CSE 331

Software Design & Implementation

Topic: Testing

Discussion: What was difficult about HW3?

Reminders

- Quite a few people are still working on HW3
- HW4 has been released!

Upcoming Deadlines

- Prep. Quiz: HW4 due Monday (7/11)
- HW4 due Thursday (7/14)

Late Days

"No questions asked" late day policy:

- No more than ***one*** **late day** per assignment.
- No more than **six late days** total during the quarter.

"Questions asked" policy:

- Email us if you need more time
- Potential Downsides:
  - we may not be able to get you feedback quickly
  - you may fall behind on future assignments

Some quick reasoning…

Assertion 1: Students feel motivated to cheat in high-stress environments.
Assertion 2: Many of you find CSE 331 to be a high-stress environment.

=> Many of you feel motivated to cheat

Don't do it!

- academically dishonest
- it won't get you a high grade on an assignment
- it will build an unhealthy reliance and degrade your thinking

Instead come to talk to the course staff. We'll help you.

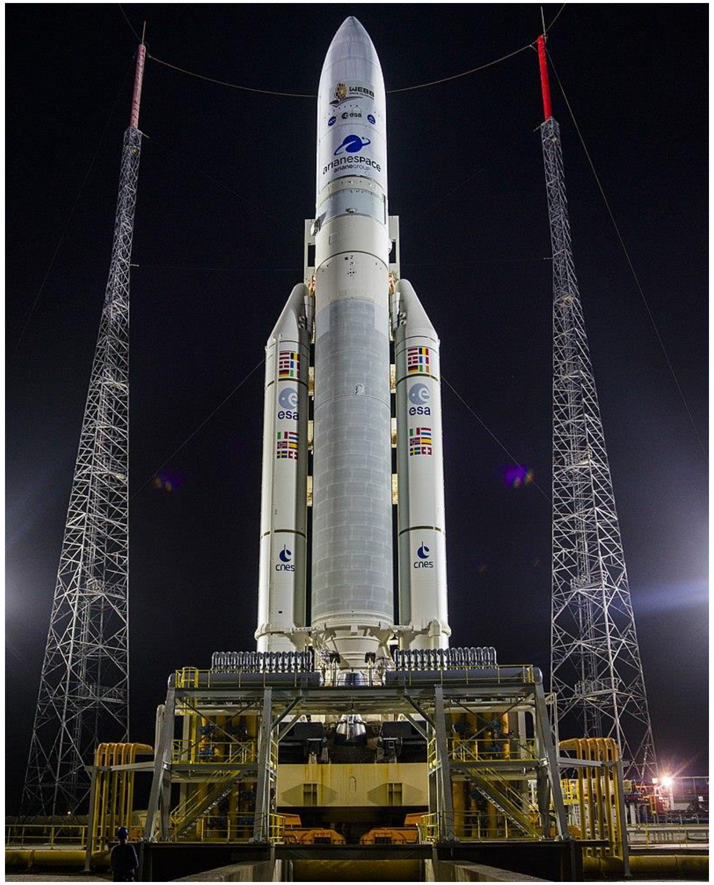
Last Time…

- Abstract Data Types
- Representation Invariants
- Representation Exposure
  - copy in/out
  - immutable
  - unmodifiable
- Abstraction Functions
  - IntDeque

Today's Agenda

- Testing
- Testing Heuristics

Ariane 5



Ariane 5 was a European rocket

- first launch in June 1996

Ariane 5

Ariane 5 was a European rocket

- first launch in June 1996

Event: Rocket self-destructed after 37s

Problem: A control software bug that went undetected

- Converted from 64-bit float to 16-bit signed integer
- Code was reused from Ariadne 4
- Threw an exception!

Cost: $500 million

Therac-25 radiation therapy machine

Designed to be a computer-controlled health tool for radiation therapy.

Event: Excessive radiation killed patients (1985-87)

Problem: Laser would fire in high-energy mode

- Previous versions had hardware interlocks
- When an operator clicked the wrong button and exited the menu quickly, it might still fire the beam

Cost: 3 human lives

How do we ensure correctness?

Best practice: use three techniques (we'll study each)

1. **Tools**
   – type checkers, test runners, etc.
2. **Inspection**
   – think through your code carefully
   – have another person review your code
3. **Testing**
   – usually >50% of the work in building software

we've just discussed inspection, i.e. "reasoning"

Together can catch >97% of bugs.

What can you learn from testing?

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

Edsgar Dijkstra

Notes on Structured Programming, 1970

Only **reasoning** can prove there are no bugs.

So why do anything else?

How do we ensure correctness?

“Beware of bugs in the above code;
I have only proved it correct, not tried it.”
-Donald Knuth, 1977

Trying it is a surprisingly useful way to find mistakes!

No **single activity** or approach can guarantee correctness

We need tools **and** inspection **and** testing to ensure correctness

Why you will care about testing

In all likelihood, you will be expected to **test your own code**

- Industry-wide trend toward developers doing more testing
  - 20 years ago, we had large test teams
  - now, test teams are small to nonexistent

- Reasons for this change:
  1. easy to update products after shipping (users are testers)
  2. often lowered quality expectations (startups, games)
     - some larger companies want to be more like startups

This has positive and negative effects…

It’s hard to test your own code

Your **psychology** is fighting against you:

- confirmation bias
  - tendency to avoid evidence that you’re wrong
- operant conditioning
  - programmers get cookies when the code works
  - testers get cookies when the code breaks

You can avoid some effects of confirmation bias by

writing most of your tests before the code

Not much you can do about operant conditioning

What is testing?

- Testing is when you run the program and observes its operation
  - Profiling a program to measure its speed or memory usage
  - Debugging code

- You’ve already seen testing in HW2 and HW3!
- For HW4, you will need to write some tests
- For HW5, you will need to write all the tests

What is testing?

A test case for the function `f(...)` consists of two parts:

- test inputs
- test oracle

```
a = 42;
b = g(...);
c = h(a, ...);

assert f(a, b) == c;
```

Kinds of testing

- Testing field has terminology for different kinds of tests
  – we won't discuss all the kinds and terms
- Here are three orthogonal dimensions [so 12 varieties total]:
  – *unit* testing versus *integration* versus *system / end-to-end* testing
    - one module's functionality versus pieces fitting together
  – *clear-box* testing versus *opaque-box / black-box* testing
    - did you look at the code before writing the test?
  – *specification* testing versus *implementation* testing
    - test only behavior guaranteed by specification or other behavior expected for the implementation

Phases of Testing

- A unit test focuses on one class / module (or even less)
  - could write a unit test for a single method
  - tests a single unit in isolation from all others

- An integration test verifies that some modules fit together properly
  - usually don't want these until the units are well tested
    - i.e., unit tests come first

- A system test runs the entire system (i.e. all modules) to check whether the system works in realistic scenarios
  - usually hard to come up with
  - may take a long time to run

How is testing done?

Write the test

1) Choose input / configuration
2) Define the expected outcome

Run the test

3) Run with input and record the actual outcome
4) Compare *actual* outcome to *expected* outcome

What’s So Hard About Testing?

“Just try it and see if it works...”

```
// requires: 1 ≤ x,y,z ≤ 100,000
// returns:  computes some f(x,y,z)
int func1(int x, int y, int z){…}
```

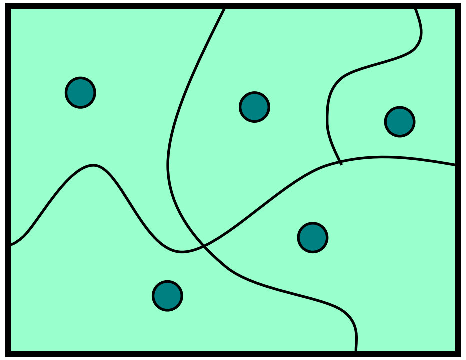
Exhaustive testing would require 1 quadrillion cases!

- impractical even for this trivially small problem

Key problem: choosing test suite

- Large/diverse enough to provide a useful amount of validation
- (Small enough to write in reasonable amount of time.)
  - need to think through the expected outcome
  - very few software projects have *too many* tests

Approach: Partition the Input Space



Ideal test suite:

Identify sets with "same behavior"
(actual and expected)

Test **at least** one input from each set
(we call this set a *subdomain*)

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;
}
```

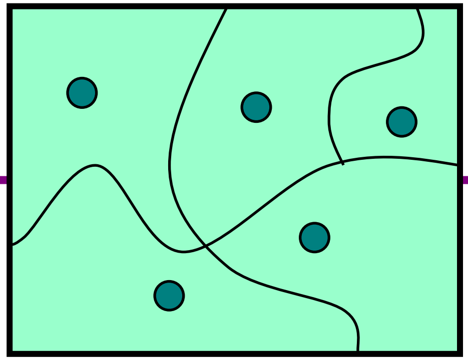
{-3, 3} does not reveal the error!

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

Three possible behaviors:

- x < -2 OK, x = -2 or x= -1 (BAD)
- x >= 0 OK

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 at least one input from a revealing subdomain to find bug
  - if you test one input from every revealing subdomain for E,
    you are guaranteed to find the bug

- The trick is to *guess* revealing subdomains for **the errors present**
  - even though your reasoning says your code is correct,
    make educated guesses where the bugs might be

Testing Heuristics

- Testing is *essential* but difficult
  - want set of tests likely to reveal the bugs present
  - but we don't know where the bugs are

- Our approach:
  - split the input space into enough subsets (subdomains) such that inputs in each one are likely all correct or incorrect
  - can then take just one example from each subdomain

- Some heuristics are useful for choosing subdomains...

Heuristics for Designing Test Suites

A good heuristic gives:

- for all errors in some class of errors E:
  high probability that some subdomain is revealing for E
- not an *absurdly* large number of subdomains

Different heuristics target different classes of errors

- in practice, combine multiple heuristics
  - (we will see several)
- a way to think about and communicate your test choices

Specification Testing

Heuristic: Explore alternate cases in the specification

Procedure is opaque-box: specification visible, internals hidden

Example

```
// 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)*

Specification Testing: Advantages

Process is not influenced by component being tested

- avoids psychological biases we discussed earlier
- can only do this for your own code if you **write tests first**

Robust with respect to changes in implementation

- test data need not be changed when code is changed

Allows others to test the code (rare nowadays)

Heuristic: Clear-box testing

Focus on features not described by specification

- control-flow details (e.g., conditions of “if” statements in code)
- performance optimizations
- alternate algorithms for different cases

Example: `abs` from before (different behavior < 0 and >= 0)

```
// @return |x|
int abs(int x) {
   if (x < 0) return -x;
   else       return x;
}
```

Clear-box Example

There are some subdomains that opaque-box testing won't catch:

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

// initialize the cache ...

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

Clear Box Testing: [Dis]Advantages

- Finds an important class of boundaries
  - yields useful test cases
  - wouldn't know about `primeTable` otherwise

Disadvantage:

- buggy code tricks you into thinking it's right once you look at it
  - (confirmation bias)
- can end up with tests having same bugs as implementation
- so also write tests **before** looking at the code

Clear-box Example

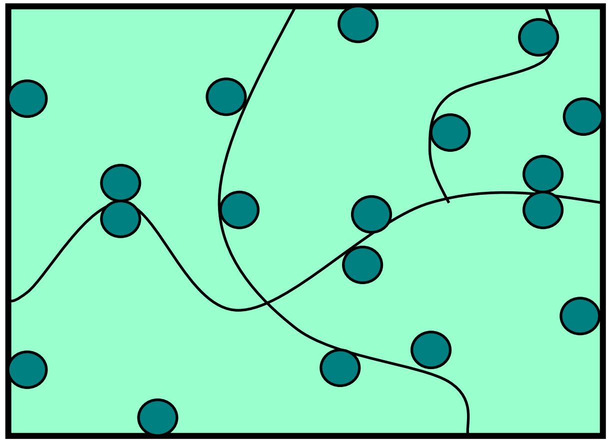
There are some subdomains that opaque-box testing won't catch:

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

// initialize the cache ...

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

Where is the bug?

Heuristic: Boundary Cases

Create tests at the boundaries between subdomains



Edges of the "main" subdomains have a high probability of revealing errors

- e.g., off-by-one bugs

Include one example on each side of the boundary

Also want to test the side edges of the subdomains…

Summary of Heuristics

Before you write the code (part of ”test-driven development”):

- split subdomains on boundaries appearing in the specification
- choose a test along both sides of each boundary

After you write the code:

- split further on boundaries appearing in the implementation

More next time…

On the other hand, don't confuse *volume* with *quality* of tests

- look for revealing subdomains
- want tests in every revealing subdomain not **just** lots of tests

Before next class…

1. Start on Prep. Quiz: HW4 as early as possible!
   - Reminds you about common set operations
     - E.g. union, intersection, complement
   - Think about some non-trivial cases needed for the homework