

Testing

Based on material by Michael Ernst,
University of Washington

Announcements

- HW0, HW1, HW2 and HW3 are graded
 - Grades and feedback in Homework Server (coming up after this class)
 - Email questions
- Exam 1, Quiz 1,2,3
 - Grades in LMS (coming up after this class)

2

Announcements

■ Exam 1

A: [135 – 150]	28
B: [120 – 135]	23
C: [105 – 120]	24

Fall 15 CSCI 2600, A. Milanova

3

Exam Review: GCD with Division

```
x = a;
y = b;
while (x != y) {
  Inv: gcd(x,y) = gcd(a,b)
  if (x > y) {
    x = x-y;
  }
  else {
    y = y-x;
  }
}
return x;
```

```
x = a;
y = b;
while (y != 0) {
  Inv: gcd(x,y) = gcd(a,b)
  t = y;
  y = x%y;
  x = t;
}
return x;
```

4

Exam Review: Specification Strength

`double sqrt(double x)`

Spec A: requires: $x \geq 0$

returns: y such that $|y*y - x| < 0.001$

Spec B: requires: $x \geq 0$

returns: y such that $|y*y - x| < 0.0001$

Spec C: returns: y such that $|y*y - x| < 0.001$ if $x \geq 0$,
and 0.0 if $x < 0$

Spec D: returns: y such that $|y*y - x| < 0.001$ if $x \geq 0$
throws: `IllegalArgumentException` if $x < 0$

5

And More...

- Benevolent side effects
- Representation exposure
- Rep invariants
 - Matrix
 - Heap

Fall 15 CSCI 2600, A. Milanova

6

Announcements

- HW4 will be out tonight, due October 20th
 - **LONG!!!**
 - **START EARLY!!!** Several tasks:
 - 1) design the ADT following the ADT methodology
 - Mutable vs. immutable
 - Operations (creators, mutators, etc.) + their specs
 - 2) write a test suite based on those specs (before coding, test-first principle)
 - 3) then code
 - 4) then augment test suite based on actual implementation and measure coverage

7

Outline

- **Testing**
 - Introduction
 - Strategies for choosing tests suites
 - Black-box testing
 - White-box testing

Fall 15 CSCI 2600, A Milanova

8

What is Testing?

- **Testing**: the process of executing software with the intent of finding errors
- **Good testing**: a high probability of finding yet-undiscovered errors
- **Successful testing**: discovers unknown errors
- “Program testing can be used to show the presence of bugs, but never to show their absence.” Edsger Dijkstra 1970

Fall 15 CSCI 2600, A Milanova

9

Quality Assurance (QA)

- The process of uncovering problems and improving the quality of software. Testing is the major part of QA
- QA is **testing** plus other activities:
 - Static analysis (finding bugs without execution)
 - Proofs of correctness (theorems)
 - Code reviews (people reading each other's code)
 - Software process (development methodology)
- No single activity or approach can guarantee software quality

10

Famous Software Bugs

- Ariane 5 rocket's first launch in 1996
 - The rocket exploded 37 seconds after launch
 - Reason: a bug in control software
 - Cost: over \$1 billion
- Therac-25 radiation therapy machine
 - Excessive radiation killed patients
 - Reason: software bug linked to a race condition, missed during testing

Fall 15 CSCI 2600, A Milanova

11

Famous Software Bugs

- Mars Polar Lander
 - Legs deployed after sensor falsely indicated craft had touched down 130 feet above surface
 - Reason: one bad line of software
 - Cost: \$110 million
- And many more...
 - Northeast blackout (2003)
 - Toyota Prius breaks and engine stalling (2005)
 - And many many more...

Fall 15 CSCI 2600, A Milanova

12

Cost to Society (Source: NIST Planning Report 2002)

- Inadequate testing infrastructure costs the US **\$22-60 billion** annually
- Testing accounts for **50% of software development cost**
 - Program understanding and debugging accounts for up to **70% of time** to ship a software product
 - Maintenance (bug fixes and upgrades) accounts for up to 95% of total software cost
- Improvement in testing infrastructure can save **one third of the cost**

13

Scope (Phases) of Testing

- Unit testing
 - Does each module do what it is supposed to do?
- Integration testing
 - Do the parts, when put together, produce the right result?
- System testing
 - Does program satisfy functional requirements?
 - Does it work within overall system?
 - Behavior under increased loads, failure behavior, etc.

Fall 15 CSCI 2600, A Milanova

14

Unit Testing

- Our focus will be on unit testing
- Tests a single unit in isolation from all others
- In object-oriented programming, unit testing mostly means **class testing**
 - Tests a single class in isolation from others

Fall 15 CSCI 2600, A Milanova

15

Why Is Testing So Hard?

- // requires: $1 \leq x, y, z \leq 10000$
// returns: computes some $f(x, y, z)$
int proc(int x, int y, int z)
- Exhaustive testing would require 1 trillion runs! And this is a trivially small problem
 - The key problem: **choosing set of inputs (i.e., test suite)**
 - Small enough to finish quickly
 - Large enough to validate program

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

16

sqrt Example

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

- What are some values of **x** worth trying?
 - $x < 0$ (exception thrown)
 - $x \geq 0$ (returns normally)
 - around 0 (boundary conditions)
 - Perfect squares, non-perfect squares
 - $x < 1$ (**sqrt(x) > x** in this case), $x = 1$, $x > 1$

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

17

Outline

- Testing
 - Introduction
 - **Strategies for choosing tests suites**
 - Black box testing
 - White box testing
- Catch up: exceptions

Fall 15 CSCI 2600, A Milanova

18

Testing Strategies

- Test case: specifies
 - Inputs + pre-test state of the software
 - Expected result (outputs and post-test state)
- **Black box testing:**
 - We ignore the code of the program. We look at the specification (roughly, given some input, was the produced output correct according to the spec?)
 - Choose inputs without looking at the code
- **White box (clear box, glass box) testing:**
 - We use knowledge of the code of the program (roughly, we write tests to "cover" internal paths)
 - Choose inputs with knowledge of implementation

Fall 15 CSCI 2600, A Milanova

19

Black Box Testing Advantages

- Robust with respect to changes in implementation (independent of implementation)
 - Test data need not be changed when code is changed
- Allows for independent testers
 - Testers need not be familiar with implementation
 - Tests can be developed before code based on specifications. (Do this in HW4!)

20

Black Box Testing Heuristic

- Choose test inputs based on paths in specification

```
// returns: a if a > b
//          b if b > a
//          a if a = b
int max(int a, int b)
```

- 3 paths, 3 test cases:
 - (4,3) => 4 (input along path a > b)
 - (3,4) => 4 (input along path b > a)
 - (3,3) => 3 (input along path a = b)

Fall 15 CSCI 2600, A Milanova

21

Black Box Testing Heuristic

- Choose test inputs based on paths in specification
 - // returns: index of first occurrence of **value** in **a**
// or -1 if **value** does not occur in **a**
int find(int[] a, int value)
- What are good test cases?
 - ([4,3,5,6], 5) => 2
 - ([4,3,5,6], 7) => -1
 - ([4,5,3,5], 5) => 1

Fall 15 CSCI 2600, A Milanova

22

sqrt Example

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

- What are some values of **x** worth trying?
 - We used this heuristic in sqrt example. It tells us to try a value of x < 0 (exception thrown) and a value of x >= 0 (returns normally) are worth trying

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

23

Black Box Heuristics

- "Paths in specification" heuristic is a form of **equivalence partitioning**
- Equivalence partitioning partitions input and output domains into **equivalence classes**
 - Intuition: values from different classes drive program through different paths
 - Intuition: values from the same equivalence class drive program through "same path", program will likely behave "equivalently"

24

Black Box Heuristics

- Choose test inputs from each equiv. class

```
// returns: 0 <= result <= 5
// throws: SomeException if arg < 0 || arg > 10
int proc(int arg)
There are three equivalence classes:
"arg < 0", "0 <= arg <= 10" and "10 < arg".
We write tests with values of arg from each class
```

- Stronger vs. weaker spec. What if the spec said // requires: $0 \leq \text{arg} \leq 10$?

25

Equivalence Partitioning

- Examples of equivalence classes
 - Valid input x in interval $[a..b]$: this defines three classes " $x < a$ ", " $a \leq x \leq b$ ", " $b < x$ "
 - Input x is boolean: classes "true" and "false"
- Choosing test values
 - Choose a **typical** value in the middle of the "main" class (the one that represents valid input)
 - Also choose values at the **boundaries** of all classes: e.g., use $a-1, a, a+1, b-1, b, b+1$

26

Black Box Testing Heuristic: Boundary Value Analysis

- Idea: choose test inputs at the edges of the equivalence classes
- Why?
 - Off-by-one bugs, forgot to handle empty container, overflow errors in arithmetic
- Cases at the **edges** of the "main" class have high probability of revealing these common errors
- Complements equivalence partitioning

27

Equivalence Partitioning and Boundary Values

- Suppose our specification says that **valid input** is an array of 4 to 24 numbers, and each number is a 3-digit positive integer
 - One dimension: partition size of array
 - Classes are " $n < 4$ ", " $4 \leq n \leq 24$ ", " $24 < n$ "
 - Chosen values: 3, 4, 5, 14, 23, 24, 25
 - Another dimension: partition integer values
 - Classes are " $x < 100$ ", " $100 \leq x \leq 999$ ", " $999 < x$ "
 - Chosen values: 99, 100, 101, 500, 998, 999, 1000

28

Equivalence Partitioning and Boundary Values

- Equivalence partitioning and boundary value analysis apply to **output** domain as well
- Suppose that the spec says "the output is an array of 3 to 6 numbers, each one an integer in the range 1000 - 2500"
 - Test with inputs that produce (for example):
 - 3 outputs with value 1000
 - 3 outputs with value 2500
 - 6 outputs with value 1000
 - 6 outputs with value 2500
 - More tests...

Fall 15 CSCI 2600, A Milanova

29

Equivalence Partitioning and Boundary Values

- ```
// returns: index of first occurrence of value in a,
// or -1 if value does not occur in a
int find(int[] a, int value)
```
- What is a good partition of the input domain?
  - One dimension: size of the array
    - People often make errors for arrays of size 1, we decide to create a separate equivalence class
    - Classes are "empty array", "array with one element", "array with many elements"

Fall 15 CSCI 2600, A Milanova

30

## Equivalence Partitioning and Boundary Values

- We can also partition the output domain: the location of the value
  - Four classes: "first element", "last element", "middle element", "not found"

| Array       | Value | Output              |
|-------------|-------|---------------------|
| Empty       | 5     | -1                  |
| [7]         | 7     | 0                   |
| [7]         | 2     | -1                  |
| [1,6,4,7,2] | 1     | 0 (boundary, start) |
| [1,6,4,7,2] | 4     | 2 (mid array)       |
| [1,6,4,7,2] | 2     | 4 (boundary, end)   |
| [1,6,4,7,2] | 3     | -1                  |

Fall 15 CSCI 2600, A Milanova

31

## Other Boundary Cases

- Arithmetic
  - Smallest/largest values
  - Zero
- Objects
  - Null
  - Circular list
  - Same object passed to multiple arguments (**aliasing**)

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

32

## Boundary Value Analysis: Arithmetic Overflow

// returns: |x|

```
public int abs(int x)
```

- What are some values worth trying?
  - Equivalence classes are  $x < 0$  and  $x \geq 0$
  - $x = -1$ ,  $x = 1$ ,  $x = 0$  (boundary condition)
- How about  $x = \text{Integer.MIN\_VALUE}$ ?
  - // this is  $-2^{31}$
  - // `System.out.println(Math.abs(x) < 0)` prints true!

Fall 15 CSCI 2600, A Milanova (based on slide by Mike Ernst)

33

## Boundary Value Analysis: Aliasing

// modifies: src, dest

// effects: removes all elements of src and appends them in reverse order to the end of dest

```
void appendList(List<Integer> src,
 List<Integer> dst) {
 while (src.size() > 0) {
 Integer elt = src.remove(src.size()-1);
 dest.add(elt);
 }
}
```

- What happens if we run `appendList(list, list)`? **Aliasing.**

34

## Summary So Far

- Testing is hard. We cannot run all inputs
- Key problem: **choose test suites** such that
  - Small enough to finish in reasonable time
  - Large enough to validate the program (reveal bugs, or build confidence in absence of bugs)
- All we have is heuristics!
  - We saw **black box testing heuristics**: run paths in spec, partition input/output into equivalence classes, run with input values at boundaries of these classes
  - There are also **white box testing heuristics**

35

## White Box Testing

- Ensure test suite **covers** (covers means **executes**) **all of the program**
- Measure quality of test suite with **% coverage**
- Assumption: high coverage implies few errors in program
- Focus: features not described in specification
  - Control-flow details
  - Performance optimizations
  - Alternate algorithms (paths) for different cases

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

36

## White Box Complements Black Box

```
boolean[] primeTable[CACHE_SIZE]

// returns: true if x is prime
// false otherwise
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];
}
```

Fall 15 CSCI 2600, A Milanova (example due to Michael Ernst)

37

## White Box Testing: Control-flow-based Testing

- **Control-flow-based white box testing:**
  - Extract a control flow graph (CFG)
  - Test suite must cover (execute) certain elements of this control-flow graph
- Idea: Define a **coverage target** and ensure test suite covers target
  - Targets: nodes, branch edges, paths
  - Coverage target approximates "all of the program"

Fall 15 CSCI 2600, A Milanova

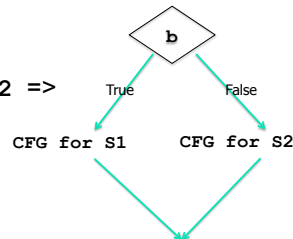
38

## Aside: Control-flow Graph (CFG)

- Assignment  $x=y+z \Rightarrow$  node in CFG:  $x=y+z$

- If-then-else

if (b) S1 else S2  $\Rightarrow$



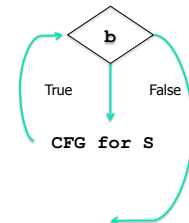
Fall 15 CSCI 2600, A Milanova

39

## Aside: Control-flow Graph (CFG)

- Loop

while (b) S  $\Rightarrow$



Fall 15 CSCI 2600, A Milanova

40

## Aside: Control Flow Graph (CFG)

- Draw the CFG for the code below:

```
1 s:= 0;
2 x:= 0;
3 while (x<y) {
4 x:=x+3;
5 y:=y+2;
6 if (x+y<10)
7 s:=s+x+y;
8 else
9 s:=s-x-y;
}
```

41

## Statement Coverage

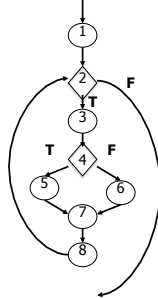
- Traditional target: **statement coverage**. Write test suite that covers **all statements**, or in other words, **all nodes in the CFG**
- Motivation: code that has never been executed during testing may contain errors
  - Often this is the "low-probability" code

Fall 15 CSCI 2600, A Milanova

42

## Example

- Suppose that we write and execute two test cases
- Test case #1: follows path 1-2-exit (e.g., we never take the loop)
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit (loop twice, and both times take the true branch)
- Problems?



## Branch Coverage

- Target: write test cases that cover all **branch edges** at predicate nodes
  - True and false branch edges of each if-then-else
  - The two branch edges corresponding to the condition of a loop
  - All alternatives in a SWITCH statement
- In modern languages, branch coverage implies statement coverage

Fall 15 CSCI 2600, A. Milanova

44

## Branch Coverage

- Motivation for branch coverage: experience shows that many errors occur in “decision making” (i.e., branching). Plus, it implies statement coverage
- Statement coverage does not imply branch coverage
  - I.e., a suite that achieves 100% statement coverage does not necessarily achieve 100% branch coverage
  - Can you think of an example?

45

## Example

```
static int min(int a, int b) {
 int r = a;
 if (a <= b)
 r = b;
 return r;
}
```

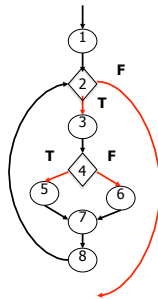
- Let's test with `min(1, 2)`
- What is the statement coverage?
- What is the branch coverage?

Fall 15 CSCI 2600, A. Milanova

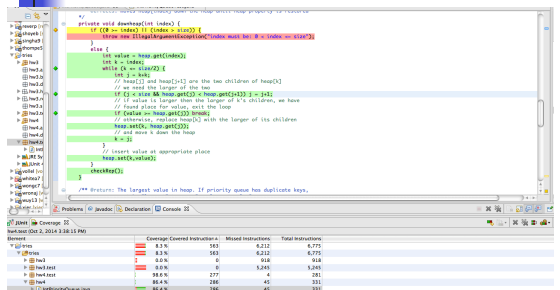
46

## Example

- We need to cover the **red** branch edges
- Test case #1: follows path 1-2-exit
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit
- What is % branch coverage?



## Code Coverage in Eclipse



Fall 15 CSCI 2600, A. Milanova

48



## Rules of Testing

- First rule of testing: Do it early and do it often
  - Best to catch bugs soon, before they hide
  - Automate the process
  - Regression testing will save time
- Second rule of testing: Be systematic
  - Writing tests is a good way to understand the spec
  - Specs can be buggy too!
  - When you find a bug, write a test first, then fix