COMP 150-SEN
Software Engineering Foundations

Testing

Spring 2019

(Some slides from Ben Liblit, UWisc CS 506; Mike Ernst, UW CSE 331)

Introduction

- As software engineers, we build stuff
  - Like any engineering activity, we might make mistakes
  - So, we need to check our work
  - “Optimism is an occupational hazard of programming: feedback is the treatment.” — Kent Beck, *Extreme Programming Explained*
- Bugs in software can have major, real-world consequences
  - For an ongoing list, see Paul G. Neumann, ACM Risks Forum, http://www.csl.sri.com/users/neumann/#3
  - Some famous examples next…

Therac-25 Radiation Therapy Machine

- Massive radiation overdoses killed or seriously injured patients (1985-1987)
  - New design removed hardware interlocks
    - All safety checks done in software
  - Equipment control task not properly synchronized
- Error missed in testing
  - Bug only triggered if operator changed setup too quickly
  - Didn’t happen during testing because operators didn’t have enough practice yet to do this

Mars Polar Lander

- 290kg robotic spacecraft lander launched in 1999
- Lander failed to reestablish communication after descent phase
- Most likely cause: engine shut down too early
  - Legs deployed led to sensor falsely indicating craft had touched down, yet it was 40m above surface
- Error traced to a single line of code
  - Known that leg deployment could lead to a bad sensor reading, but never addressed

Ariane 5 Failure

- In 1996, Ariane 5 launch vehicle failed 39s after liftoff
  - Caused destruction of over $100 million is satellites!
- Cause of failure
  - To save money, inertial reference system (SRC) from Ariane 4 reused in Ariane 5
  - SRI tried to compute a floating point number out of range to an integer; issued error message (as an int); that int was read by the guidance system, causing nozzle to move accordingly
  - The backup system did the same thing
  - Result was rocket moved toward horizontal
  - Vehicle than had to be destroyed
- Ultimate cause: Ariane 5 has more pronounced angle of attack than Ariane 4
  - The out of range value was actually appropriate

Software Bugs Cost Money

- [T]he national cost estimate of an inadequate infrastructure for software testing is $59.5 billion — *The Economic Impacts of Inadequate Infrastructure for Software Testing*, NIST 2002
- Software bugs cost global economy $312 billion per year (Cambridge University, 2013)
- $6 billion loss from 2003 blackout in northeast US
  - Software bug in alarm system in Ohio power control room
- $440 million loss by Knight Capital Group in 30min (2012)
- Economies and lives destroyed by austerity measures based on study linking national debt to slow growth (2010)
  - Spreadsheet calculations riddled with bugs

Software in Buggy

- On average, 1-5 errors per KLoC (kilo-lines-of-code)
  - In mature software
  - >10 bugs per KLoC in prototype software
- Windows 2000
  - 35 million lines of code
  - 63,000 known bugs at release time
  - 2 bugs per KLoC

Software Quality Assurance (QA)

- Testing: run software, look for failures
  - Limits: risk of missing behaviors due to inadequate test set
- Static analysis: assess source code without running it
  - Limits: hard to scale, typically has many false positives
- Program verification: prove program correct
  - Limits: very difficult, very expensive, not scalable
- Code reviews: manual review of program text
  - Limits: informal, uneven, easy to miss issues
- Software process: development/team methodology
  - Limits: one step removed from the code
- …and many more!

No Single QA Approach is Perfect

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

"Program testing can be used to show the presence of bugs, but never to show their absence!" — Edsgar Dijkstra, *Notes on Structured Programming*, 1970

- Most popular QA approach? Testing
  - Static analysis has made huge inroads recently, but is a drop in the bucket compared to testing
  - Verification is on the horizon, but is out of reach for most systems, still

What Can Testing Achieve?

- Make sure code does some of what it is supposed to
- Uncover problems, increase confidence

- Two key rules:
  - Do testing early and often
    - Catch bugs quickly, before they can hide
    - Automate the process if you can
  - Be systematic
    - Have a strategy for testing everything
    - If you thrash about randomly, the bugs will hide in the corner until you're gone

Levels of Testing

- Unit testing: One component at a time
  - A component could be a method, class, or package
  - If test fails, defect localized to small region
  - Done early in software lifecycle, ideally when/before component is developed, and whenever it changes
- Integration/system testing: The whole system together
  - Ensures components work together correctly
  - Possible even if system not complete, as long as there's some end-to-end slice of its functionality
- Other testing terms
  - "Acceptance test" — test system against user requirements
  - "Regression test" — make sure new version of software behaves same as the old version

Automated Unit Testing with JUnit

- xUnit test frameworks for language x
  - Original was SUnit (Smalltalk), by Kent Beck (1989)
  - JUnit popularized the approach
- Easy to build
  - “Never in the annals of software engineering was so much owed by so many to so few lines of code.” — Martin Fowler
- Key: test cases run and checked automatically
  - This means we can run them early and often
- Testing terminology:
  - System Under Test (SUT) — (doesn’t need definition!)
  - Test case — code that runs part of SUT and checks result
    - Test cases can *pass* or *fail*, no gray areas
  - Test suite — a set of test cases

Installing JUnit 4

- Download junit
  - https://search.maven.org/artifact/junit/junit/4.13-beta-2/jar
  - Documentation here: https://junit.org/junit4/
- Downlaod hamcrest
  - https://search.maven.org/artifact/org.hamcrest/hamcrest/2.1/jar
- Add them to your CLASSPATH

```
# bash, both files in $HOME/java
#add the following as a single line to .bash_profile
export CLASSPATH=$HOME/java/junit-4.13-beta-2.jar:
$HOME/java/hamcrest-2.1.jar:.
```

- Test to see if junit is available

```
$ java org.junit.runner.JUnitCore
JUnit version 4.13-beta-2
…
```

Basic JUnit Example

```
# run with “java org.junit.runner.JUnitCore ListTests”
import static org.junit.Assert.*;
import org.junit.*;
import java.util.*;

public class ListTests {
 @Test public void testAdd() {
  List<Object> l = new LinkedList<>();
  Object o = new Object();
  l.add(o);
  assertTrue(“list should contain o”, l.contains(o));
 }
 @Test public void testIsEmpty() {
  List<Object> l = new LinkedList<>();
  assertTrue(“list should be empty”, l.isEmpty());
}}
```

Things to Notice

- A test case in JUnit is just a class
  - Test methods are *annotated* with `@Test`
    - Java annotations begin with `@`, can be examined via reflection
- Each test method has one or more assertions
  - From `org.junit.Assert`
  - `assertTrue`, `assertFalse`, `assertEquals`, `assertNull`, etc
- Running tests shows passes (`.`) and failures (`E`)
  - Failures come with backtrace
  - Test methods run in deterministic but undefined order
    - Make sure success/failure does not depend on ordering!
  - Why does it report the running time?
    - For large projects, running all tests take significant amount of time

Tips for Assertions

- Use `assertEquals` etc rather than `assertTrue`
  - Will get a more useful message if case fails
  - Note: first arg to `assertEquals` is expected value

- Always put messages in assertions

- It might be useful to make your own assertions
  - `assertStringContains(String expected, String s)`
  - `assertAlmostEqual(double expected, double actual, double  delta)`
    - Check `expected-delta ≤ actual ≤ expected+delta`

Tips for Test Cases

- How do you know you've written enough tests
  - We'll talk more about this shortly, but here are some tips
- At least one test per API method (*method coverage)*

```
class ListTests {
  @Test void testAdd() { … }
  @Test void testRemove() { … } …
}
```

- Might want to test corner cases separately

```
// check contains on empty list
@Test void testcontainsEmpty() { … }
```

- Test cases fail if they throw an (uncaught) exception
  - JUnit will catch the exception and keep running other tests

Tips for Test Cases (cont'd)

- Ideally, each test case should check one thing
  - But sometimes okay to break this rule

```
@Test void testContains {
  List l1 = …, l2 = …;
  assertTrue(l1.contains(1));
  assertFalse(l2.contains(1));
}
```

- If test cases catch exceptions, be specific

```
@Test void testRemoveErr() {
  List l1 = …;
  try {
    l1.remove(-1);
    fail("Removed at position -1?!");
  }
  catch (IndexOutOfBoundsException e) { }
}
```

Test Fixtures

- Creating objects per-test can be painful
  - Sometimes, tests need complex web of objects
    - Expensive to reallocate for every test, leads to some duplicate code
- A *test fixture* is an initial set of objects/state of the world for running a set of test cases
  - Test fixtures are “set up” before tests are run
  - They are “torn down” after tests are run
    - E.g., to close files
- JUnit supports four test fixtures annotations
  - `@BeforeClass`, `@AfterClass` — methods to run once per test case class
  - `@Before`, `@After` — methods to run once per test method

Test Fixtures Example

```
class LinkedListTest {
  List<Integer> l; BufferedReader f;

  @BeforeClass void setUp() {
    l = new LinkedList<Integer>();
    l.add(1); l.add(2); l.add(3);
    f = …
  }

  @AfterClass void tearDown() {
    f.close();
} }
```

- Be careful if you mutate fixtures
  - Might use `@Before`/`@After` instead of the `*Class` varieties
- Make sure `tearDown` releases all resources
  - Even in the presence of exceptions

Test Automation

- JUnit tests are completely automated
  - Run from a single command line invocation
  - Test results checked automatically, without human intervention
  - Critically: tests must be repeatable; avoid non-determinism
- Drawback: Adds cost
  - Have to write code and tests together
  - Have to ensure tests and code remain in sync over time
- Major benefits
  - Tests can be run often
  - Code maintenance and evolution becomes much safer
    - Rerunning tests after making a change provides a lot of confidence that the change was correct

Regression Testing

- Key idea: When you find a bug
  - Write a test that exhibits the bug
  - Always run the test when code changes
  - ⇒ ensures bug doesn't reappear
- Helps ensure *forward progress*
  - Ideally, we never go back to old bugs
- Note that automation is key
  - Set of test cases increases over time
  - Without automation, would be too hard to re-execute

Nightly Builds

- Want to run tests as often as possible
  - If bug appears after small code change, easy to attribute bug to that change
  - If bug appears after 1,000 code changes or very big change, tracking down the problem is harder
- Often, too expensive to run all tests on every save
  - Especially as project gets large
- Split tests into two groups
  - *Smoke test* that makes sure nothing is horribly wrong
    - These tests run quickly, not exhaustive
    - Run these all the time
  - Full test suite less often
    - Once per night, once per week, etc

Constructing a Test Suite

- Combine tests from different classes
  - To create set of smoke tests, nightly tests, etc
  - (Example from JUnit documentation)

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  TestFeatureLogin.class,
  TestFeatureLogout.class,
  TestFeatureNavigate.class,
  TestFeatureUpdate.class
})
public class FeatureTestSuite {
  // class is empty, used only for annotations
}
```

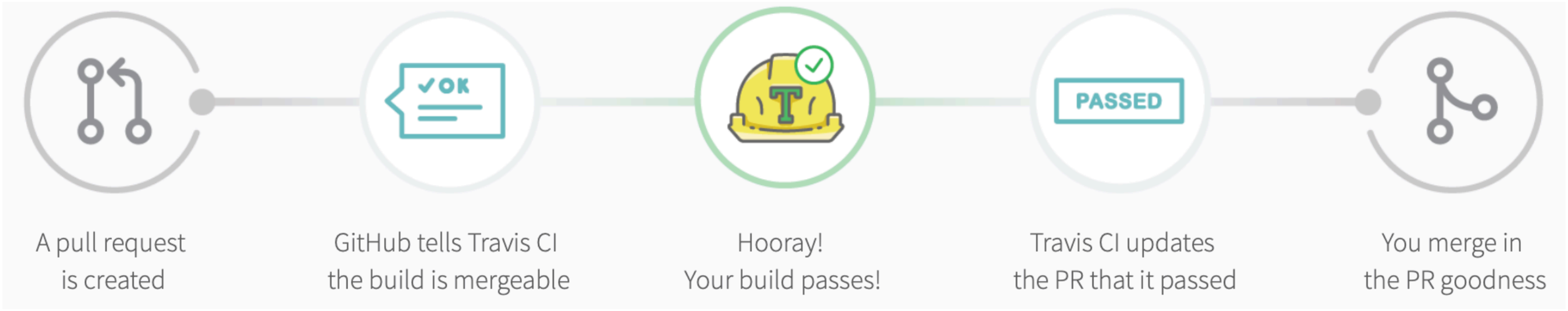
Labeling Tests with Categories

```
public interface TFast { /* category marker */ }

public class A {
 @Test public void a() { … }
 @Category(TFast.class) @Test public void b() { … }
}

@RunWith(Categories.class)
@IncludeCategory(TFast.class)
@SuiteClasses({A.class})
public class FastTestSuite {
 // Will run A.b but not A.a
}
```

- Enables flexible groups of tests

Continuous Integration

- *Continuous integration (CI)* = developers merge changes often
  - Typically by pushing to central version control repository
  - Helps ensure different changes do not conflict
- Creates a natural testing workflow: test before push
  - Helps maintain invariant that main branch tests succeed
- Many CI systems support this model
  - From Travis CI:



Record-and-Replay Testing

- What about testing GUIs?
  - Can unit test individual methods
  - But how do we test clicking buttons etc?
  - Standard approach: record and replay manual tests
- Key challenges
  - Test recording is fragile
    - Either tightly tied to UI or dependent on OS hooks for keyboard/mouse
  - Test replay is fragile
    - Breaks if UI changes
    - If record *(x,y)* coordinates, breaks with different screen layouts etc
  - Note: manual testers would adapt to these conditions

Developing Test Cases

- Now that we know how to run tests, how do we come up with those test cases?
  - This is a hard problem!

- First key question: What properties to test
- Second key question: How to find good test cases

What to Test: Specs. vs. Impls.

- Specifications are almost always partial
  - It's hard to really specify everything
    - E.g., what is the full specification for a web browser?
- Implementations need to make choices
  - Implementations actually run; they can't be partial
  - Often, implementations satisfy a stronger specification

Specification	Implementation
return some value that…	return *smallest* value that…
return a list that…	return a *sorted* list that…
requires `x > 0`	requires `x ≥ 0`
requires `y≠null`	throws exception if `y==null`

Spec. Test vs. Impl. Test

- A *specification test* verifies behavior from the spec
- An *implementation test* verifies the additional behavior of the implementation
  - Often, one person's implementation detail is another person's specification
  - (Mostly because specifications are often not written down very precisely)

- In practice, we'll use both kinds of tests
  - Specification tests for the basic functionality
  - Implementation tests for other details that seem important
    - Don't over-constrain impl. tests; might prevent future improvements
    - E.g., for most applications, don't check timestamps in tests

Finding Good Tests: Sqrt

```
// requires: x ≥ 0
// returns: ret such that ret*ret == x, approximately
public double sqrt(double x) { … }
```

- What values of `x` might be worth testing?
  - `x < 0` — exception thrown
  - `x ≥ 0` — returns normally
  - `x` near `0` — boundary condition
  - perfect squares
  - non-perfect squares
  - …?

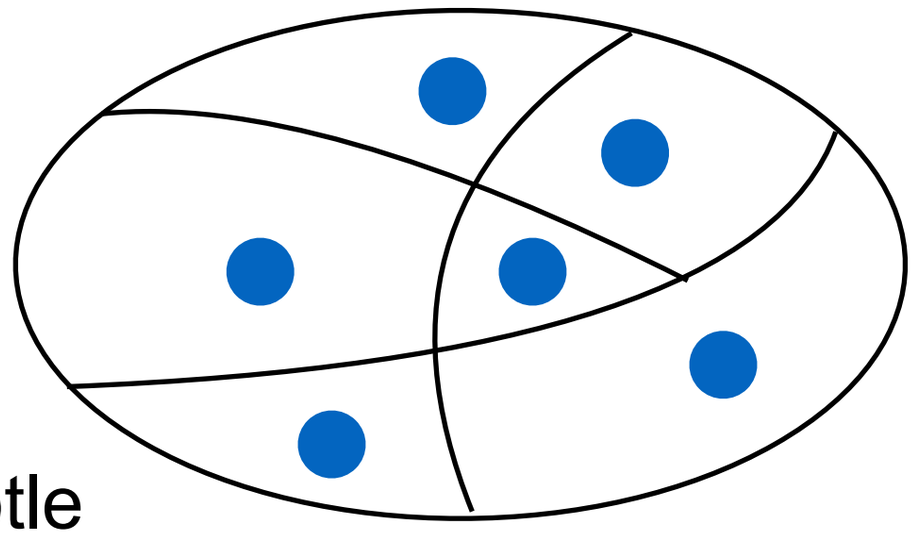
Why is Testing Hard?

```
// requires: 1 ≤ x,y,z ≤ 10000
// returns: …
public int f(int x, int y, int z) { … }
```

- Exhaustive testing would require 10^12 runs
  - Completely impractical
- Key problem: choosing a test suite
  - Small enough to finish quickly
  - Large enough to validate the program
    - Each actual, concrete test must represent many other tests
    - I.e., if that test passes, many other tests would also pass

Partitioning Input Space

- Ideal test suite:
  - Identify tests with same behavior
  - Try one input from each set
- Two key problems
  - Notion of *the same behavior* is subtle
    - Naive approach: execution equivalence
      - Program takes same sequence of steps
    - Better approach: *revealing subdomains*
  - Discovering the sets requires perfect knowledge
    - Use heuristics instead



Execution Equivalence

```
// requires: if x<0 then returns -x
//                otherwise returns x
public int abs(int x) {
  if (x < 0) { return -x; }
  else { return x; }
}
```

- All `x<0` runs are execution equivalent
  - Method takes same sequence of steps
- All `x≥0` runs are execution equivalent

- So, maybe we need two test inputs
  - {-3, 3} might be a good test suite

Execution Equivalence Insufficient

```
// requires: if x<0 then returns -x
//                otherwise returns x
public int abs(int x) {
  if (x < -2) { return -x; }
  else { return x; }
}
```

- All `x<-2` runs execution equivalent
- All `x≥-2` runs are execution equivalent
  - So is {-3, 3} a good test suite? No!
- Problem: we didn't consider the spec!
  - `x<-2` — okay
  - `x≥0` — okay
  - `x=-2` or `x=-1` — bug — not covered in test suite

Better: Revealing Subdomains

- A *subdomain* is a set of possible inputs

- A subdomain is *revealing for error E* if either
  - *Every* input in that subdomain triggers E or
  - *No* input in that subdomain triggers E

- Need to test only one input from each revealing subdomain
  - If revealing subdomains cover entire input space, guaranteed to detect error if present

Example Revealing Subdomains

```
// requires: if x<0 then returns -x
//                otherwise returns x
public int abs(int x) {
  if (x < -2) { return -x; }
  else { return x; }
}
```

- Possible revealing subdomains
  - … {-2} {-1} {0} {1} {2} … — too many tests
  - {…, -4, -3} {-2, -1} {0, 1, …} — minimal number of tests
  - …, {-6, -5, -4} {-3, -2, -1} {0, 1, 2} — not all revealing, specifically, {-3, -2, -1} is not a revealing subdomain

Developing Good Tests

- Finding revealing subdomains is quite difficult!
  - They depend on both the implementation and the spec
  - To find them, we need perfect knowledge about the program and its errors…but if we had perfect knowledge, we wouldn't need to test the program!
- In practice, we use heuristics
  - Heuristics for designing test suites
  - = Heuristics for choosing inputs
  - = Heuristics for dividing the domain
- Good heuristics give
  - Few subdomains
  - High probability that some subdomain is revealing

Black Box Testing

- Look only at specification, not at code

Consider Each Path in Spec

- Look at the spec and consider conditional branches

```
// Return true if x in a, else return false
boolean contains(int[] a, int x);
```

  - Two "paths" through spec
    - One test where `x` in `a`, one test case where `x` not in `a`
    - Maybe another one: what if `x` appears twice in `a`?

```
// Return maximum of a and b
int max(int a, int b)
```

  - Three paths through spec
    - if `a<b` returns `b`; if `a>b` — returns `a`; if `a=b` — returns `a`

- In all cases, actual tests will need concrete values
  - E.g., test `max` with `(3, 4)` to cover first case

Test Boundary Conditions

- Create tests at the edges of the “main” subdomains to look for
  - Off-by-one errors
  - Forgetting to handle empty container
  - Forgetting to handle null
  - Arithmetic overflow
  - Aliasing
- Experience suggests such subdomains have a high probability of revealing bugs
  - Also, you might have mis-drawn the boundaries

Arithmetic Overflow

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

- What are some good values/ranges? to test
  - `x<0` (flips sign) or `x≥0` (returns unchanged)
  - around `x=0` (boundary condition)
    - Specific tests might be `x=-1`, `x=0`, `x=1`
- What about the following:

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

  - Docs: “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.”

Duplicates and Aliasing

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

- What happens if `src` and `dst` are same object?
  - This is *aliasing* and it's easy to forget! Watch out for this
- Other useful cases (for other methods)
  - `null`
  - Circular lists

Finding Boundaries

- Two values are *adjacent* if they are one operation apart
  - Example: list of integers
    - [2,3] is adjacent to [2,3,4]
    - [2,3] is adjacent to [2]

- A value is on a *boundary* if either
  - There exists an adjacent point in a different subdomain
  - Some basic operation cannot be applied to the point
    - [] — can’t apply remove

Boundary Value Example

```
interface List {
  // Inserts elt at position index in the list. Shifts
  // the elt currently at that position (if any) and
  // any subsequent elts to the right (adds one to
  // their indices)

  public void add(int index, Object elt);
}
```

- Test with empty list
- Test with index and first/last elt
- Others?

Black Box Testing Advantages

- Process not influenced by tested component
  - Code's assumptions not propagated to test suite
  - Tests are all about using redundancy / redundency to find mistakes
  - To create useful redundancy, avoid strict duplication
- Robust with respect to implementation changes
  - Shouldn't need to change black box tests when code changed
- Allows testers to be independent
  - Testers need not be familiar with code
  - Tests can be developed before writing code

Clear Box Testing

- Look at implementation
  - (= Glass Box Testing = White Box Testing)
- Focus on features not described by specification
  - Control-flow details
  - Performance optimizations
  - Alternate algorithms for different cases

Clear Box Motivation

```
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];
} }
```

- Subdomain boundary at `x=CACHE_SIZE`
  - Not apparent from specification

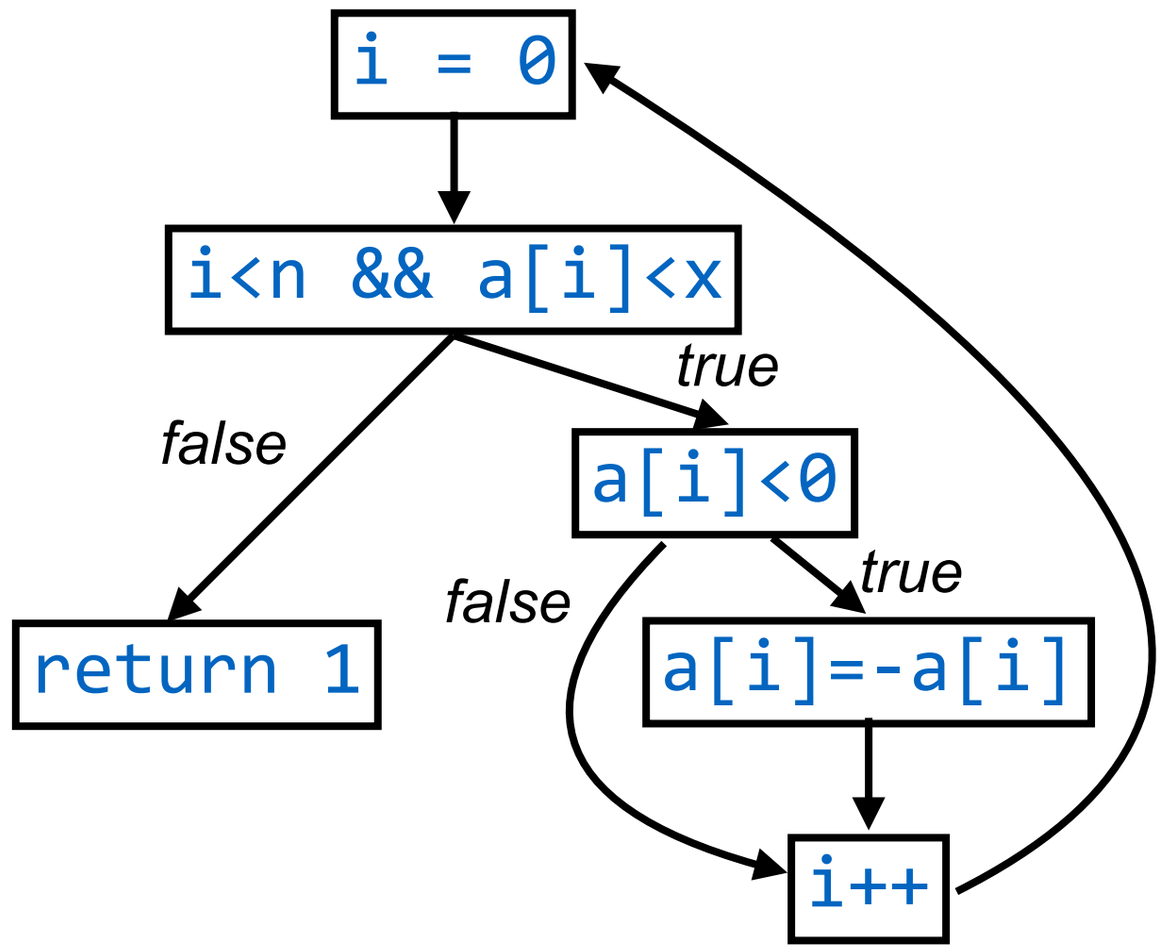
Coverage Criteria

- Common metric for test suite quality: *coverage*
  - Goal: test suite covers all possible program behaviors
  - Assumption: high coverage ⇒ few mistakes remain in program
    - Certainly, if test suite doesn't cover some behavior, we aren't checking if there is a bug in it or not
- But what is a behavior? Probably not measurable
- Instead: *Structural coverage testing*
  - Divide a program into elements (e.g., statements)
  - Coverage of a test suite is

$$\frac{\text{\# of elements executed by suite}}{\text{\# elements in program}}$$

Statement Coverage

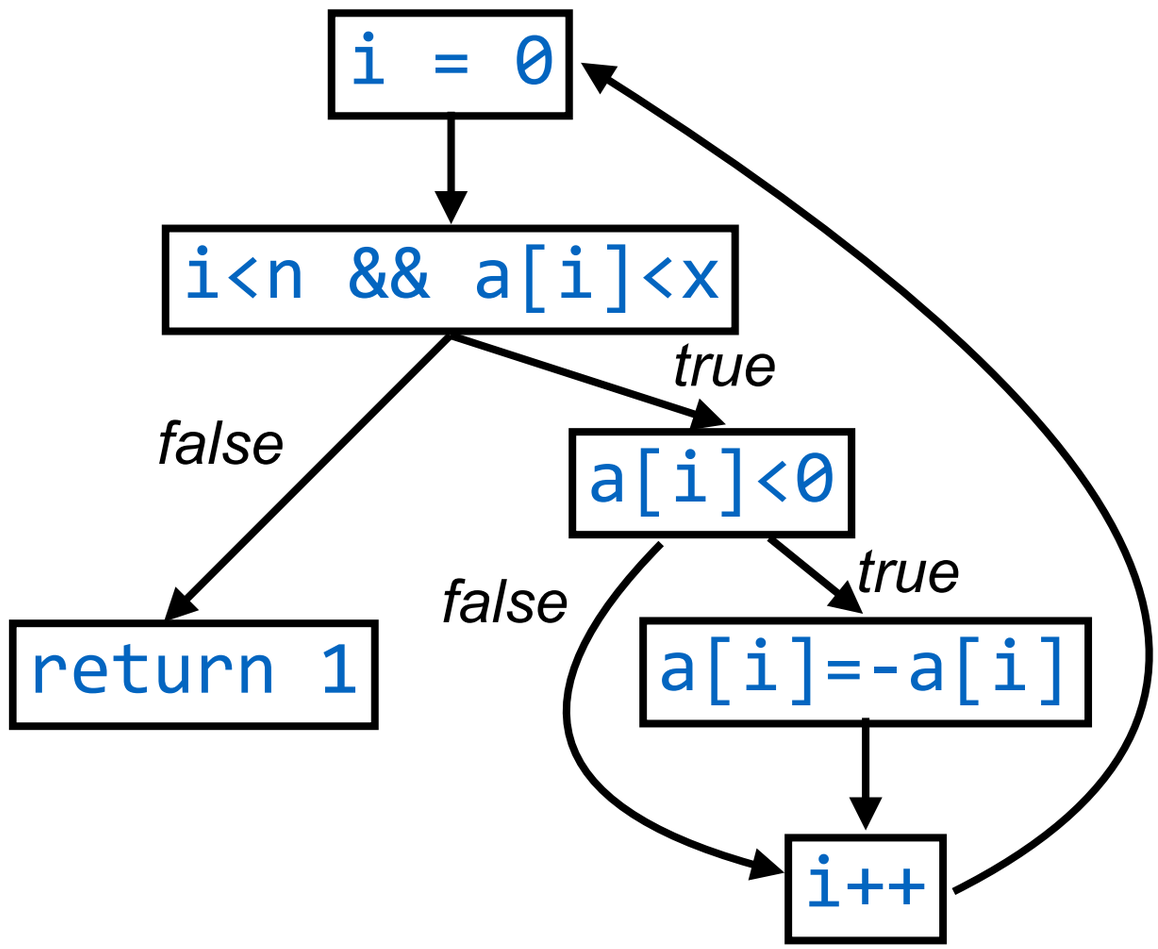
```
int select(int[] a,
           int n, int x) {
 int i=0;
 while (i<n && a[i]<x) {
  if (a[i]<0) {
   a[i]=-a[i];
  }
  i++;
 }
 return 1
}
```



- Consider test `(n=1 a[0]=-7 x=9)`
  - Covers all statements
  - But, doesn't consider case where `a[i]<0`

Condition Coverage

```
int select(int[] a,
           int n, int x) {
 int i=0;
 while (i<n && a[i]<x) {
  if (a[i]<0) {
   a[i]=-a[i];
  }
  i++;
 }
 return 1
}
```



- Add test `(n=1 a[0]=7 x=0)`
  - Covers all branches (all edges in the graph)
  - But, for `i<n&&a[i]<x`, has cases where `i<n`, `i≥n`, `a[i]<x`, but no case where `a[i]≥x` is checked
  - I.e., the branches due to short-circuiting are not covered

Path Coverage

- Execute every path through the program
  - Challenge 1: Which paths are *realizable*, i.e., could occur at runtime
    - Often not obvious from looking at the program text
    - So it's hard to know how many of the possible paths have been covered
  - Challenge 2: Acyclic programs can have exponential number of paths
    - `if(…){…} else {…}; if(…){…} else {…}; if(…){…} else {…};` has eight paths
  - Challenge 3: Programs with loops might have an unbounded number of paths
    - E.g., a program that reads data from the network and processes it in a loop
  - ⇒ Path coverage is not a common metric

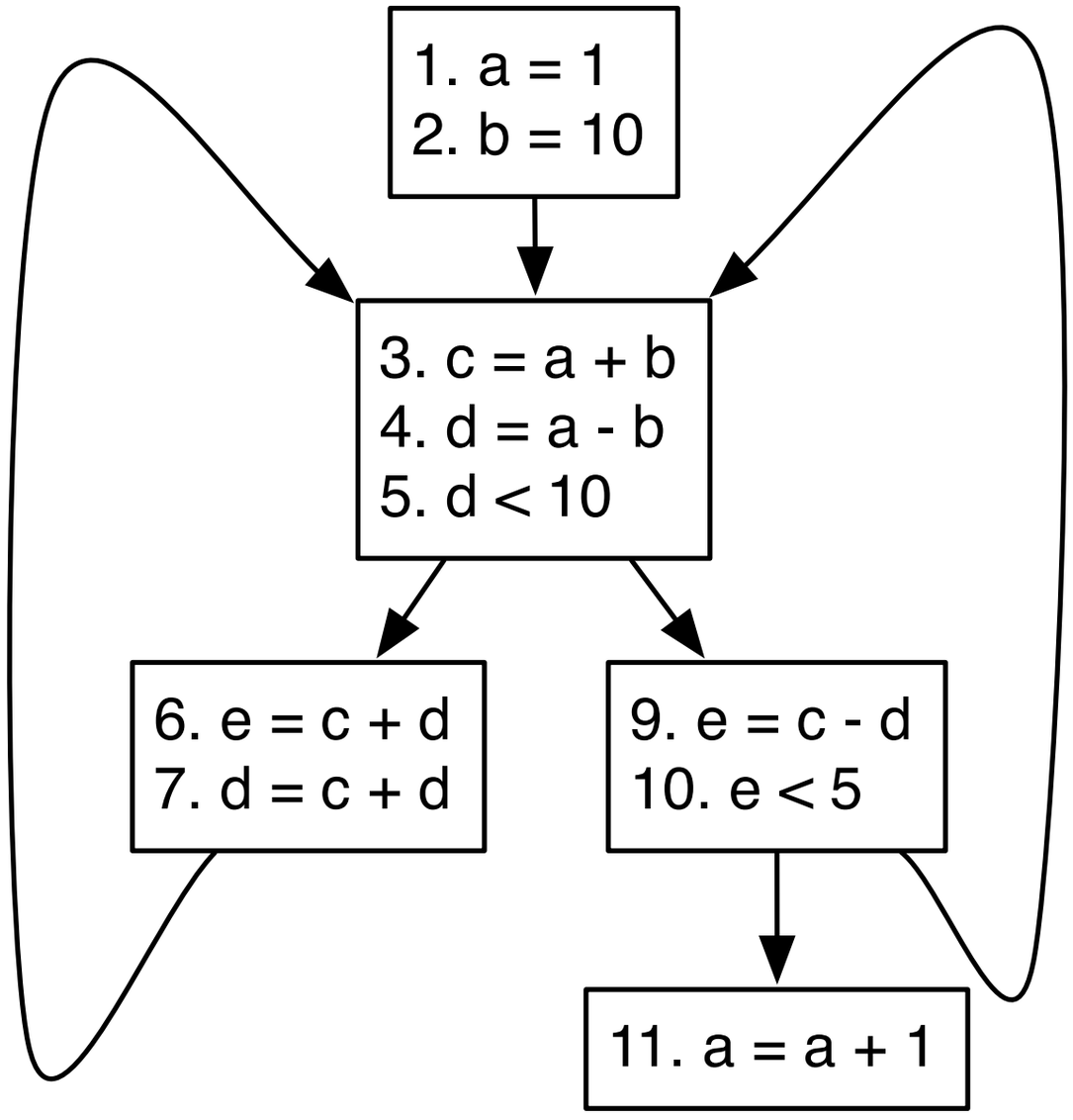
Basic Blocks and CFGs

- A *basic block* is a sequence of three-addr code with
  - (a) no jumps from it except the last statement
  - (b) no jumps into the middle of the basic block

- A *control flow graph* (CFG) is a graphical representation of the basic blocks of a three-address program
  - Nodes are basic blocks
  - Edges represent jump from one basic block to another
    - Conditional branches identify true/false cases either by convention (e.g., all left branches true, all right branches false) or by labeling edges with true/false condition

Example

```
1.  a = 1
2.  b = 10
3.  c = a + b
4.  d = a - b
5.  if (d < 10) goto 9
6.  e = c + d
7.  d = c + d
8.  goto 3
9.  e = c - d
10. if (e < 5) goto 3
11. a = a + 1
```



Def-Use Pairs

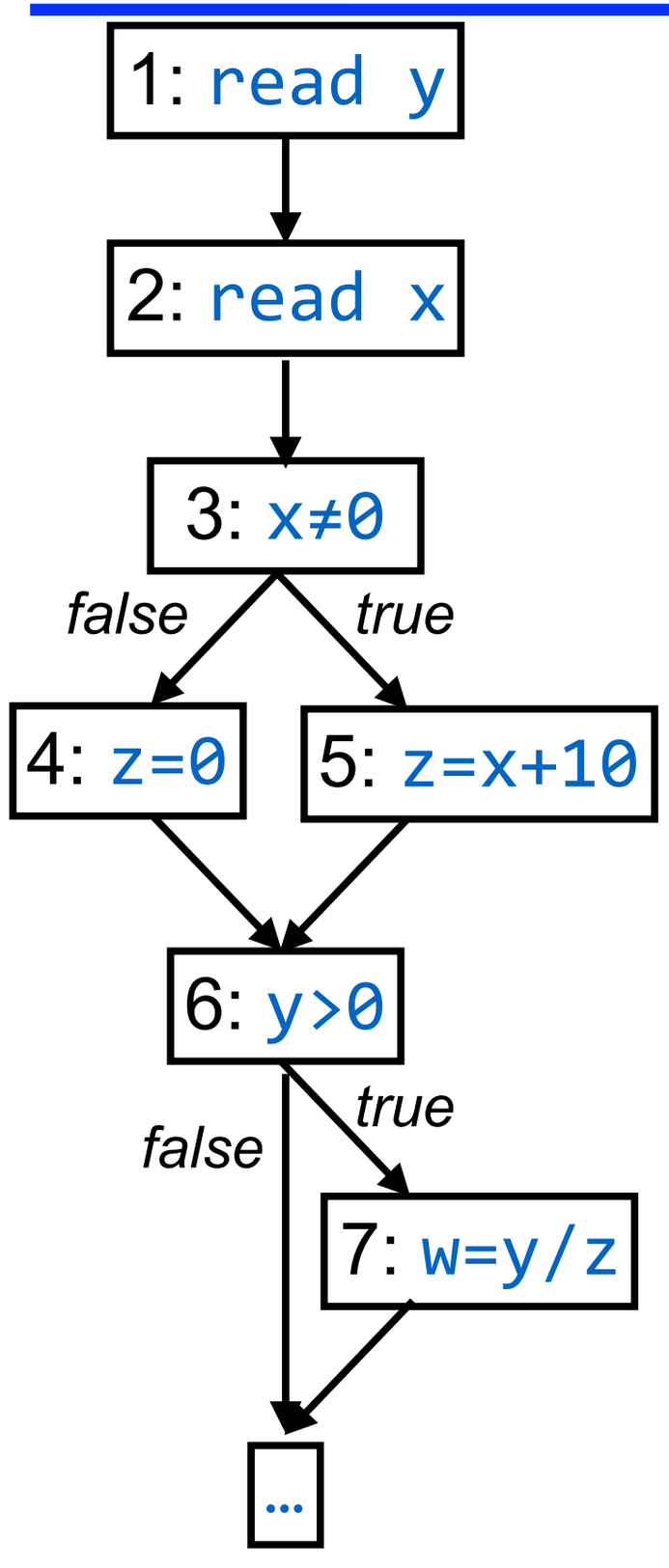
- A *definition (def)* of a variable is an assignment to it
  - `x=3` is a def of `x`
- A *use* of a variable is a read of it
  - `y=x+z` is a def of `y` and a use of `x` and `z`
- A def is *paired* with a use when the value assigned by the def can flow to the use in some execution

```
x=3;   // def (1)
y=x+2; // use of (1)
```

```
x=3; // def (1)
x=4; // def (2)
y=x+2;  // use of (2)
        // not a use of (1)
```

```
x=3; // def (1)
if (…) {
  y=x+2; // use of (1)
} else {
  z=x+2; // use of (1)
}
```

Data Flow Coverage



- Cover all def-use pairs
  - Possible pairs:
    - (1,6), (1,7)
    - (2,3), (2,5)
    - (4,7)
    - (5,7)
  - Test cases
    - `x=1 y=22`
      - Covers (1,6), (1,7), (2,3), (2,5), (5,7)
    - `x=0 y=10`
      - Covers (1,6), (1,7), (2,3), (4,7)
    - Combination gives full data flow coverage

Code Coverage Limitations

- Code coverage has proven value
  - Can help identify weak test suites
    - Test suites that lack coverage are probably inadequate in other ways
  - Tricky code with low coverage is a danger sign
- But, 100% coverage does not mean no bugs
  - And, 100% coverage almost never achieved
- Reality: time and budget is limited
  - Should we spend money testing code or adding new features that our customers want?
  - Where should we direct testing effort?
    - “High risk” code (= bugs could cause severe damage) is a good target!

In Practice…

- Statement coverage is most common criterion used
  - Many coverage tools provide basic block coverage
    - I.e., not quite statement information
- Branch coverage is done in terms of CFG edges
  - Assuming the CFG expands out && and || into more branches, there's no issue with branch vs. condition coverage
- Full coverage not often achieved
  - Common to reach 85% coverage
  - Safety-critical software should get 100% statement coverage (feasible)
  - Are remaining statements unreachable (dead code)? Just hard to get to? Hard to know for sure.

Two Rules of Testing

1. Test early and often
   - Best to catch bugs as soon as possible
   - Automate the process
   - Regression testing will save you time
2. Be systematic
   - If you test at random, bugs will hide where you don’t test
   - Writing tests is a good way to understand the spec
   - The spec can be buggy too!
   - When you find a bug, write a test for it, show that the test fails, then fix the bug

Summary

- Testing matters
  - You need to convince others that your code works
- Testing can help catch problems early
  - If you make a small change and a test case files, usually much easier to understand what happened
- Learn to use code coverage tools for your language
  - These are common, mature tools worth learning
- Don’t confuse *volume* of tests with *quality* of tests
  - Can get in the way of systematic testing
- Choose test data to cover (black box, clear box)
- Testing can’t prove the absence o f bugs
  - But it can increase quality and confidence