CSE 331
Software Design & Implementation

Hal Perkins
Winter 2018
Testing
• HW4 due Thursday night
  – Gotta implement things as specified

• Watch the late days: some people have used 3 of 4 as of hw2, a couple have used all of them already.
  – Assignments are *not* accepted late except for the few late days (2 max per assignment, 4 *total* for the quarter). No sliding penalties, etc.
    • (Obviously serious emergencies can be a different story, but those are few and far between)
  – Turn in your best effort on time or when you are out of late days and we’ll award credit based on work done
• HW5 out by late Wed.; HW6 out shortly after that
  – HW5: design/implement/test a Graph ADT
    • Will take time – start early (we’re not kidding)
  – More in sections this week (don’t miss)
  – Do a preliminary design yourself (for sure have a first design by end of the weekend) then discuss ideas & tradeoffs with others (use whiteboards, etc.)
  – HW6: graph application. Good for insight on some of the things your Graph ADT needs to support
• Lots of readings for next few lectures – quizzes soon!
Outline

• Why correct software matters
  – Motivates testing and more than testing, but now seems like a fine time for the discussion

• Testing principles and strategies
  – Purpose of testing
  – Kinds of testing
  – Heuristics for good test suites
  – Black-box testing
  – Clear-box testing and coverage metrics
  – Regression testing
Non-outline

• Modern development ecosystems have much built-in support for testing
  – Unit-testing frameworks like JUnit
  – Regression-testing frameworks connected to builds and version control
  – Continuous testing
  – ...

• No tool details covered here
  – See homework, section, internships, …
Ariane 5 rocket (1996)

Rocket self-destructed 37 seconds after launch
  – Cost: over $1 billion

Reason: Undetected bug in control software
  – Conversion from 64-bit floating point to 16-bit signed integer caused an exception
  – The floating point number was larger than 32767
  – Efficiency considerations led to the disabling of the exception handler, so program crashed, so rocket crashed
Therac-25 radiation therapy machine

Excessive radiation killed patients (1985-87)

- New design removed hardware prevents the electron-beam from operating in its high-energy mode. Now safety checks done in software.

- Equipment control task did not properly synchronize with the operator interface task, so race conditions occurred if the operator changed the setup too quickly.

- Missed during testing because it took practice before operators worked quickly enough for the problem to occur.
Legs deployed → Sensor signal falsely indicated that the craft had touched down (130 feet above the surface)

Then the descent engines shut down prematurely

Error later traced to a single bad line of software code

Why didn’t they blame the sensor?
More examples

• Mariner I space probe (1962)
• **Microsoft Zune New Year’s Eve crash (2008)**
• iPhone alarm (2011)
• Denver Airport baggage-handling system (1994)
• Air-Traffic Control System in LA Airport (2004)
• AT&T network outage (1990)
• Northeast blackout (2003)
• USS Yorktown Incapacitated (1997)
• Intel Pentium floating point divide (1993)
• Excel: 65,535 displays as 100,000 (2007)
• Prius brakes and engine stalling (2005)
• Soviet gas pipeline (1982)
• **Study linking national debt to slow growth (2010)**
• …
Software bugs cost money

- 2013 Cambridge University study: Software bugs cost global economy $312 Billion per year
  - [http://www.prweb.com/releases/2013/1/prweb10298185.htm](http://www.prweb.com/releases/2013/1/prweb10298185.htm)

- $440 million loss by Knight Capital Group in 30 minutes
  - August 2012 high-frequency trading error

- $6 billion loss from 2003 blackout in NE USA & Canada
  - Software bug in alarm system in Ohio power control room
Building Quality Software

What Affects *Software Quality*?

**External**
- Correctness: Does it do what it supposed to do?
- Reliability: Does it do it accurately all the time?
- Efficiency: Does it do without excessive resources?
- Integrity: Is it secure?

**Internal**
- Portability: Can I use it under different conditions?
- Maintainability: Can I fix it?
- Flexibility: Can I change it or extend it or reuse it?

**Quality Assurance (QA)**
- Process of uncovering problems and improving software quality
- Testing is a major part of QA
Software Quality Assurance (QA)

Testing plus other activities including:

– 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)
– …and many other ways to find problems and increase confidence

No single activity or approach can guarantee software quality

“Beware of bugs in the above code;
I have only proved it correct, not tried it.”
-Donald Knuth, 1977
What can you learn from testing?

“Program testing can be used to show the presence of bugs, but never to show their absence!”

*Edsger Dijkstra*

*Notes on Structured Programming, 1970*

Nevertheless testing is essential. Why?
What Is Testing For?

Validation = reasoning + testing
  – Make sure module does what it is specified to do
  – Uncover problems, increase confidence

Two rules:

1. Do it **early** and **often**
   – Catch bugs quickly, before they have a chance to hide
   – **Automate** the process wherever feasible

2. Be **systematic**
   – If you thrash about randomly, the bugs will hide in the corner until you're gone
   – Understand what has been tested for and what has not
   – Have a strategy!
Kinds of testing

• Testing is so important the field has terminology for different kinds of tests
  – Won’t discuss all the kinds and terms

• Here are three orthogonal dimensions [so 8 varieties total]:
  – *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
How is testing done?

Write the test
1) Choose input data/configuration
2) Define the expected outcome

Run the test
3) Run with input and record the outcome
4) Compare observed outcome to expected outcome
sqrt example

```java
// 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 $x = -1, 0, 0.5, 1, 4$
What’s So Hard About Testing?

“Just try it and see if it works…”

```c
// requires: 1 \leq x,y,z \leq 10000
// returns: computes some f(x,y,z)
int proc1(int x, int y, int z){…}
```

Exhaustive testing would require 1 trillion runs!
- Sounds totally impractical – and this is a trivially small problem

Key problem: choosing test suite
- Small enough to finish in a useful amount of time
- Large enough to provide a useful amount of validation
Approach: Partition the Input Space

Ideal test suite:
  Identify sets with same behavior
  Try one input from each set

Two problems:

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

2. Discovering the sets requires perfect knowledge
   • If we had it, we wouldn’t need to test
   • Use heuristics to approximate cheaply
Naive Approach: Execution Equivalence

// 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 ≥ 0 are execution equivalent

Suggests that {-3, 3}, for example, is a good test suite
Execution Equivalence Can Be Wrong

// returns: x < 0 => returns -x
// otherwise => returns x

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

{-3, 3} does not reveal the error!

Two possible executions: x < -2 and x >= -2

Three possible behaviors:
  - x < -2 OK, x = -2 or x= -1 (BAD)
  - x >= 0 OK
Heuristic: Revealing Subdomains

- 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
For buggy `abs`, what are revealing subdomains?

- Value tested on is a good (clear-box) hint

```cpp
// returns: x < 0 => returns -x
// otherwise => returns x

int abs(int x) {
    if (x < -2) return -x;
    else return x;
}
```

Example sets of subdomains:

- Which is best?

```
... { -2} { -1} { 0} { 1} ... 
{ ..., -4, -3} {-2, -1} {0, 1, ...} 
```

Why not:

```
{ ..., -6, -5, -4} {-3, -2, -1} {0, 1, 2, ...} 
```
Heuristics for Designing Test Suites

A good heuristic gives:

- Few subdomains
- ∀ errors in some class of errors $E$: High probability that some subdomain is revealing for $E$ and triggers $E$

Different heuristics target different classes of errors

- In practice, combine multiple heuristics
- Really a way to think about and communicate your test choices
Black-Box Testing

Heuristic: Explore alternate cases in the specification

Procedure is a black box: interface visible, internals hidden

Example

```cpp
// returns:  a > b => returns a
//           a < b => returns b
//           a = b => returns a

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

3 cases lead 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
More Complex Example

Write tests based on cases in the specification

```java
// returns: the smallest i such
//          that a[i] == value
// throws: Missing if value is not in a
int find(int[] a, int value) throws Missing
```

Two obvious tests:

1. `( [4, 5, 6], 5 )` => 1
2. `( [4, 5, 6], 7 )` => throw Missing

Have we captured all the cases?

Must hunt for multiple cases

- Including scrutiny of effects and modifies
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)
Other Boundary Cases

Arithmetic
  – Smallest/largest values
  – Zero

Objects
  – null
  – Circular list
  – Same object passed as multiple arguments (aliasing)
Boundary Cases: 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 $x \geq 0$ (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 Cases: Duplicates & 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
Heuristic: Clear (glass, white)-box testing

*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
Glass-box Motivation

There 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];
    }
}
```
Glass Box Testing: [Dis]Advantages

- Finds an important class of boundaries
  - Yields useful test cases

- Consider `CACHE_SIZE` in `isPrime` example
  - Important tests `CACHE_SIZE-1, CACHE_SIZE, CACHE_SIZE+1`
  - If `CACHE_SIZE` is mutable, may need to test with different `CACHE_SIZE` values

Disadvantage:
- Tests may have same bugs as implementation
- Buggy code tricks you into complacency once you look at it
Code coverage: what is enough?

```c
int min(int a, int b) {
    int r = a;
    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
Code coverage: what is enough?

```c
int quadrant(int x, int y) {
    int ans;
    if(x >= 0)
        ans=1;
    else
        ans=2;
    if(y < 0)
        ans=4;
    return ans;
}
```

- Consider two-test suite: (2,-2) and (-2,2). Misses the bug.
- *Branch coverage* (all tests “go both ways”) is not enough
  - Here, *path coverage* is enough (there are 4 paths)
Code coverage: what is enough?

```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
Code coverage: what is enough?

```c
int sum_three(int a, int b, int c) {
    return a+b;
}
```

- **Path coverage** is not enough
  - Consider test suites where `c` is always 0

- Typically a “moot point” since path coverage is unattainable for realistic programs
  - But do not assume a tested path is correct
  - Even though it is more likely correct than an untested path

- Another example: buggy `abs` method from earlier in lecture
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
Pragmatics: How Many/What Tests?

• Ideal: each test checks one specific thing (method, …)
  – Failure points to responsible component

• Reality: can’t always test in complete isolation
  – Example: need to use observer(s) to see if creator, mutator, or producer yields correct result(s)

• Reality: try to structure test suites so each test checks one new thing and has minimal dependence on others
  – Failure more likely to point to a single responsible component
Pragmatics: 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
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
Closing thoughts on testing

Testing matters

- You need to convince others that the module works

Catch problems earlier

- Bugs become obscure beyond the unit they occur in

Don't confuse volume with quality of test data

- Can lose relevant cases in mass of irrelevant ones
- Look for revealing subdomains

Choose test data to cover:

- Specification (black box testing)
- Code (glass box testing)

Testing can't generally prove absence of bugs

- But it can increase quality and confidence