More on Software Testing

CSE 403 Software Engineering
Winter 2025

Today's outline

Software testing
- White box testing
  - Code coverage
  - Mutations
- Integration testing

Teammate survey - see Ed Chat for your link (required)
- **due today by 11:59pm**

Guest industry speaker this Wednesday, Zach Sperske, Affirm
- survey (your takeaway) (required) **due after-class Wednesday**

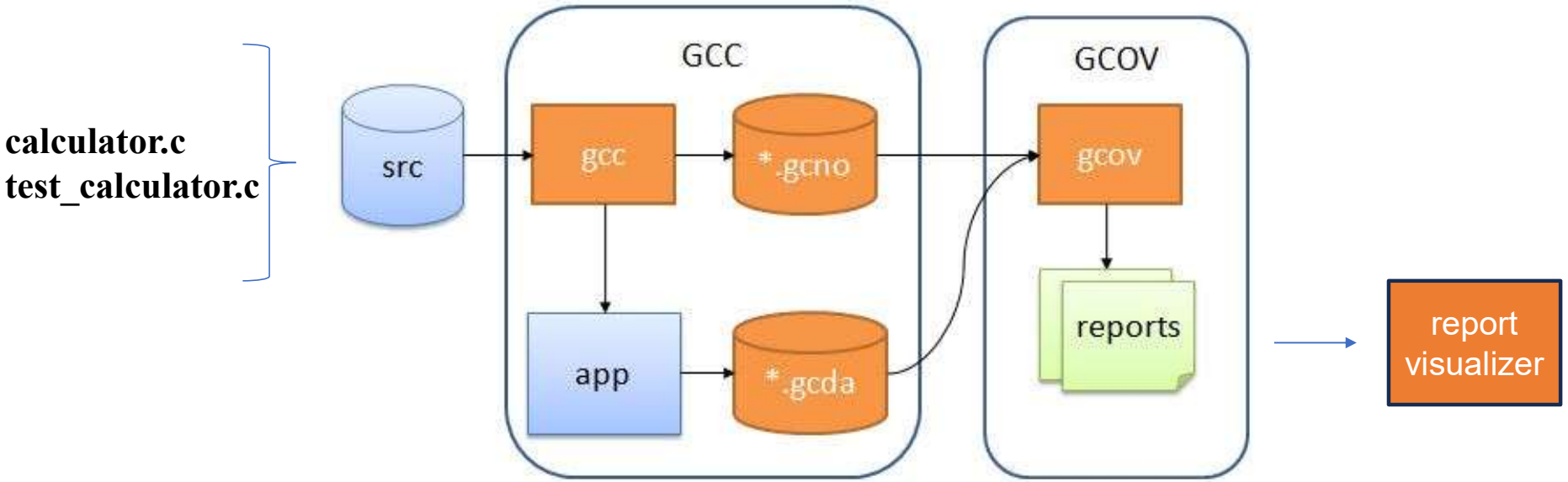
Watch Ed and the Calendar for class updates!

Jumping into a demo – calculator module

Scenario

- You've inherited responsibility for some code
- There is a test suite! Woohoo!
- But you don't know how well the tests cover the code / how adequate they are
- So you'll run **coverage** analysis to provide some insights

GNU's gcov is an available option

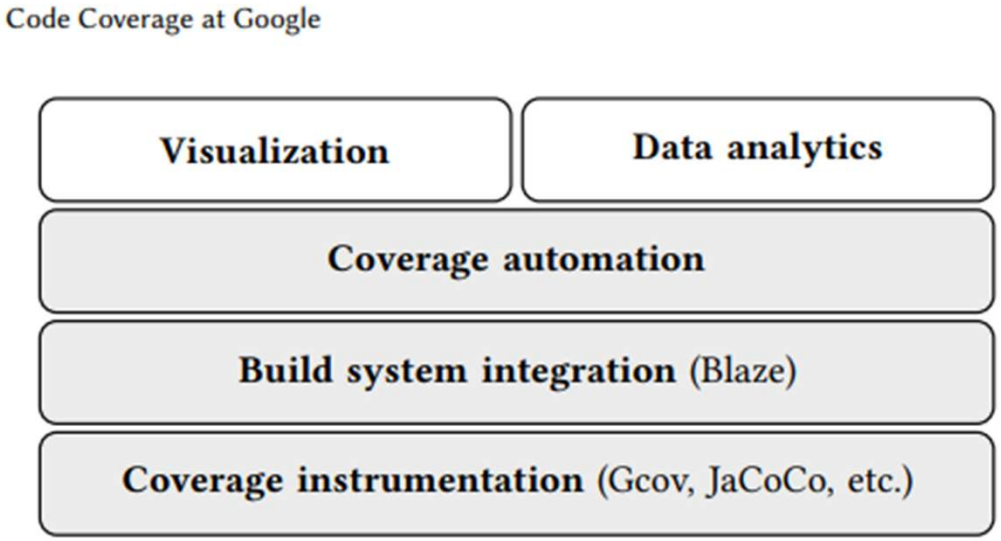


Intro to gcov demo

Link it to your CI automation

Consult as part of your testing process and code reviews, too

Code coverage at Google



Layered architecture!

Visualization tools are built on top of code instrumentation tools

Read: https://homes.cs.washington.edu/~rjust/publ/google_coverage_fse_2019.pdf

Back to basics: code coverage metrics

Code coverage testing: examines what fraction of the code under test is reached by existing unit tests

Structural code coverage metrics include:

- Statement coverage (what we looked at with gcov)
- Condition coverage
- Decision coverage

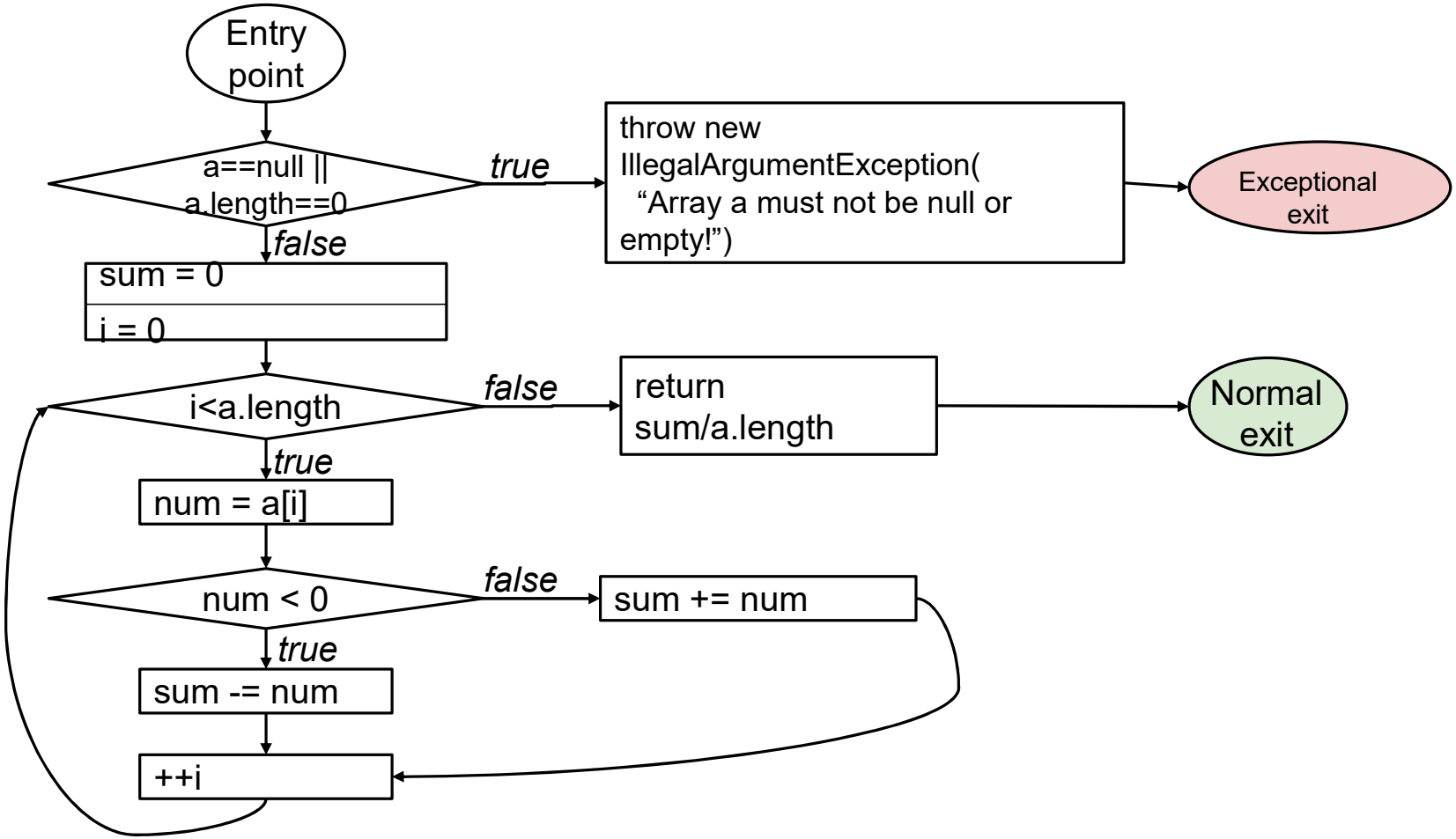
Which type of coverage requires the most tests?

Code coverage: the basics

Average of the absolute values of an array of doubles

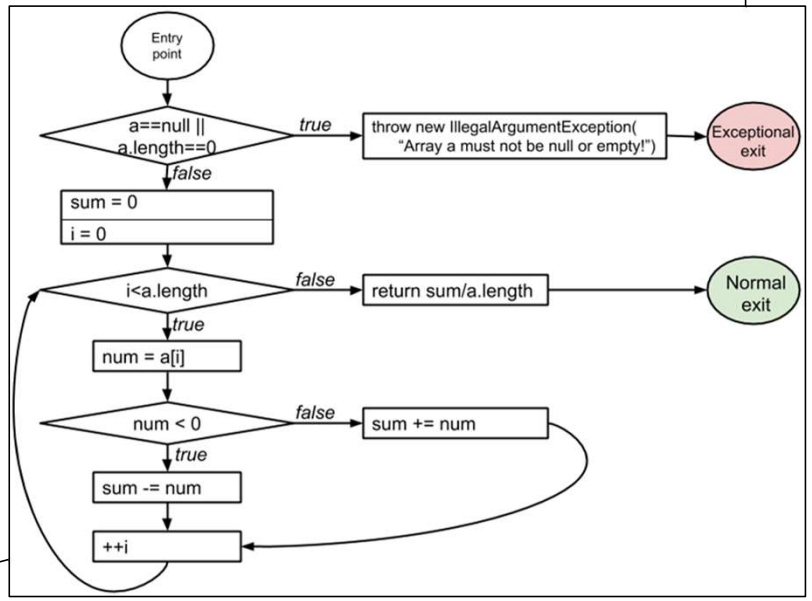
```
public double avgAbs(double ... numbers) {

  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
    throw new IllegalArgumentException(“Nums cannot be null or empty!");
  }

  double sum = 0;
  for (int i=0; i<numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) {
      sum -= d;
    } else {
      sum += d;
    }
  }
  return sum/numbers.length;
}
```

Create the control flow graph



And align the two to help identify tests

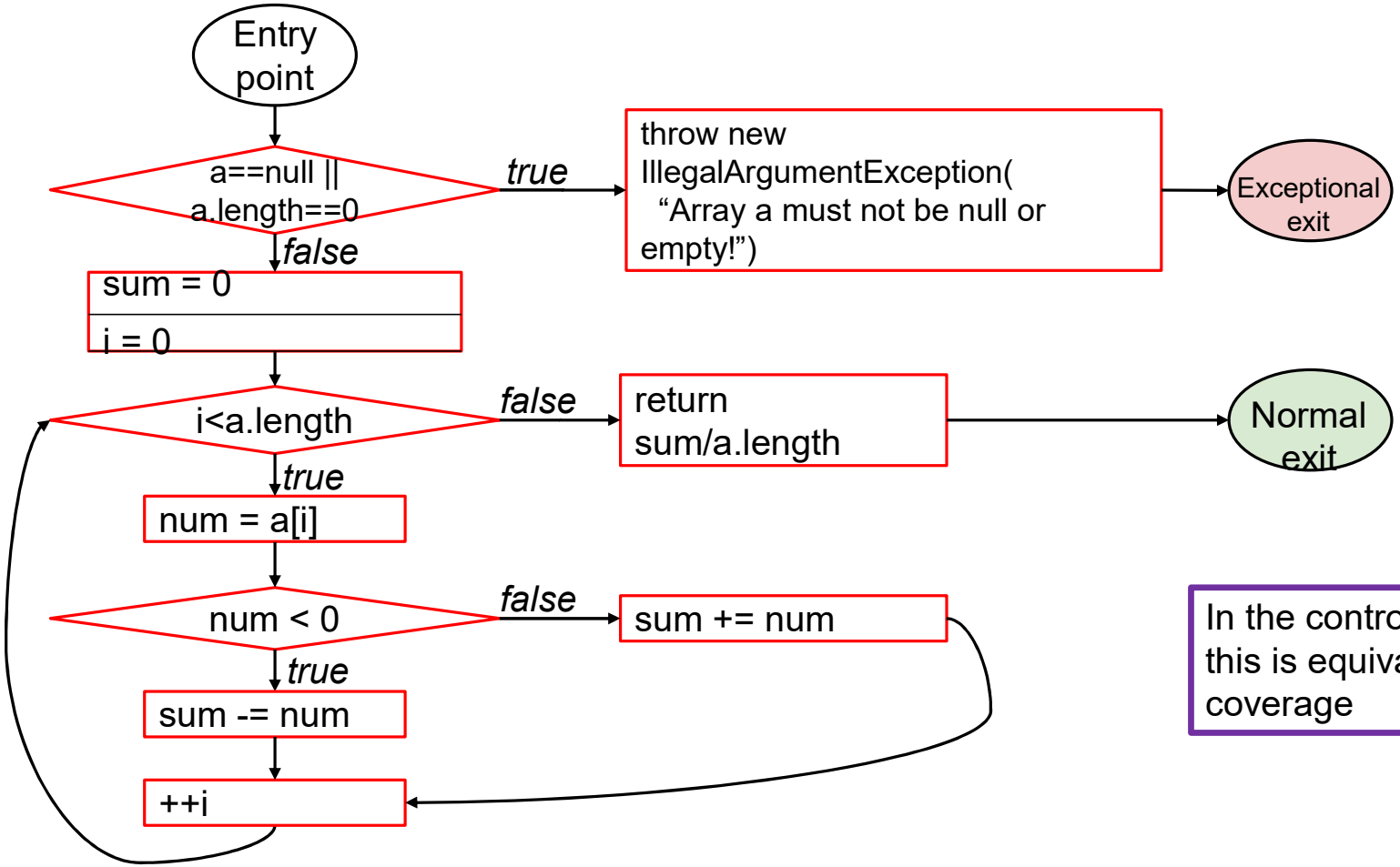
```
public double avgAbs(double ... numbers) {

  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
    throw new IllegalArgumentException("Numbers must not be null or empty!");
  }

  double sum = 0;
  for (int i=0; i<numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) {
      sum -= d;
    } else {
      sum += d;
    }
  }
  return sum/numbers.length;
}
```



Statement coverage

Every statement in the program must be **executed at least once** by the tests

Statement coverage



In the control flow graph, this is equivalent to **node** coverage

Condition and decision coverage

Condition: a boolean expression that cannot be decomposed into simpler boolean expressions (e.g., an atomic boolean expression)

Decision: a boolean expression that is composed of conditions, using 0 or more logical connectors (a decision with 0 logical connectors is a condition)

Quiz:

```
If (a | b) { …}
```

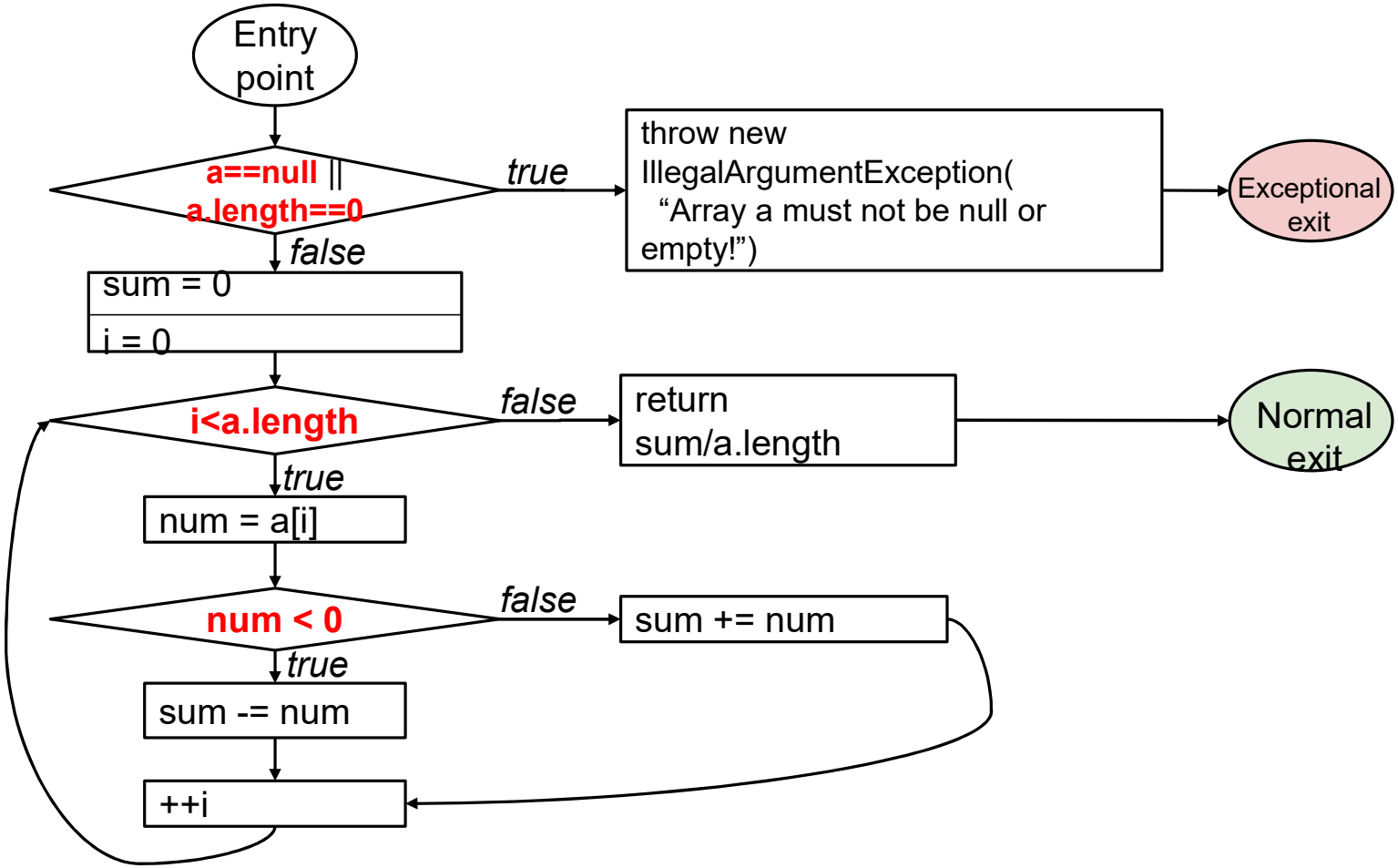
What are a and b?
What is the boolean expression (a | b)?

Condition coverage

Condition: a boolean expression that cannot be decomposed into simpler boolean expressions (atomic)

Condition coverage: every condition in the program must take on all possible **outcomes (true/false) at least once**

Condition coverage

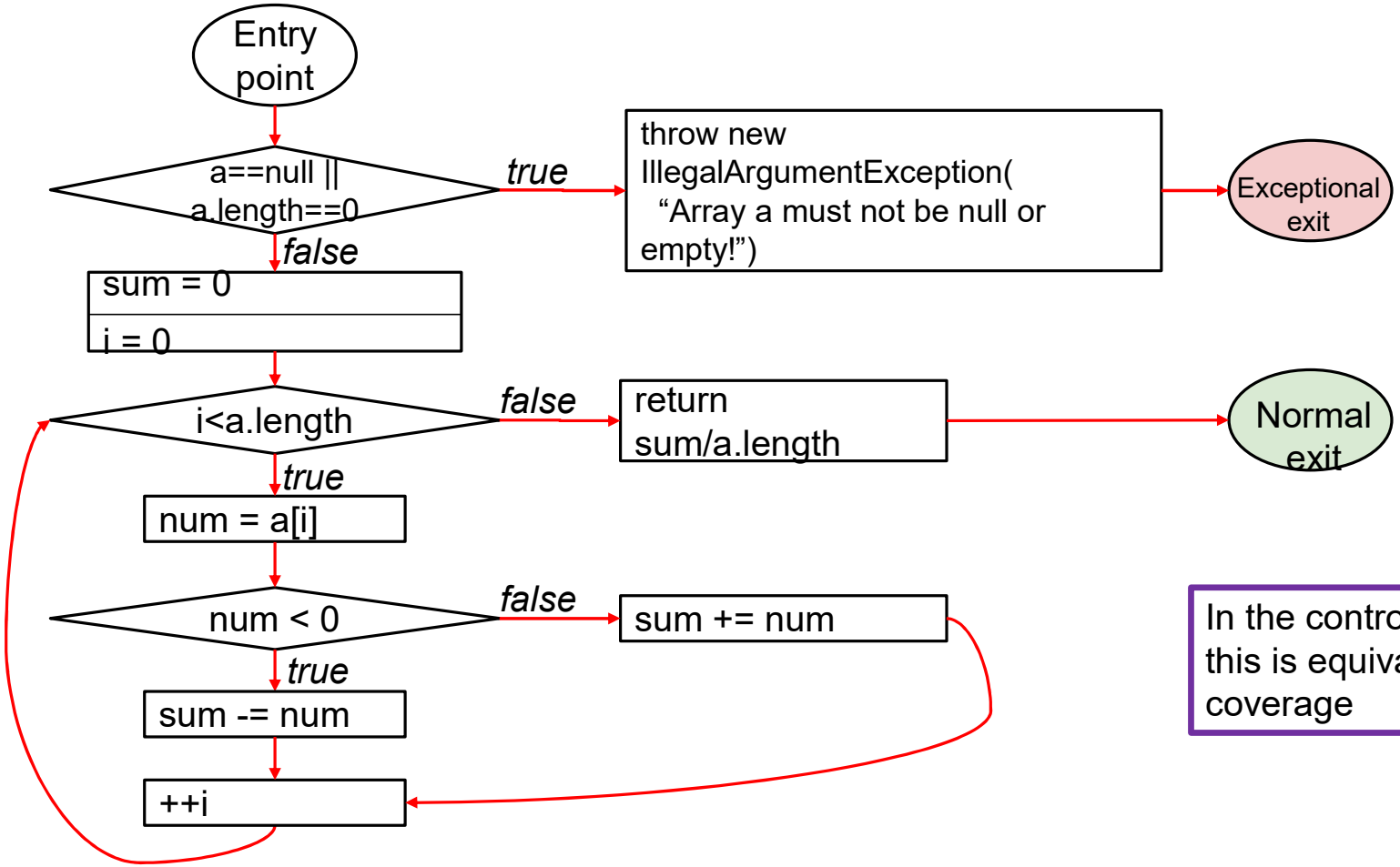


Decision coverage

Decision: a boolean expression that is composed of conditions, using 0 or more logical connectors

Decision coverage: every decision in the program must take on all possible **outcomes (true/false) at least once**

Decision coverage



In the control flow graph, this is equivalent to **edge** coverage

There is a concept of "subsumption"

Given two coverage metrics A and B,
A subsumes B if and only if **satisfying A implies satisfying B**

- Subsumption relationships (true or false):
  1. Does **statement** coverage subsume **decision** coverage?
  2. Does **decision** coverage subsume **statement** coverage?
  3. Does **decision** coverage subsume **condition** coverage?
  4. Does **condition** coverage subsume **decision** coverage?

https://pollev.com/cse403wi

Code Coverage - Do coverage types subsume each other

0 surveys completed

0 surveys underway

Title logo

Does statement coverage subsume decision coverage?

Yes

No

Title logo

Does decision coverage subsume statement coverage?

Yes

No

Title logo

Does decision coverage subsume condition coverage?

Yes

No

Title logo

Does condition coverage subsume decision coverage?

Yes

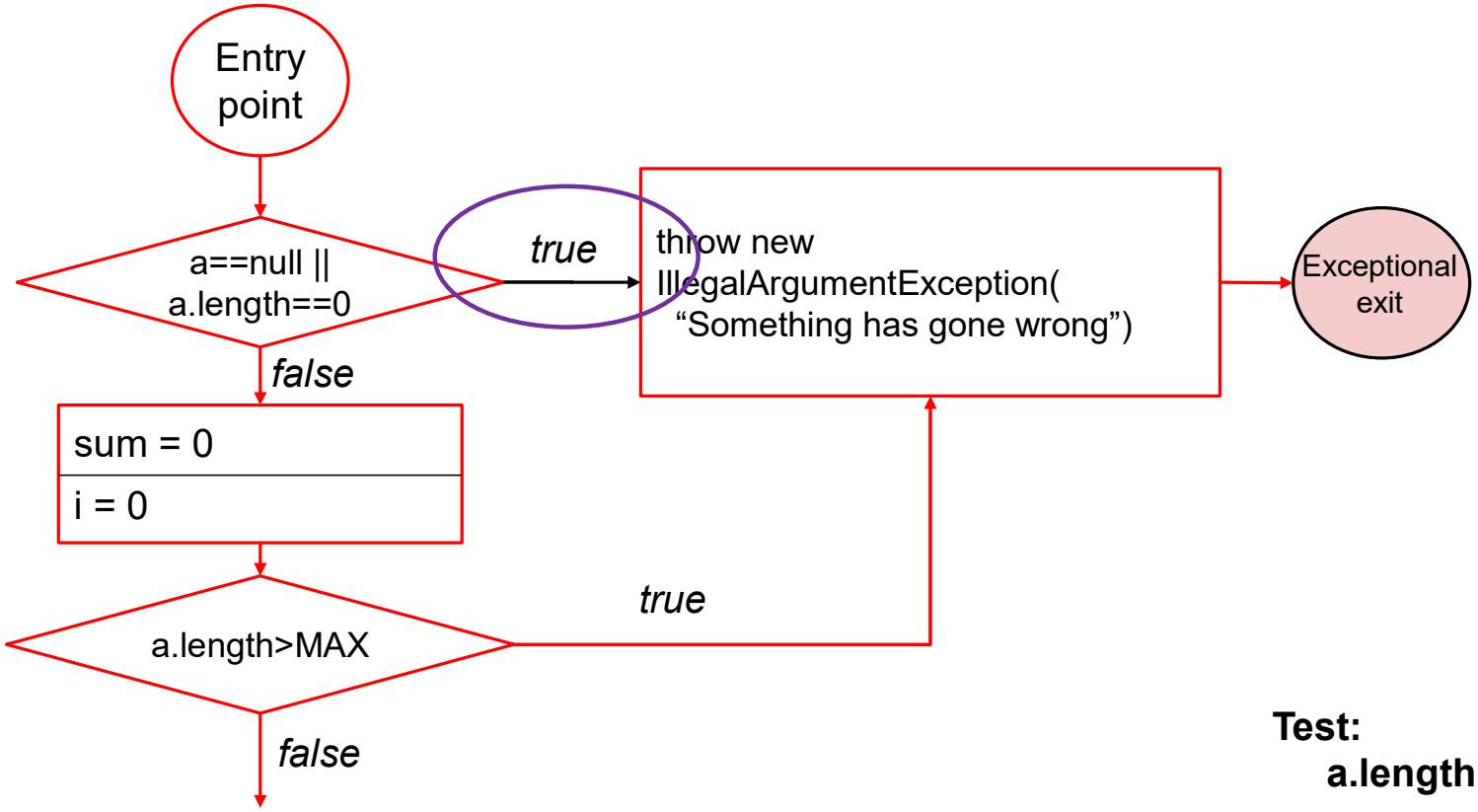
No

And the experts say…

Given two coverage criteria A and B,
A subsumes B iff **satisfying A implies satisfying B**

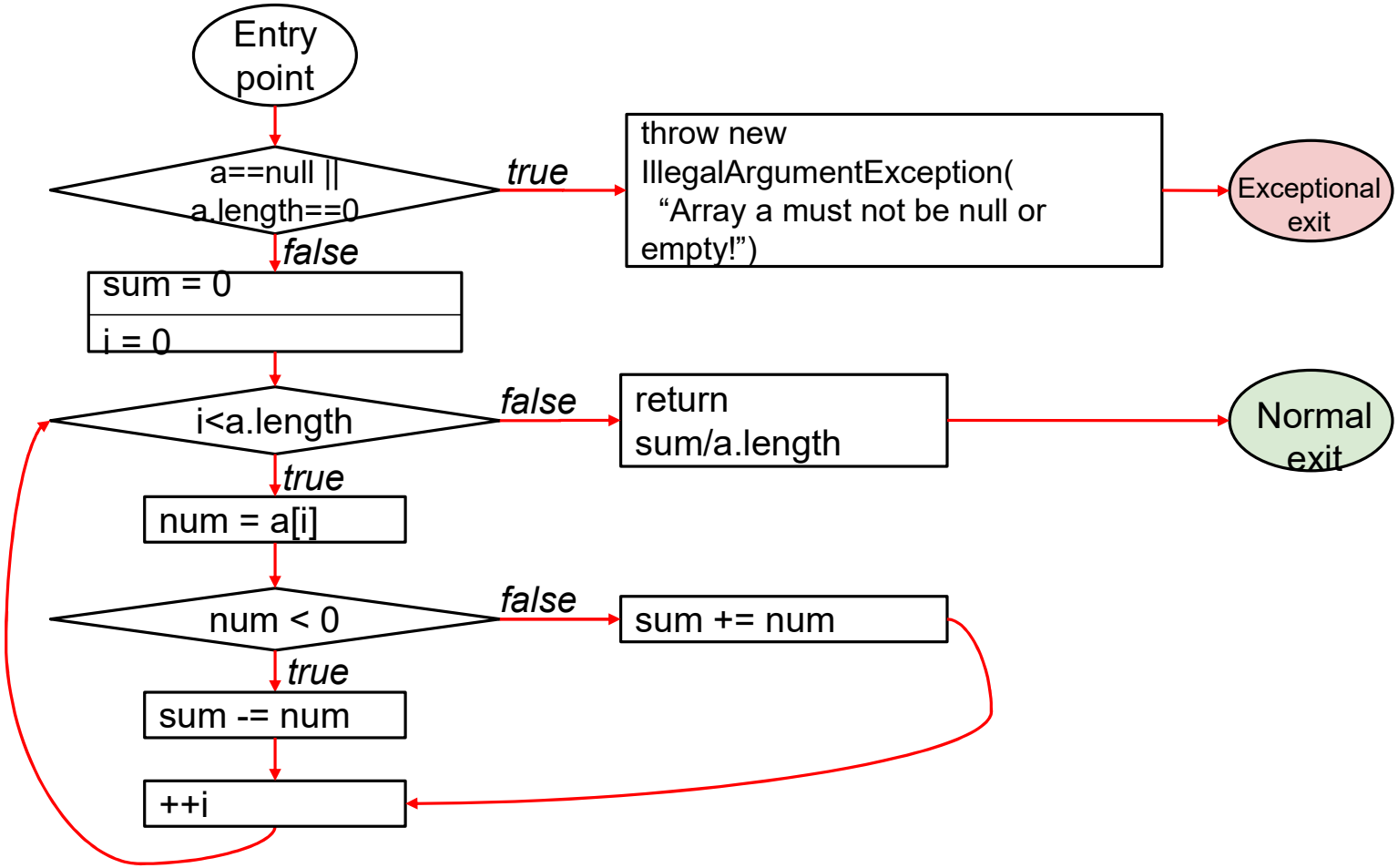
- Subsumption relationships :
  1. **Statement** coverage **does not** subsume **decision** coverage
  2. **Decision coverage subsumes statement coverage**
  3. **Decision** coverage **does not** subsume **condition** coverage
  4. **Condition** coverage **does not** subsume **decision** coverage

Statement does not subsume Decision coverage



Test:
a.length = MAX+1

Decision subsumes Statement coverage



Decision and Condition – neither subsumes the other

4 possible tests for the decision:

```
If (a | b) { …}
```

1. a = 0, b = 0
2. a = 0, b = 1
3. a = 1, b = 0
4. a = 1, b = 1

a	b	a | b
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	1

These two satisfy **condition coverage** but **not decision coverage**

a	b	a | b
0	**0**	**0**
0	**1**	**1**
1	0	1
1	1	1

These two satisfy **decision coverage** but **not condition coverage**

How much coverage is enough? 100%?

May be subject to the law of diminishing returns … shoot for 80%

Good resource on code coverage and code coverage tools:
https://www.atlassian.com/continuous-delivery/software-testing/code-coverage
And a good list of coverage tools:
https://www.browserstack.com/guide/code-coverage-tools

Code coverage takeaways

- Code coverage can provide valuable insights into your code and into your testing adequacy
- It is intuitive to interpret
- There are great tools available to help compute code coverage of your tests
- Code coverage itself is not sufficient to ensure correctness
- Code coverage is well known and used in industry

Next up -

Testing with mutations

You'll practice this on Friday with an in-class exercise

Mutation testing

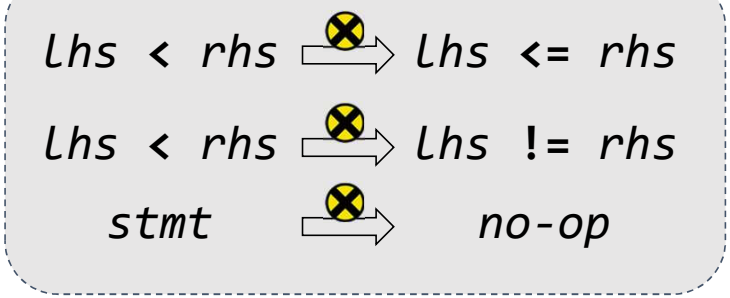
Program

Mutation testing

Mutation testing: mutant generation

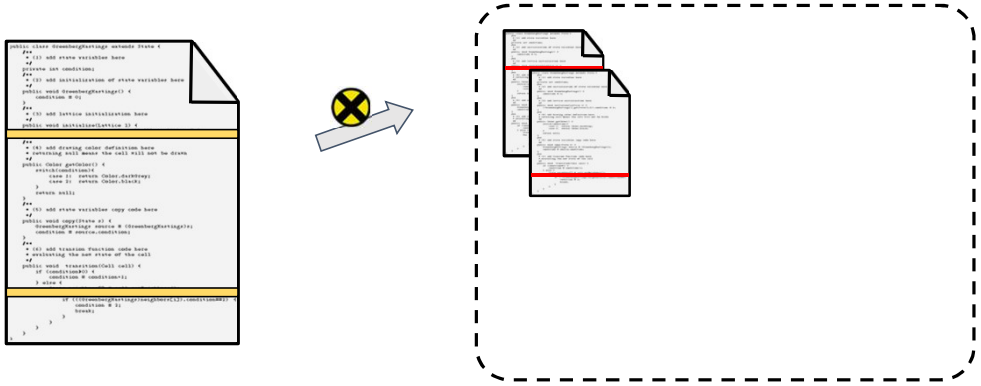
Program

Mutation testing



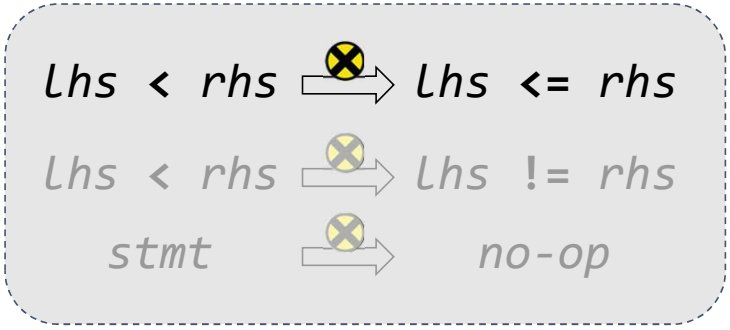
Mutation operators

Mutation testing: mutant generation



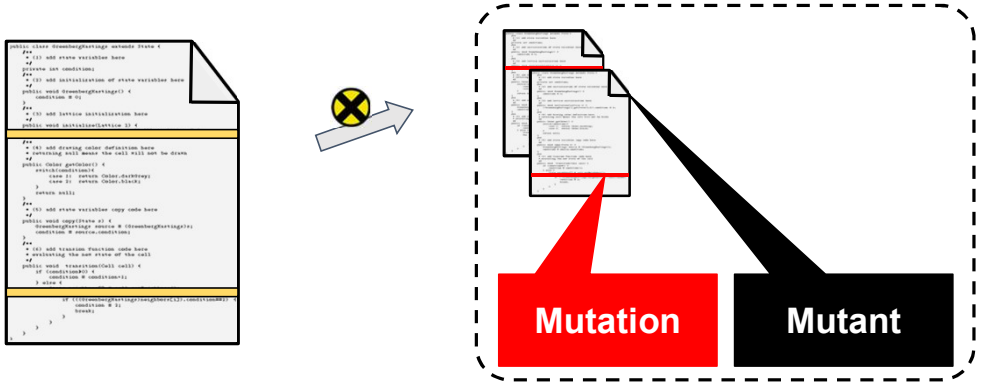
Program

Mutants



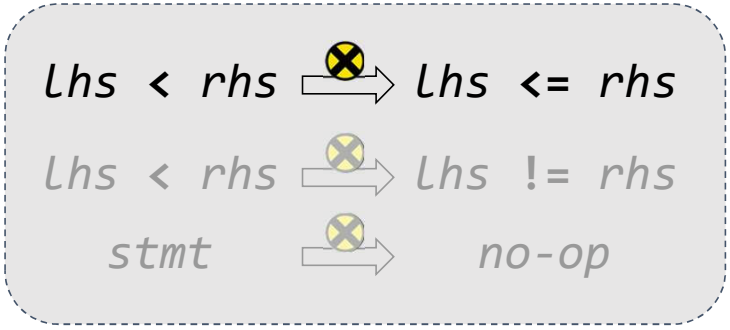
Mutation operators

Mutation testing: mutant generation



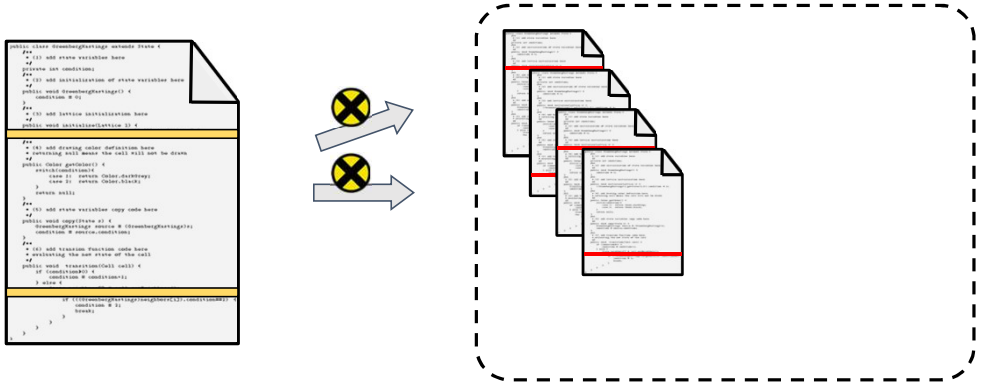
Program

Mutants



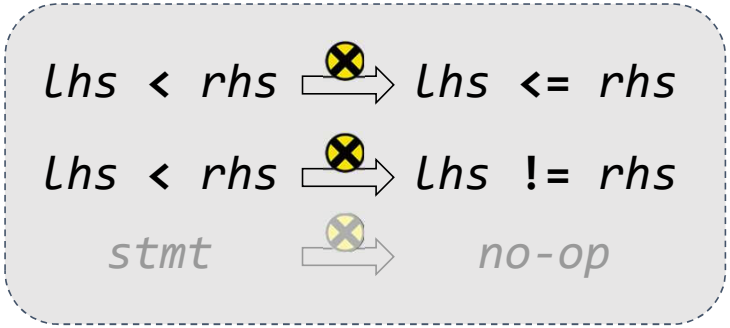
Mutation operators

Mutation testing: mutant generation



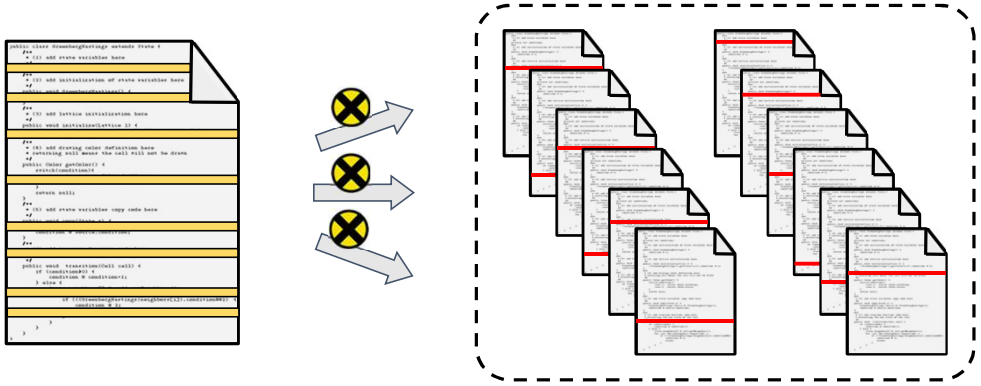
Program

Mutants



Mutation operators

Mutation testing: mutant generation



Program

Mutants

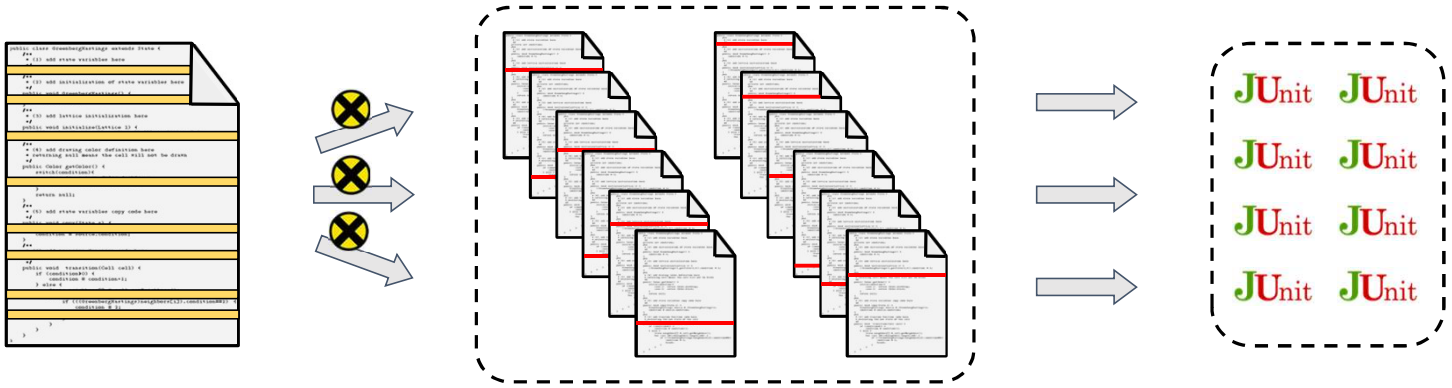
lhs **<** *rhs* ⇒ *lhs* **<=** *rhs*

lhs **<** *rhs* ⇒ *lhs* **!=** *rhs*

stmt ⇒ *no-op*

Mutation operators

Mutation testing: test creation



Program Mutants Tests

Assumptions

- Mutants are coupled to real faults
- Mutant detection is correlated with real-fault detection

https://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf,
https://homes.cs.washington.edu/~rjust/publ/mutation_testing_practices_icse_2021.pdf

Mutation: a concrete example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant 1:

```
public int min(int a, int b) {
    return a;
}
```

Mutation: another example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant 2:

```
public int min(int a, int b) {
    return b;
}
```

Mutation: yet another example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant 3:

```
public int min(int a, int b) {
    return a >= b ? a : b;
}
```

Mutation: last example (I promise)

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant 4:

```
public int min(int a, int b) {
    return a <= b ? a : b;
}
```

Mutation score

Input: a test suite and a set of mutants
Score: fraction of mutants failing (killed/detected) by the test suite

Example: test suite fails for 3 of the 4 mutants; score = .75

Jargon: to “kill” a mutant is for the test to fail
Why is a test failure good?

Mutation testing: let's practice

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

For each mutant, provide a test case that detects it
(i.e., passes on the original program but fails on the mutant)

Mutation testing: let's practice

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

a	*b*	Original	M1	M2	M3	M4
1	2	1				
1	1	1				
2	1	1				

Mutation testing: let's practice

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

a	*b*	Original	M1	M2	M3	M4
1	2	1	1			
1	1	1	1			
2	1	1	**2**			

Mutation testing: let's practice

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

a	*b*	Original	M1	M2	M3	M4
1	2	1	1	**2**		
1	1	1	1	1		
2	1	1	**2**	1		

Mutation testing: let's practice

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

a	*b*	Original	M1	M2	M3	M4
1	2	1	1	**2**	**2**	
1	1	1	1	1	1	
2	1	1	**2**	1	**2**	

Mutation testing: let's practice

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

M4 cannot be detected (equivalent mutant)

a	*b*	Original	M1	M2	M3	M4
1	2	1	1	**2**	**2**	1
1	1	1	1	1	1	1
2	1	1	**2**	1	**2**	1

Mutation testing: let's practice

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

Which mutant(s) should we show to a developer?

a	*b*	Original	M1	M2	M3	M4
1	2	1	1	**2**	**2**	1
1	1	1	1	1	1	1
2	1	1	**2**	1	**2**	1

Mutation testing: let's practice

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

Redundant

Equivalent

a	*b*	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: challenges

- **Redundant mutants** (produce same output as another mutant (s))
  - Inflate the mutant detection ratio
  - Hard to assess progress and remaining effort
- **Equivalent mutants** (produce same output as original program)
  - Max mutant detection ratio != 100%
  - Waste resources

Redundant **Equivalent**

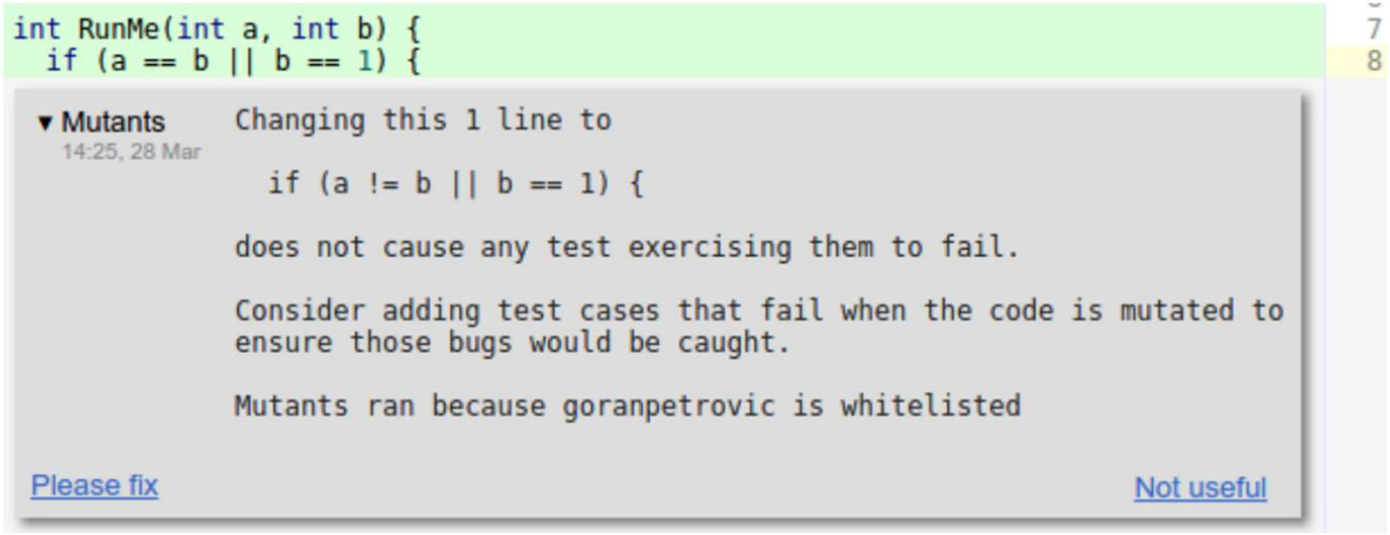
a	*b*	Original	M1	M2	M3	M4
1	2	1	1	**2**	**2**	1
1	1	1	1	1	1	1
2	1	1	**2**	1	**2**	1

Productive mutants

A mutant is **productive** if it is

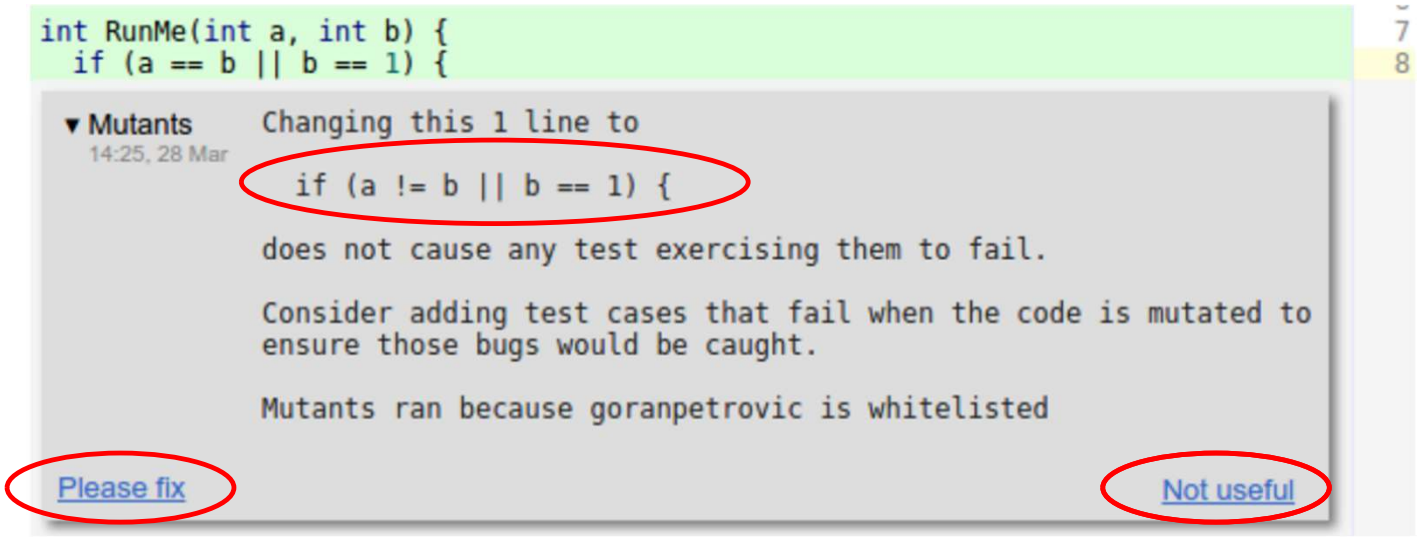
1. detectable and elicits an effective test or
2. equivalent and advances code quality or knowledge

Productive mutants: mutation testing at Google



Practical Mutation Testing at Scale: A view from Google (Reading)

Productive mutants: mutation testing at Google



Practical Mutation Testing at Scale: A view from Google (Reading)

Looking ahead -

Mutation in-class exercise on Friday

Bring laptop, work in partners

Read mutation testing basics beforehand (link on Calendar): https://courses.cs.washington.edu/courses/cse403/25wi/project/mutation-basics.html

Last topic for today -

Integration testing

Do you get the expected results when the parts are put together?

Start with plain, “integration”

Integration: combining 2 or more software units and getting the expected results

Why do we care about integration?

- New problems will inevitably surface
  - Many modules are now together that have never been together before
- If done poorly, all problems will present themselves at once
  - This can be hard to diagnose, debug, fix
- There can be a cascade of interdependencies
  - Cannot find and solve problems one-at-a-time

What do you think of phased integration

Phased ("big-bang") integration:

- Design, code, test, debug each class/unit/subsystem separately
- Combine them all
- Hope for the best

In contrast to incremental integration

Incremental integration:

- Repeat
  - Design, code, test, debug a new component
  - Integrate this component with another (a larger part of the system)
  - Test the combination
- Can start with a functional "skeleton" system (e.g., zero feature release)
  - And incrementally “flesh it out”

Is it obvious which is more successful?

- **Incremental integration** benefits:
  - Errors easier to isolate, find, fix
    - reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - good for customer, developer morale

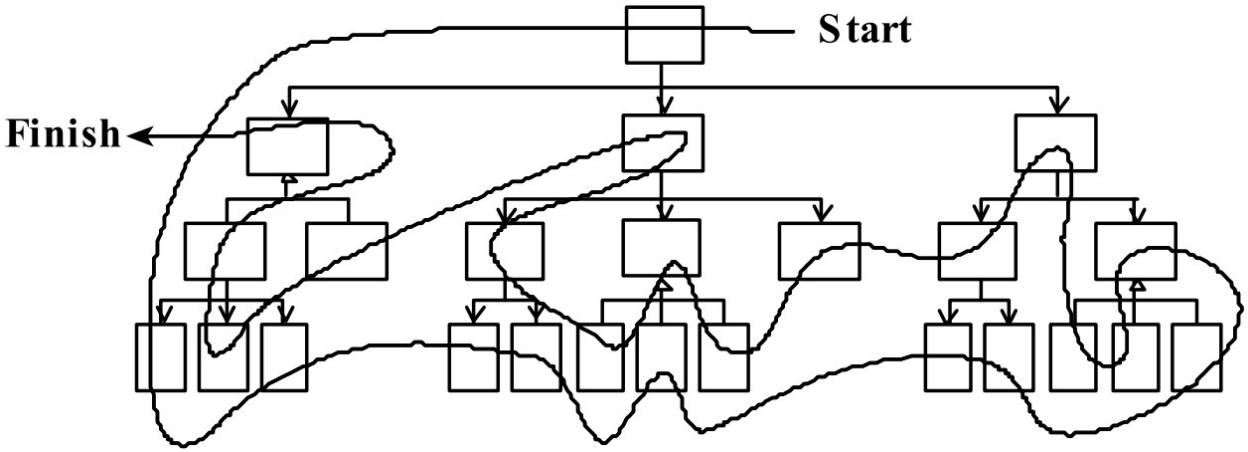
- But it isn't without challenges:
  - May need to create "**stub**" versions of some features that aren't yet available

Incrementally from the top, bottom or “sandwich"?

“Sandwich" integration by fleshing out a skeleton system

Connect top-level UI with crucial bottom-level components

- Add middle layers incrementally
- More common and agile approach



Milestone 05: Beta

Demo a skeleton implementation of your product showing the main components are integrated

Integration testing

Integration testing: verifying software quality by testing two or more dependent software modules as a group

Can be quite challenging as:

- Combined units can fail in more places and in more complicated ways
- May need to use stubs to "rig" behavior if not all pieces yet exist

That’s a wrap (for now) – testing takeaways

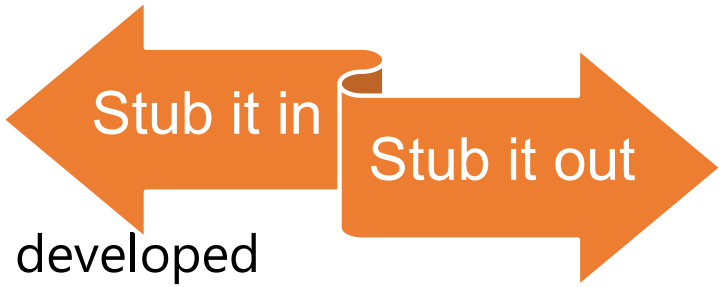
- Testing matters!!!
- Test early, test often
  - Bugs become well-hidden beyond the unit in which they occur
- Don't confuse volume with quality of test data
  - Can lose relevant cases in mass of irrelevant ones
  - Look for revealing subdomains (“characteristic tests”)
- Choose test data to cover:
  - Specification (black box testing)
  - Code (white box testing)
- Testing can't generally prove absence of bugs
  - But it can increase quality and confidence

Additional reference material

What's a stub?

Stub: a controllable replacement for a software unit

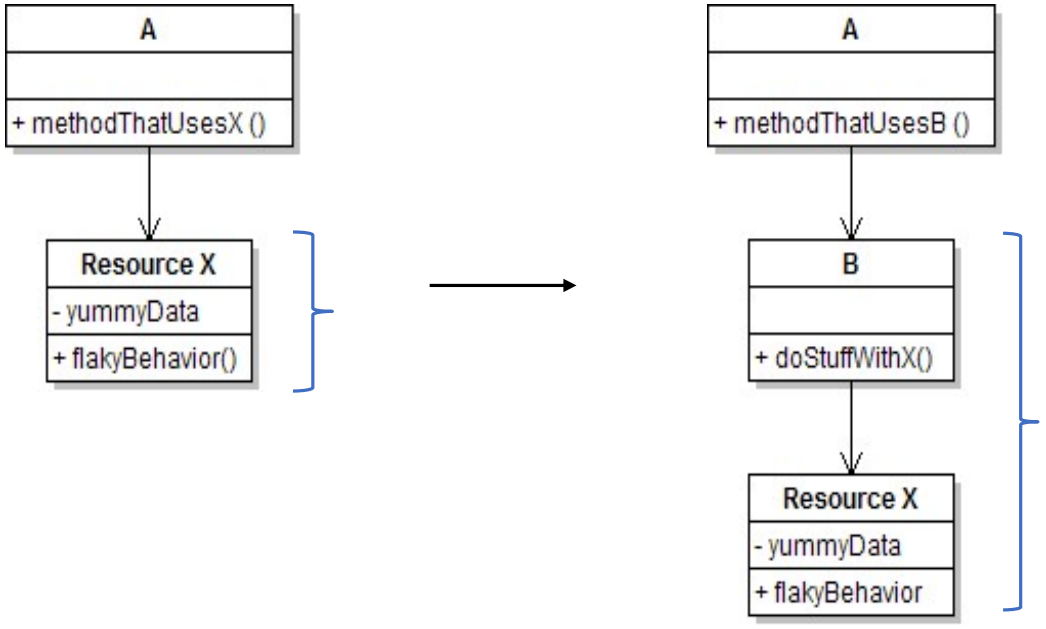
- Useful for simulating difficult-to-control elements, e.g.,
  - network / internet
  - database
  - files



- Useful for simulating components not yet developed

How to create a stub, step 1

1. Identify the dependency
   a) This is either a resource or a class/object that is challenging or not yet written
   b) If it isn't an object, wrap it up into one



Goal: Test class A

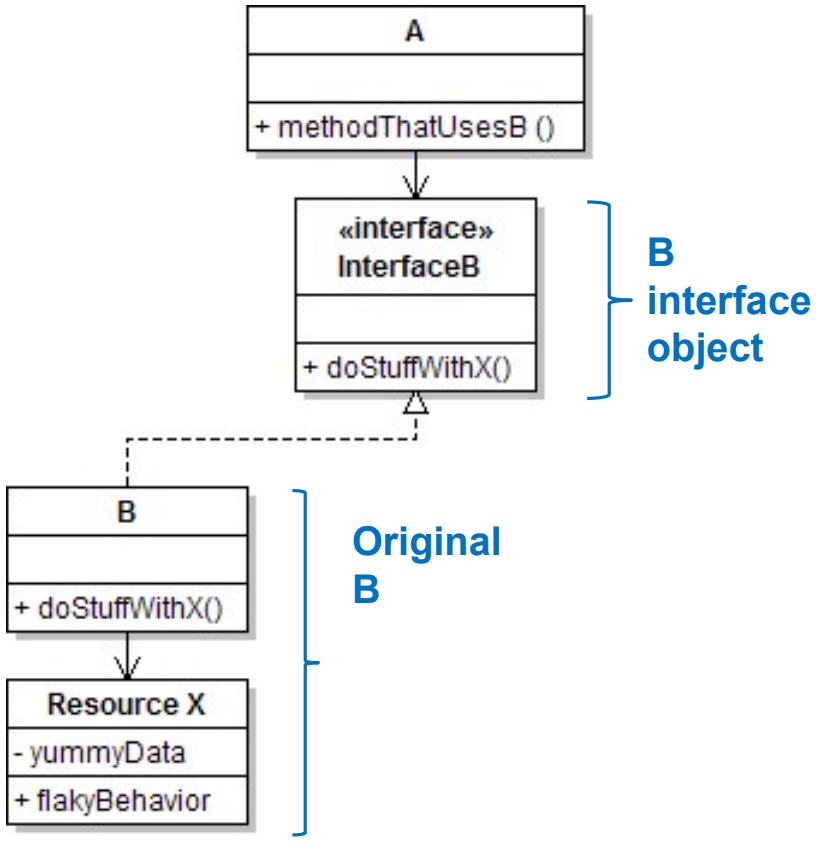
Create Class B to represent the challenging/missing dependency (as needed)

How to create a stub, step 2

2. Extract the core functionality of the object into an interface

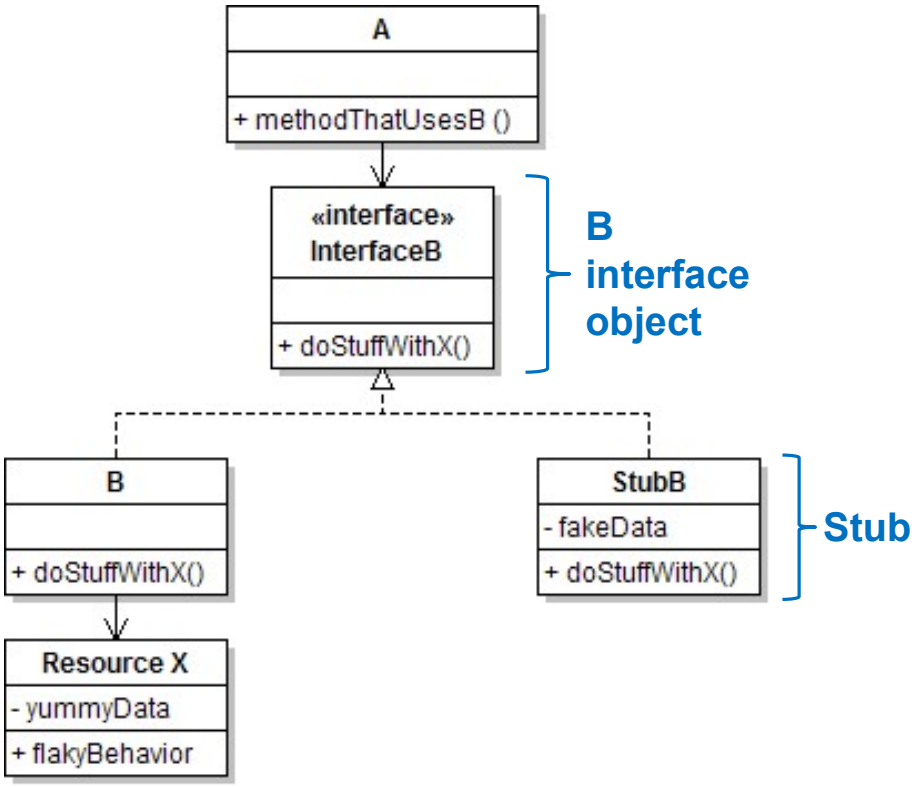
Create a **stub** InterfaceB based on B

Update A's code to work with type InterfaceB, not B



Create a stub, step 3

3. Write a second "stub" class that also implements the interface,
   but returns pre-determined fake data



Now A's dependency on B is dodged and can be tested easily

Can focus on how well A *integrates* with B's expected behavior

Inject the stub, step 4

So cool! Where inject the stub in the code so Class A will reference it?

- At construction
  apple = new A(**new StubB()**);
- Through a getter/setter method
  apple.setResource(**new StubB()**);
- Just before usage, as a parameter
  apple.methodThatUsesB(**new StubB()**);

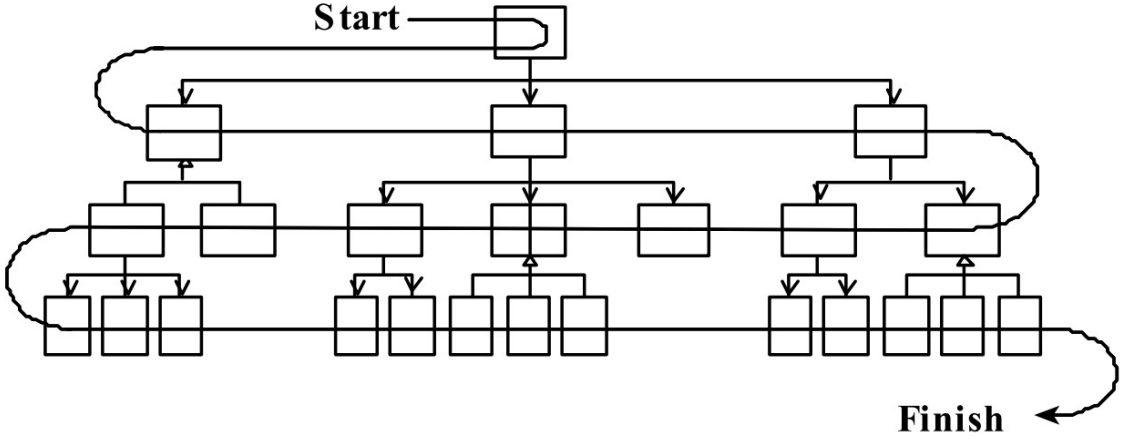
Think about how to minimize code changes when you no longer depend on the stub

There are different ways to approach integration

Top-down integration:

Start with outer UI layers and work inward

- Must write (lots of) lower level stubs for UI to interact with
- Allows postponing tough design/implementation decisions (
- bad?)



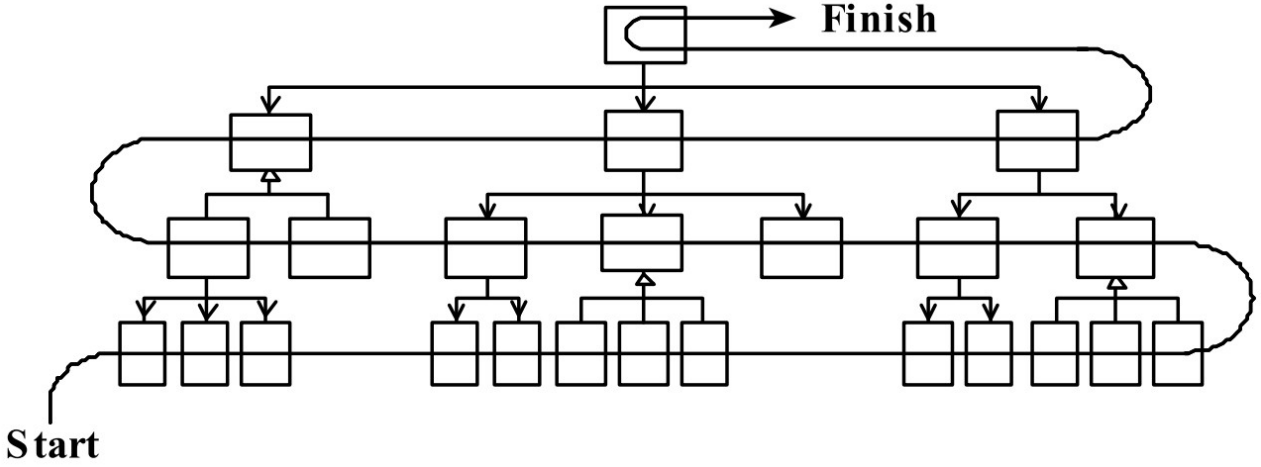
Steve McConnel, Code Complete 2

Or bottom-up

Bottom-up integration:

Start with low-level data/logic layers and work outward

- Must write upper level stubs to drive these layers
- Won't discover high-level / UI design flaws until late



Evaluating a test suite: maximize a metric

Input: a test suite and something else

Output: a measurement of the something else

Something else:

- Lines of code executed = code coverage
- Conditions evaluated to true and/or false = branch coverage
- …
- Mutation score