-test suite: {0,0} and {1}. Misses the bug.
- Or consider one-test suite: {0,1,0}. Misses the bug.

- *Branch coverage* is not enough
  - Here, *path coverage* is enough, but *no bound* on path-count
Varieties of Coverage

Various coverage metrics (there are more):

- Statement coverage
- Branch coverage
- Loop coverage
- Condition/Decision coverage
- Path coverage

Limitations of coverage:

1. 100% coverage is not always a reasonable target
   100% may be unattainable (dead code)
   *High cost* to approach the limit
2. Coverage is *just a heuristic*
   We really want the revealing subdomains
Regression Testing

• Whenever you find a bug
  – Store the input that elicited that bug, plus the correct output
  – **Add these to the test suite**
  – Verify that the test suite fails
  – Fix the bug
  – Verify the fix

• Ensures that your fix solves the problem
  – Don’t add a test that succeeded to begin with!

• Helps to populate test suite with good tests

• Protects against reversions that reintroduce bug
  – It happened at least once, and it might happen again
System or Integration Testing

Tests of whether the system as a whole works — whether the (individually correct, unit-tested) modules fit together to achieve correct functionality

• All of the previous topics (black-box, clear-box, regression testing, determining test cases) still apply
• End-to-End tests will test your system from the users (front end) to the persistent data storage (back end)
• Usually involves more complicated operations than unit tests
General Rules of Testing

First rule of testing: *Do it early and do it often*
- Best to catch bugs soon, before they have a chance to hide
- Automate the process if you can
- Regression testing will save time

Second rule of testing: *Be systematic*
- If you randomly thrash, bugs will hide in the corner until later
- Writing tests is a good way to understand the spec
- Think about revealing domains and boundary cases
  - If the spec is confusing, write more tests
- Spec can be buggy too
  - Incorrect, incomplete, ambiguous, missing corner cases
- When you find a bug, write a test for it first and then fix it
Hints on Testing

• Write small tests
• Choose good names for your tests:
  – use the proper instance of the assert method
  – write good messages
• Think carefully whether alternative solutions should be correct
  – (e.g., is there more than one shortest path for the given graph?).
• Write targeted tests
  – not an arbitrary number of random examples
• Keep your unit tests de-coupled
  – don’t have one test case test multiple things
  – don’t rely on certain state in the middle of the test that is not related to the test case
Test Driven Development

Write your tests **before** starting to write any code.

**First:**

use the specification to identify the abstract-value domain of each non-trivial public method

- what is the set of objects that the method can be called on, and the set of allowed inputs?

**Then:**

when you actually implement the code, you’ll have thought about these cases, cleared up any confusion with the specification, and you are less likely to make mistakes.
JUnit: Testing Framework

• A Java library for unit testing, comes included with Eclipse
  – OR can be downloaded for free from the JUnit web site at http://junit.org
  – JUnit is distributed as a "JAR" which is a compressed archive containing Java .class files

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public void name() { // a test case method
        ...
    }
}
```

A method with @Test is flagged as a JUnit test case and run
JUnit Asserts and Exceptions

• A test will pass if the assert statements all pass and if no exception thrown. Examples of assert statements:
  - `assertTrue(message, value)`
  - `assertFalse(message, value)`
  - `assertEquals(message, expected, actual)`
  - `assertNull(message, value)`
  - `assertNotNull(message, value)`
  - `fail(message)`

• Tests can expect exceptions or timeouts
  ```java
  @Test(expected = ExceptionType.class)
  public void name() {
      ...
  }
  ```
Today’s Takeaways

• Understand some basic testing principles and strategies
  – Unit testing
  – Heuristics for good test suites
  – Black-box testing
  – Clear-box testing and coverage metrics
  – Regression testing
  – Integration/System tests
  – Test Driven Development

• Understand how to write some basic JUnit