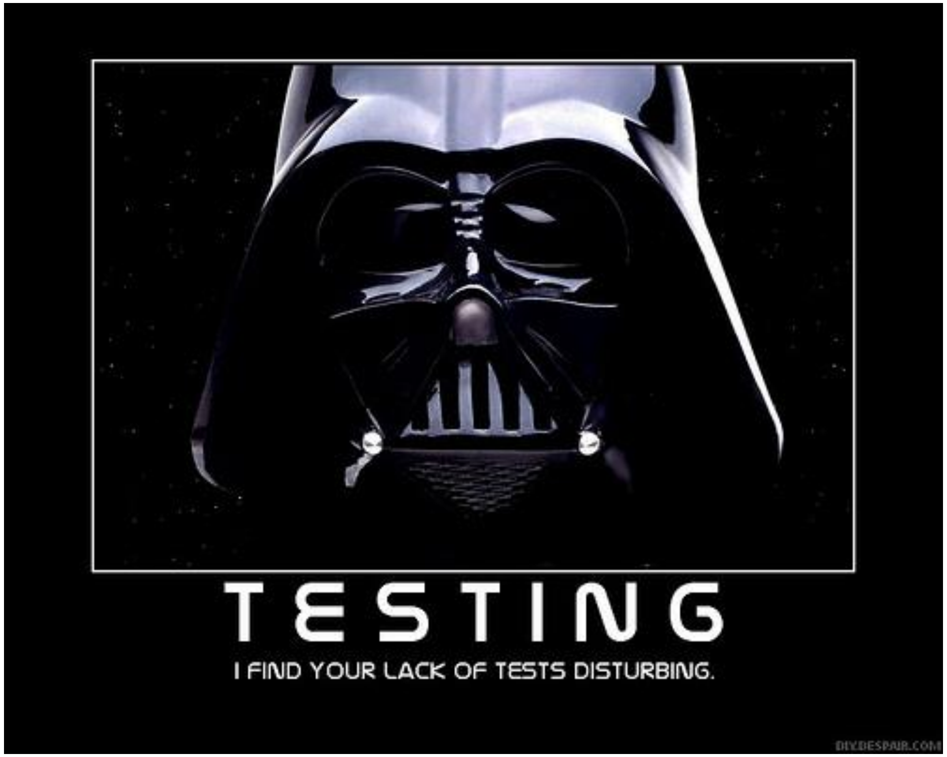
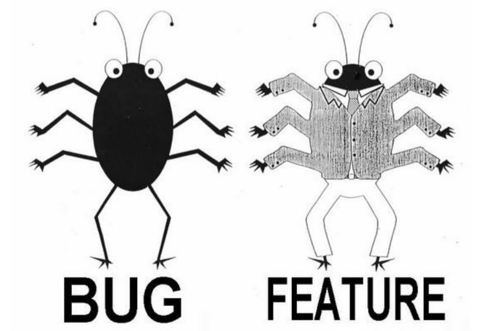
Testing


TESTING
I FIND YOUR LACK OF TESTS DISTURBING.
DIY.DESPAIR.COM

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

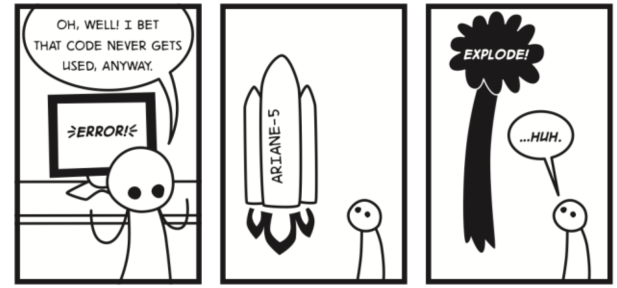
- "Program testing can be used to show the presence of bugs, but never to show their absence." Edsger Dijkstra 1970

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

Reasoning about code

Famous Software Bugs



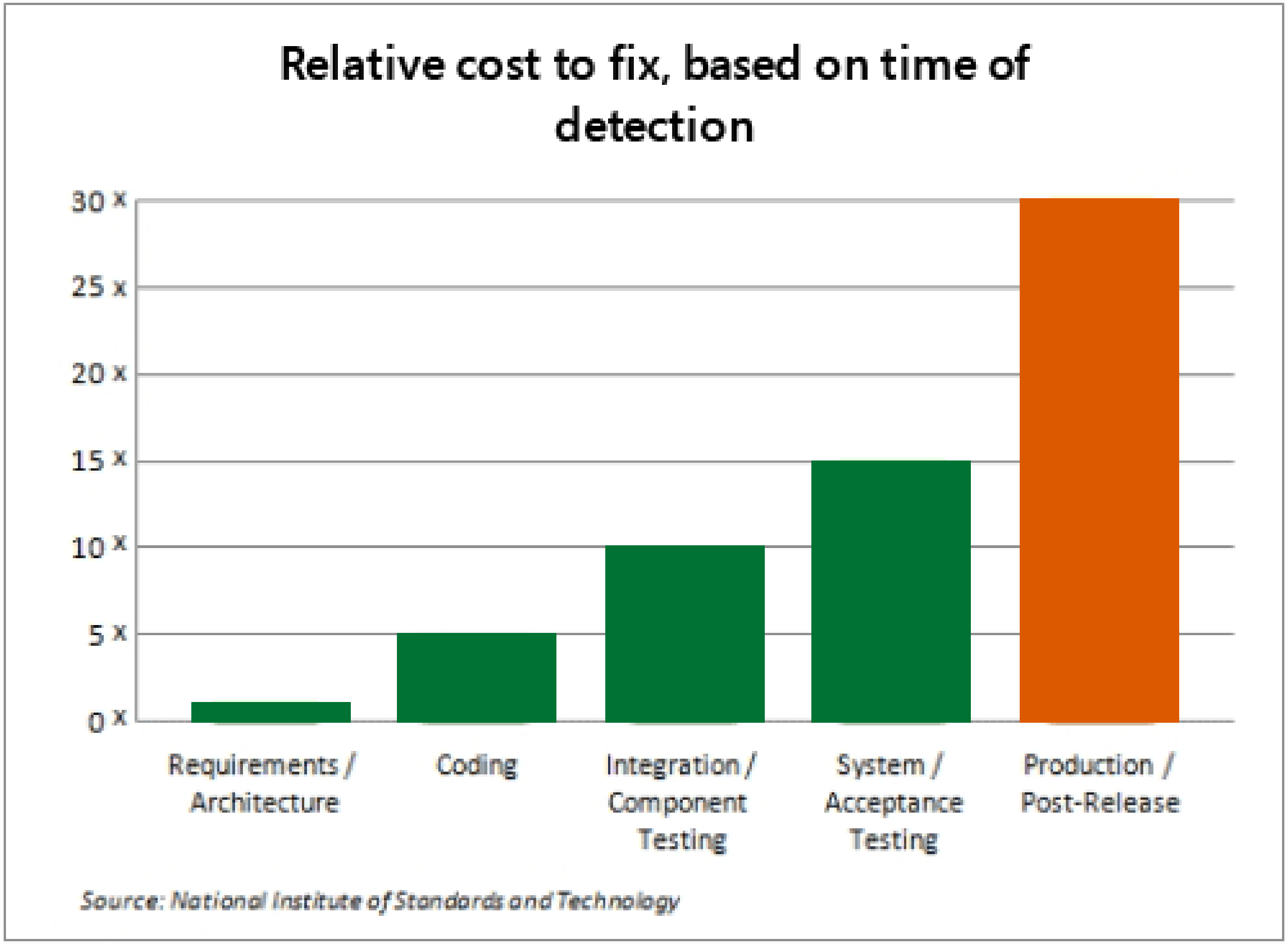
- Ariane 5 rocket's first launch in 1996
  - The rocket exploded 37 seconds after launch
  - Reason: a bug in control software
  - Cost: over $1 billion
- Therac-25 radiation therapy machine
  - Excessive radiation killed patients
  - Reason: software bug linked to a race condition, missed during testing

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)
  - Facebook bug made 14 million users' posts public
  - Mt. Gox hack – 200,000 bitcoins lost
    - Security "bug" lead to theft
  - And many, many more…
  - https://raygun.com/blog/costly-software-errors-history/
  - https://en.wikipedia.org/wiki/List_of_software_bugs

Cost to Society (NIST)

- Software errors cost the US ~$60 billion annually
  - http://www.ashireporter.org/HomeInspection/Articles/Software-Errors-Cost-U-S-Economy-59-5-Billion-Annually/740
- The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure
- Testing typically accounts for 50% of software development cost



https://www.microsoft.com/en-us/SDL/about/benefits.aspx

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.

Seven Rules of Testing

- Exhaustive testing is usually not possible
- Defect Clustering
  - a small number of modules usually contain most of the defects
- Pesticide Paradox
  - Repetitive use of the same pesticide builds stronger bugs
  - If the same set of repetitive tests are conducted, the method will be useless for discovering new defects.
- Testing shows the presence of defects
  - Not absence
- Absence of Error – fallacy
  - Absence of evidence is not evidence of absence
- Test early and often
- Testing is context dependent
  - Testing an e-mail app is different than testing a student information system
  - Different data will give different results, reveal different bugs
- https://www.guru99.com/software-testing-seven-principles.html

Without Proper Testing

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
  - JUnit testing

Why Is Testing So Hard?

// requires: 1 <= x,y,z <= 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
  - Doesn’t test what happens when you violate preconditions
- The key problem: choosing a set of inputs (i.e., test suite)
  - Small enough to finish quickly
  - Large enough to validate program

sqrt Example

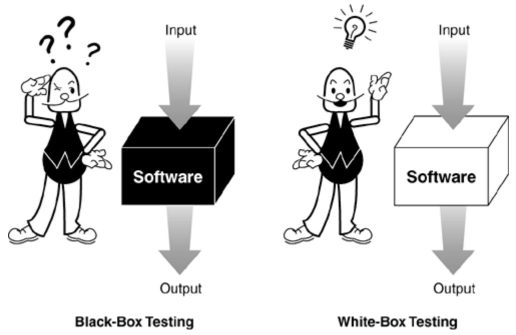
// throws: IllegalArgumentException if x < 0

// returns: approximation to square root of x

`public double sqrt(double x)`

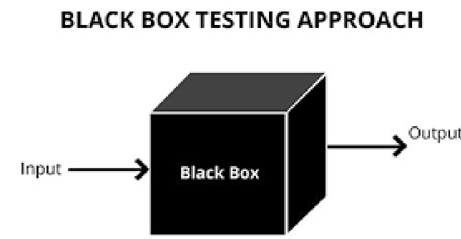
- What are some values of **`x`** worth trying?
  - x < 0 (exception thrown)
  - x >= 0 (returns normally)
  - 0 and around 0 (boundary conditions)
  - Perfect squares, non-perfect squares
  - x < 1 (**`sqrt(x)`** > x in this case), x = 1, x > 1
  - Big numbers: 2,147,483,647, 2,147,483,648
- Edge Cases are important!

Testing Strategies



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

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 <u>before</u> code based on <u>specifications</u>.
    - Do this in HW4!
- Special test methods are needed for black boxes based on AI/ML, IoT, sensors, etc. methods

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)

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
  - ([], 1) => -1

`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
  - Probably should try 0 (edge condition), very large number

Black Box Heuristics

- “Paths in specification” heuristic is a form of equivalence partitioning
- Equivalence partitioning divides 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”
  - We will not formally define equivalence classes
  - Intuitively
    - Input values have valid and invalid ranges
    - We want to choose tests from the valid, invalid regions and values near or at the boundaries of the regions

Equivalence partitioning

- Equivalence partitioning
  - Divides the input data of a software unit into partitions of equivalent data from which test cases can be derived.
  - Usually applied to input data
  - Try to test each partition at least once
- Informally, a method allows valid input for some range of arguments
  - Fails for others
  - Example int representation of months
    - Valid for 1..12
    - Invalid for < 1 and > 12
    - 3 classes of inputs
    - Boundary regions are important also

Black Box Heuristics

- Choose test inputs from each equivalence 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 <= **`arg`** <= 10 and doesn't throw anything?

Equivalence Partitioning

- Examples of equivalence classes
  - Valid input **`x`** in interval [a..b]: this defines three classes "**`x`**<a", "a<=**`x`**<=b", "**`x`**`>b`"
  - 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

Note:

- We can only run tests on invalid arguments if the spec tells us what will happen for invalid data
  - If behavior is undefined if client violates requirements, how do we test undefined behaviors?
- Black box tests are specification tests.
  - They test whether implementation conforms to specification
  - Argues for strong specs

Black Box Testing Heuristic: Boundary Value Analysis

- 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

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<=**n**<=24”, “n > 24”
    - Chosen values: 3,4,5, 14, 23,24,25
  - Another dimension: partition integer values
    - Classes are “**x**<100”, “100<=**x**<=999”, “x > 999”
    - Chosen values: 99,100,101, 500, 998,999,1000
- Dimensions are orthogonal
  - We need to test a range of array sizes and values in the array

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…
  - Of course, in this case we need to know what input values produce the various output values

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"
  - What happens if a is null?
- Previously, we partitioned the output domain: we forced -1, we forced normal output.
  - Need to test data values also

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

Other Boundary Cases

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

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 >= 0
  - x = -1, x = 1, x = 0 (boundary condition)

How about x = Integer.MIN_VALUE?
// this is -2147483648 = -2^{31}
// System.out.println(Math.abs(x) < 0) prints true!

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.
  - Infinite loop – why?

Black Box Testing

- Even with simple numerical arguments, testing can be complex
- With more complex arguments (names, addresses, complex objects, etc.) finding the correct argument partitions can be difficult
- Test complex systems early, often
  - At each stage of integration
- Use mock objects to test complex arguments

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 the 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

White/Clear Box Testing

- Testing with knowledge of the code
- Ensure test suite covers (covers means executes) all of the program
    - Executes each statement
- Measure quality of test suite with % coverage
- Assumption: successful tests with high coverage imply few errors in program
- Focus: features not described in specification
    - Control-flow details
    - Performance optimizations
    - Alternate algorithms (paths) for different cases

White Box Complements Black Box

```
boolean[] primeTable[CACHE_SIZE]
 // Requires x >= 0
// 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];
}
```

White Box Testing: Control-flow-based Testing

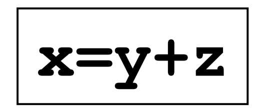
- Control-flow-based white box testing:
  - Extract a control flow graph (CFG)
  - Test suite must cover (execute) certain elements of this control-flow graph graph
- Idea: Define a coverage target and ensure test suite covers target
  - Targets: nodes, branch edges, paths
  - Coverage target approximates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

Control-flow Graph (CFG)

- Can be obtained from the program's flow graph
- Each node represents a basic block
- Two designated blocks:
  - Entry block
  - Exit block
- Directed! Edges represent jumps in the control flow
- Every edge A $\longrightarrow$ B has the property:
  outdegree(A) > 1 or indegree(B) > 1 (or both)
  - Indegree is the number of incoming edges
  - Outdegree is the number of outgoing edges

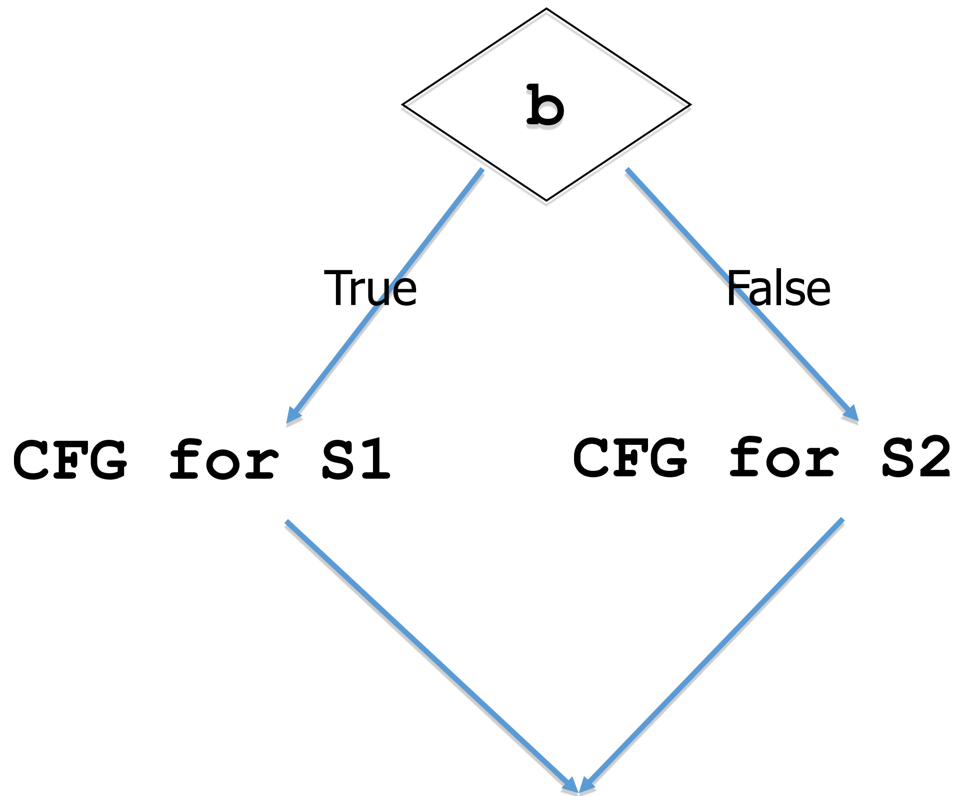
Control-flow Graph (CFG)

- Assignment **`x=y+z`** => node in CFG:



- If-then-else

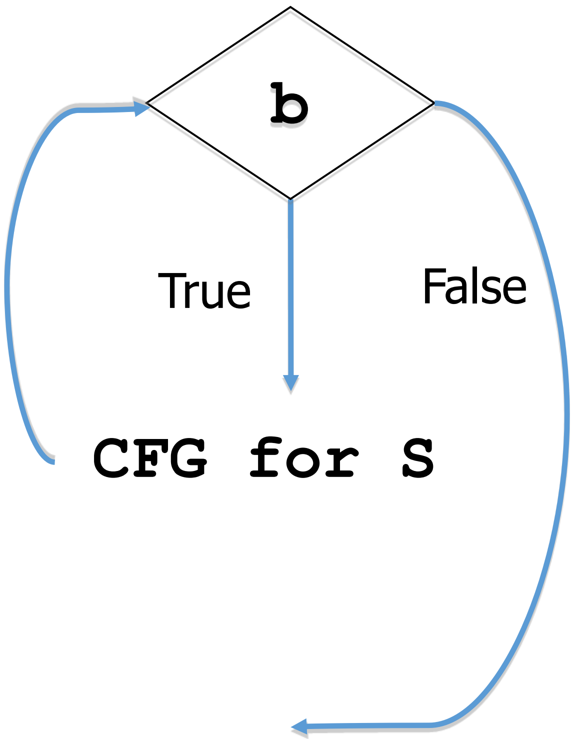
`if (b) S1 else S2` =>



Aside: Control-flow Graph (CFG)

- Loop

`while (b) S  =>`



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;
      else
8         s := s +  x - y;
  }
```

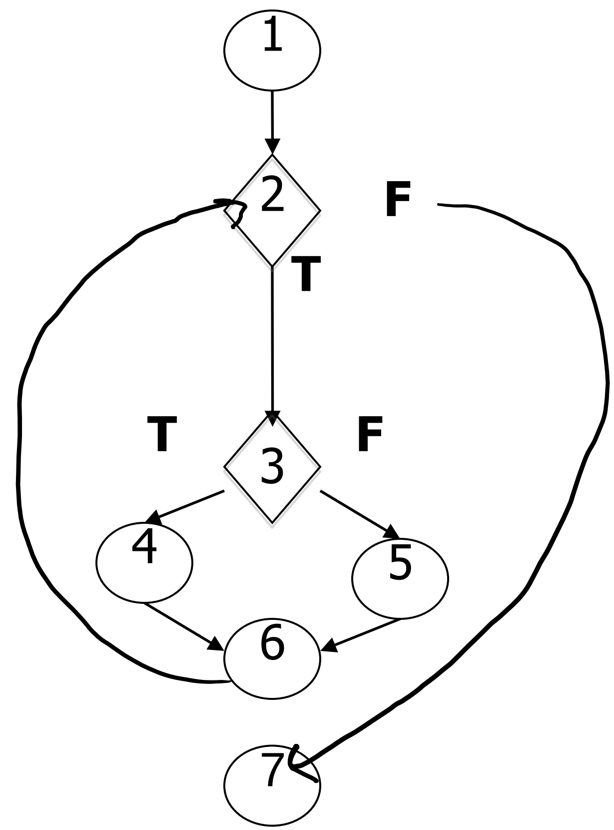
Statement Coverage

- 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

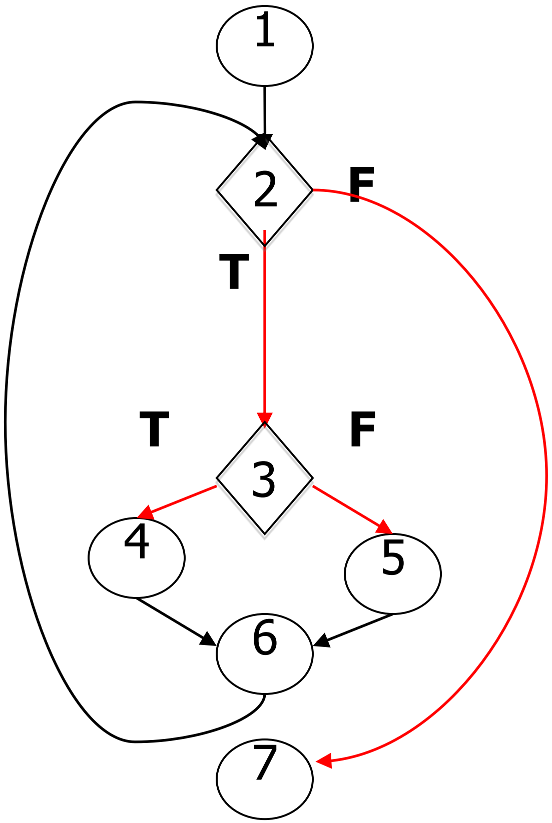
Example

- Suppose that we write and execute two test cases
- Test case #1: follows path 1-2-7 (e.g., we never take the loop)
- Test case #2: 1-2-3-4-6-2-3-4-6-2-7 (loop twice, and both times take the true branch)
- Problems?



Example

- We need to cover the red branch edges
- Test case #1: follows path 1-2-7
- Test case #2: 1-2-3-4-6-2-3-4-6-2-7
- What is % branch coverage?



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

Branch Coverage

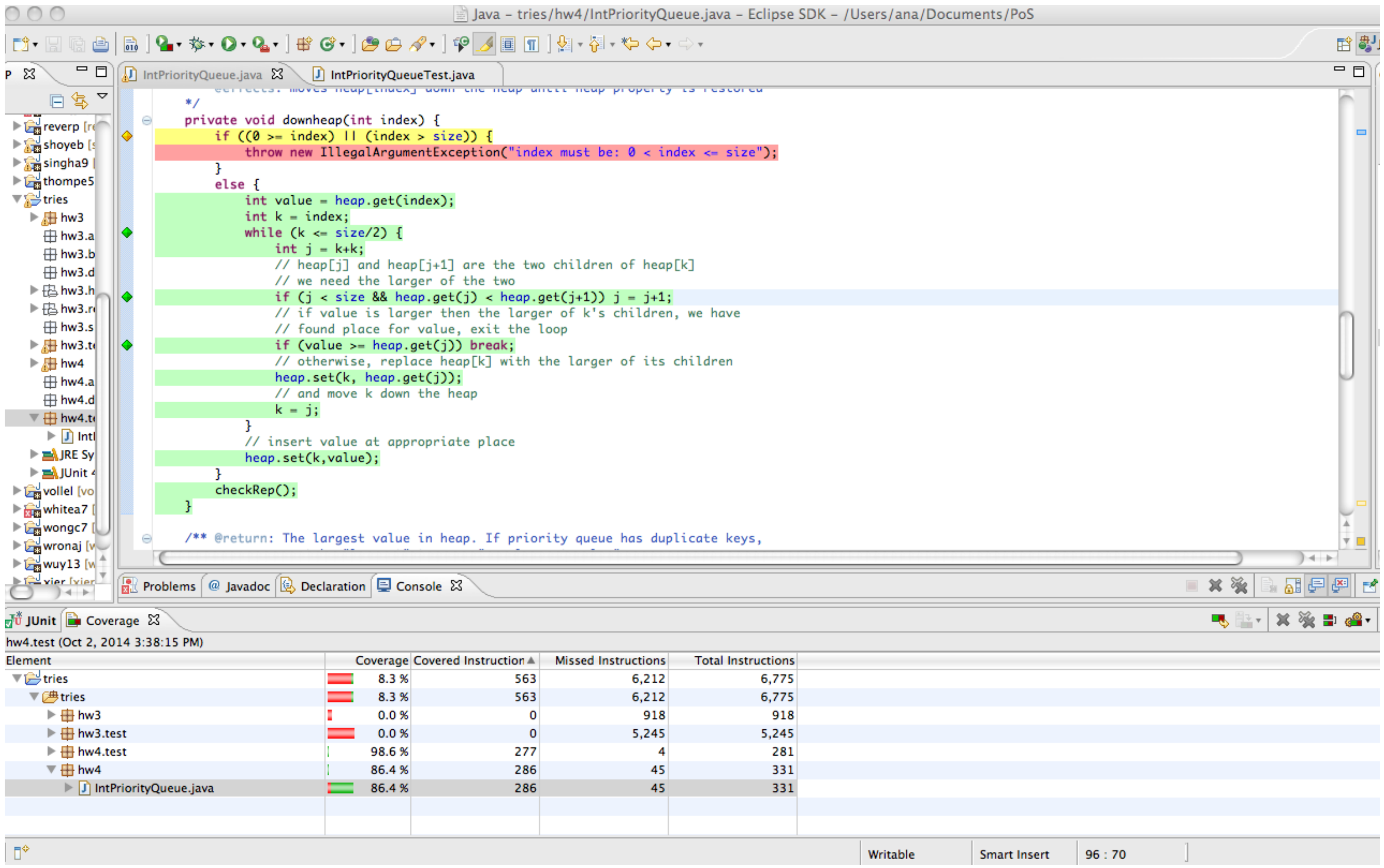
- Motivation for branch coverage: experience shows that many errors occur in “decision making” (i.e., branching). Plus, it implies statement coverage
- Statement coverage does not imply branch coverage
  - I.e., a suite that achieves 100% statement coverage does not necessarily achieve 100% branch coverage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

Example

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

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

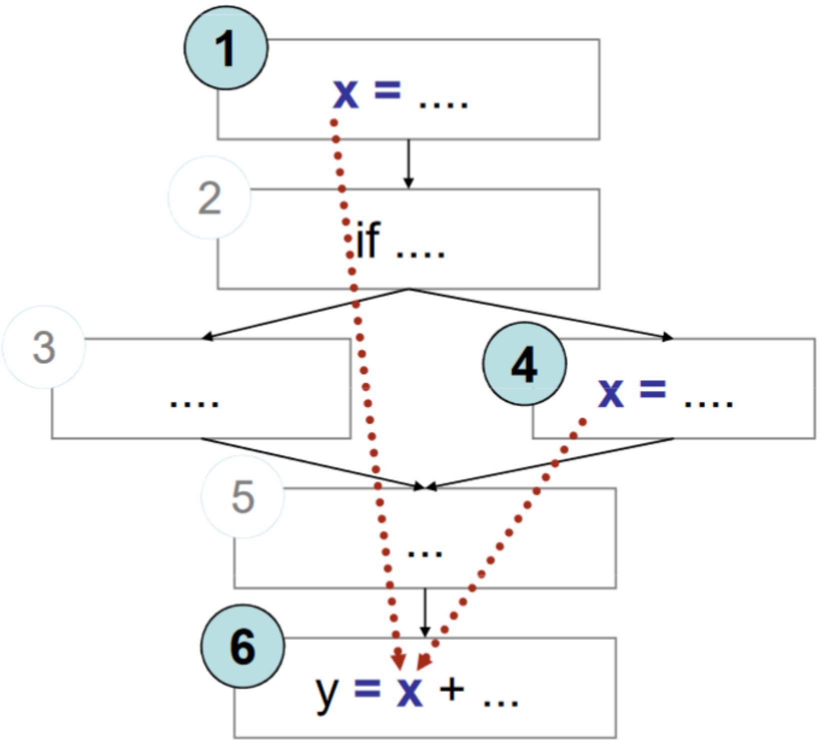
Code Coverage in Eclipse



Other White Box Heuristics

- Equivalence partitioning and boundary value analysis
- Loop testing
  - Skip loop
  - Run loop once
  - Run loop twice
  - Run loop with typical value
  - Run loop with max number of iterations
  - Run with boundary values near loop exit condition
- Branch testing
  - Run with values at the boundaries of branch condition

Difficulties



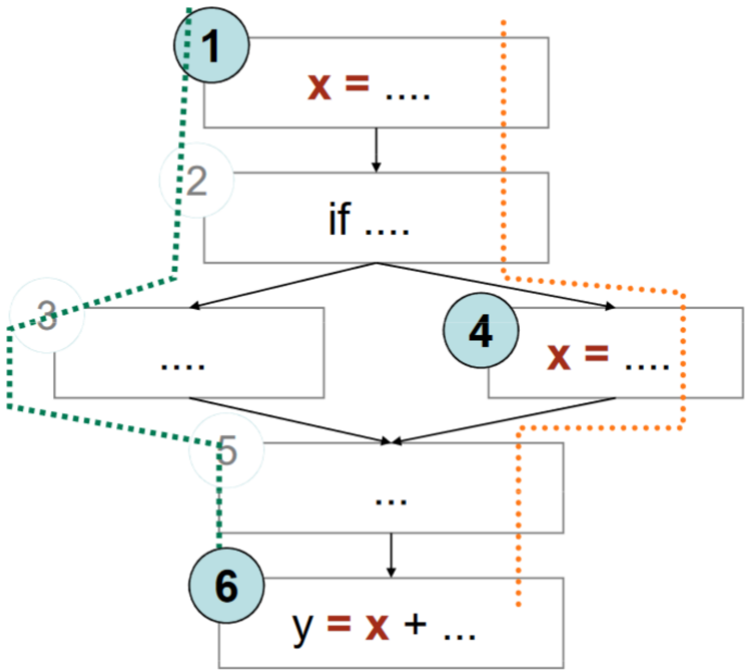
- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are *def-use (DU) pairs*
  - defs at 1,4
  - use at 6

http://www.inf.ed.ac.uk/teaching/courses/st/2015-16/Ch13.pdf

Definition-use Pairs

- A def-use (DU) pair
  - A pair of a definition and use of a variable such that at least one path exists from the definition to the use
  - x = 1;   // definition
  - y = x + 3 // use
- DU path
  - A path from the definition of a variable to a use of the same variable with no other definition of the variable on the path
  - Loops can create infinite DU paths

Definition-clear path



- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is "killed" (reassigned) at node 4
- (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

http://www.inf.ed.ac.uk/teaching/courses/st/2015-16/Ch13.pdf

Adequacy

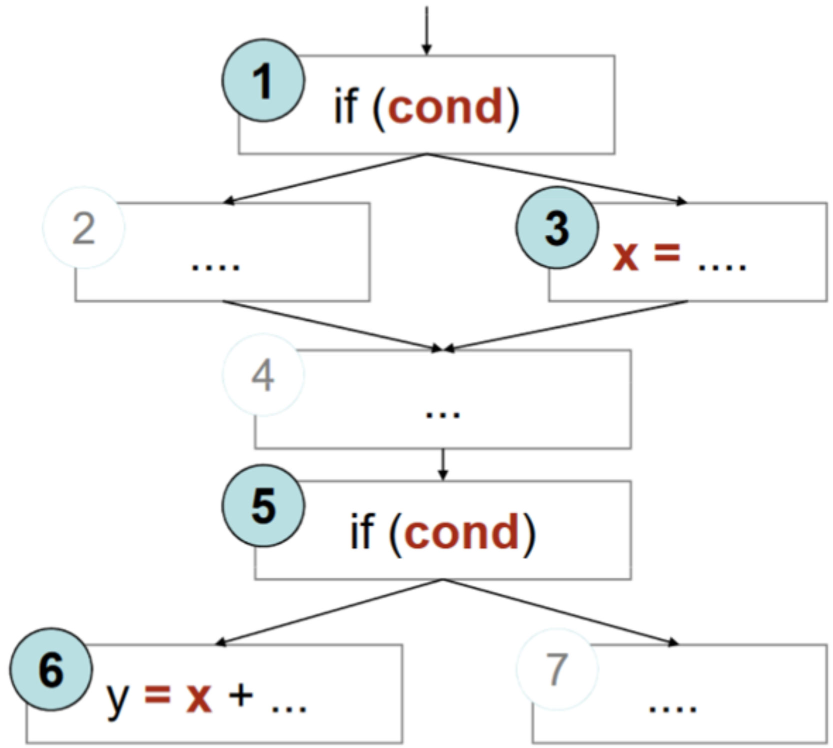
- We want to test:
- All DU pairs
  - Each DU pair tested at least once
- All DU paths
  - Each path is tested at least once
- All definitions
  - For each definition, there is at least one test that exercises a DU path containing it
  - Every computed value is used at least once

Difficulties

- x[i] = some_value; y = x[j];
  - DU pair only if i == j
- Obj x = new Obj(); x = some_value; y = x;
  - y is an alias of x
  - What happens when x or y is used?
    - If x is changed, y is changed
- m.putFoo(); y = n.getFoo();
  - Are m and n the same object?
  - Do m and n share a foo?
- Aliases can be a problem

Infeasibility

- Suppose cond doesn't change between 1 and 5
  - Or conditions could be different, but 1 implies 5
- (3, 6) is not a feasible DU path
- It is very difficult to find infeasible paths
- Infeasible paths are a problem
  - Difficult to find
  - Impossible to test



Infeasibility

- Detecting infeasibility can be difficult
  - Combination of elements matter
  - No general way to detect infeasible paths
- In practice the goal is reasonable coverage
  - Number of paths can be large
  - Doing all DU paths might be impractical
- Problems
  - Aliases
  - Infeasible paths
  - Worst case is bad
    - Exponential number of paths
    - Undecidable properties
  - Be pragmatic

Testing Guidelines

- Do it early and do it often
  - Write tests first
  - Best to catch bugs soon, before they hide
  - Automate the process
  - <u>Regression testing</u> will save time
- 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