CSE 403 Software Engineering

More Testing

Autumn 2023

Today’s outline

Software testing

- Code coverage
- Integration and integration testing

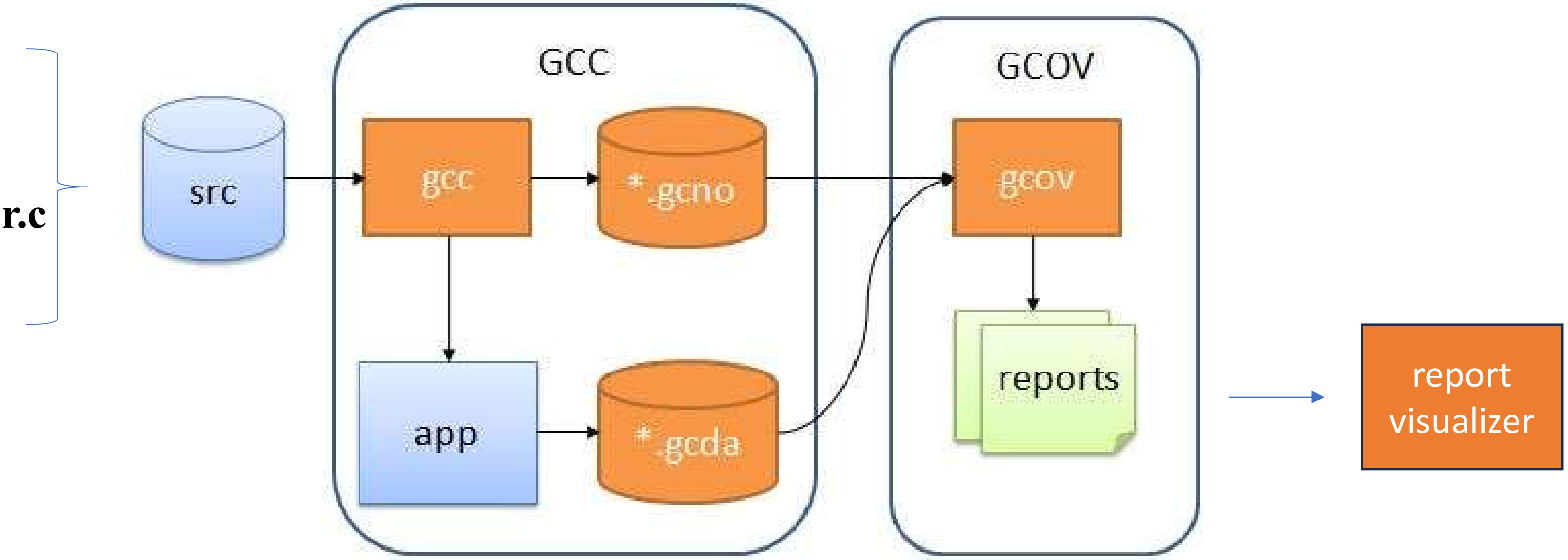
Jumping into a demo – calculator module

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

calculator.c
test_calculator.c



How gcov works (Medium.com)

Intro to gcov demo

Link to CI in github

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

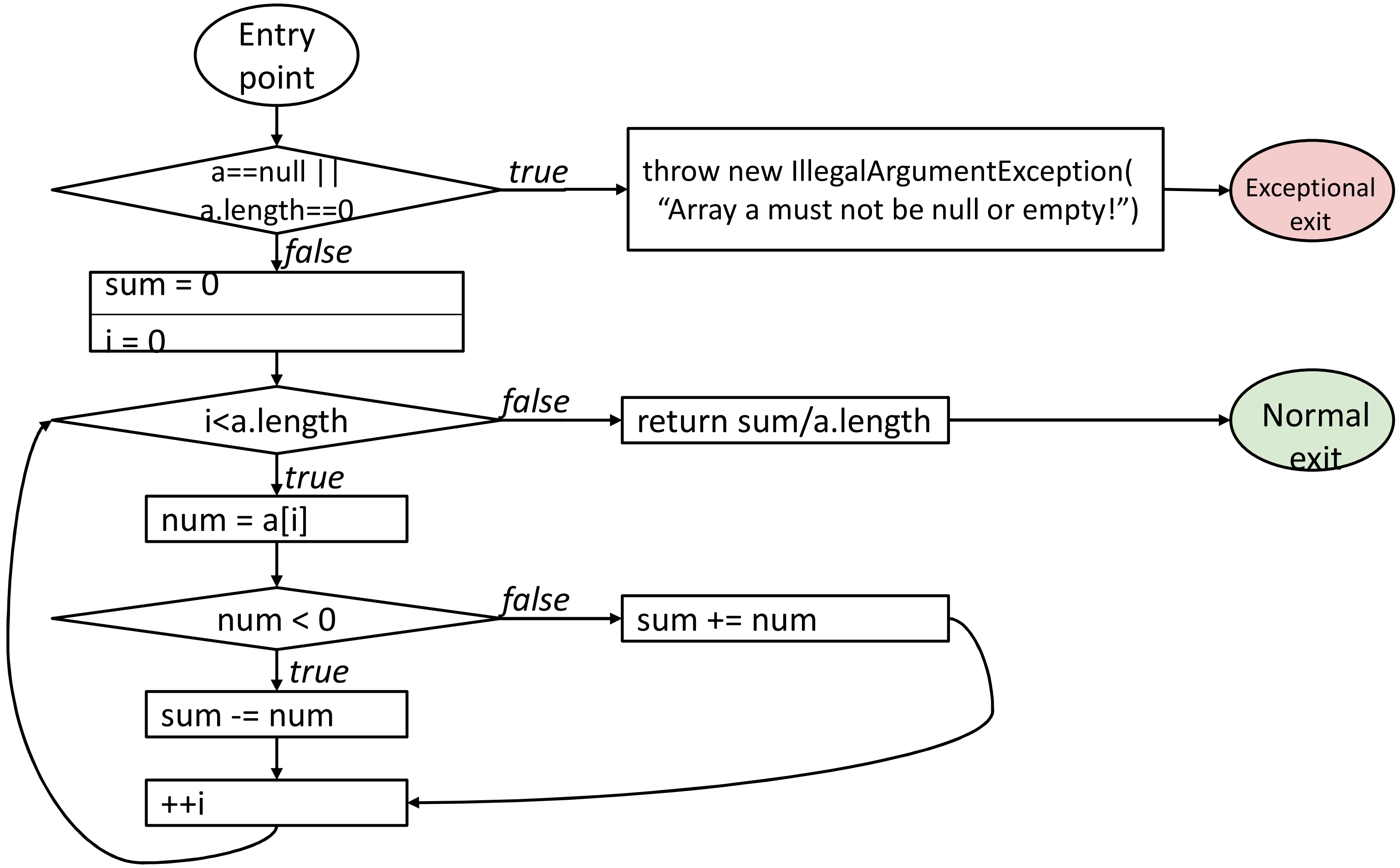
Which type of coverage requires the most tests?

Structural code coverage: the basics

Average of the absolute values of an array of doubles

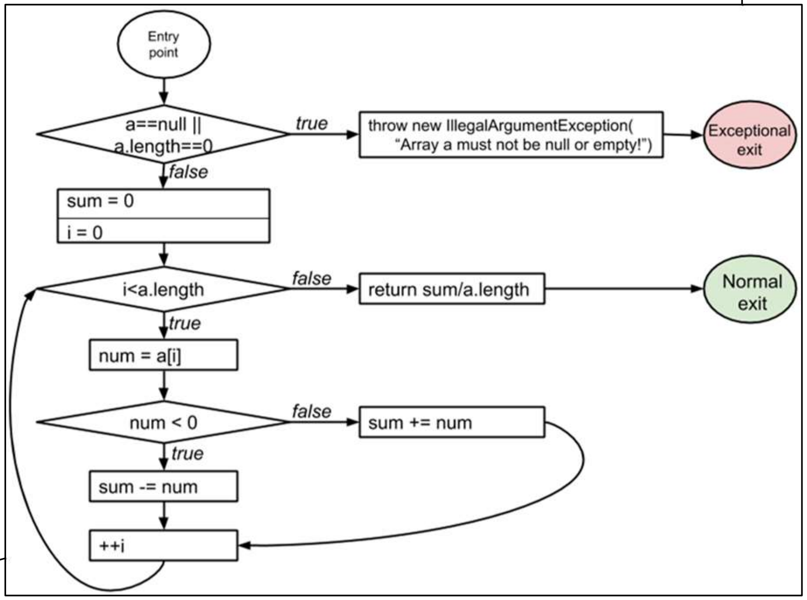
```
public double avgAbs(double ... numbers) {

  // 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

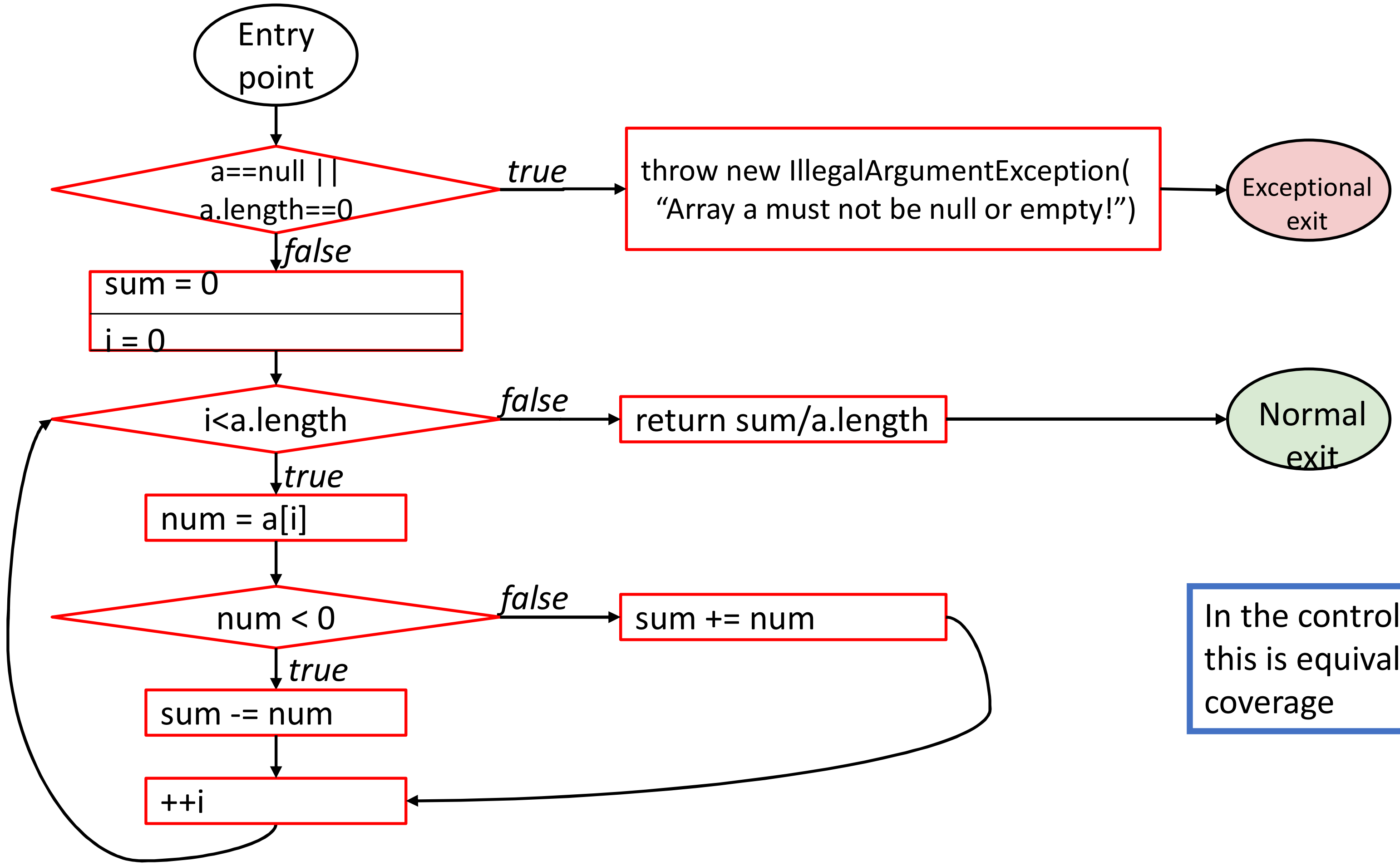
```
public double avgAbs(double ... numbers) {

  // 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) { …}
```

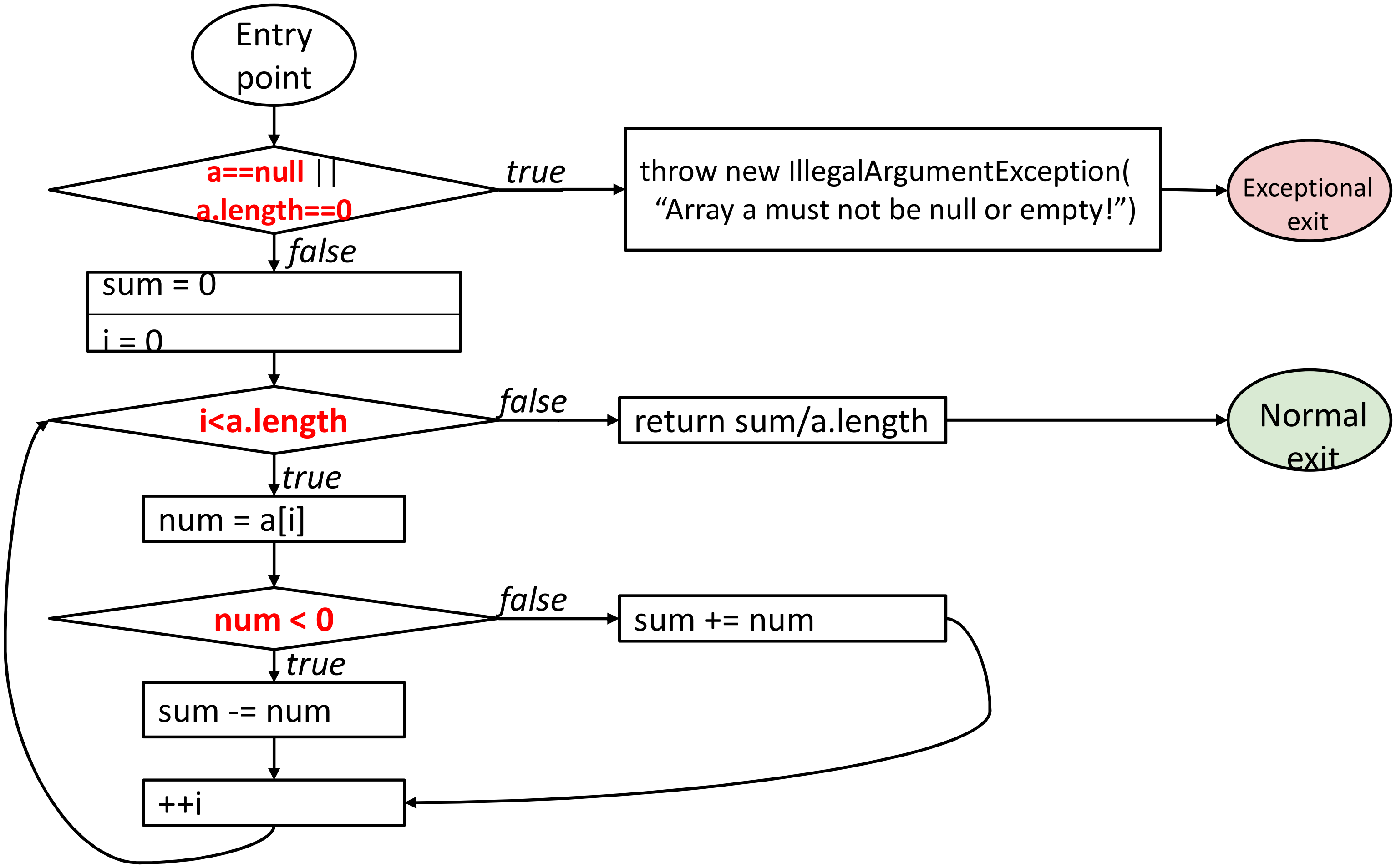
What are a and b?
What is the boolean expression (a | b)?

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

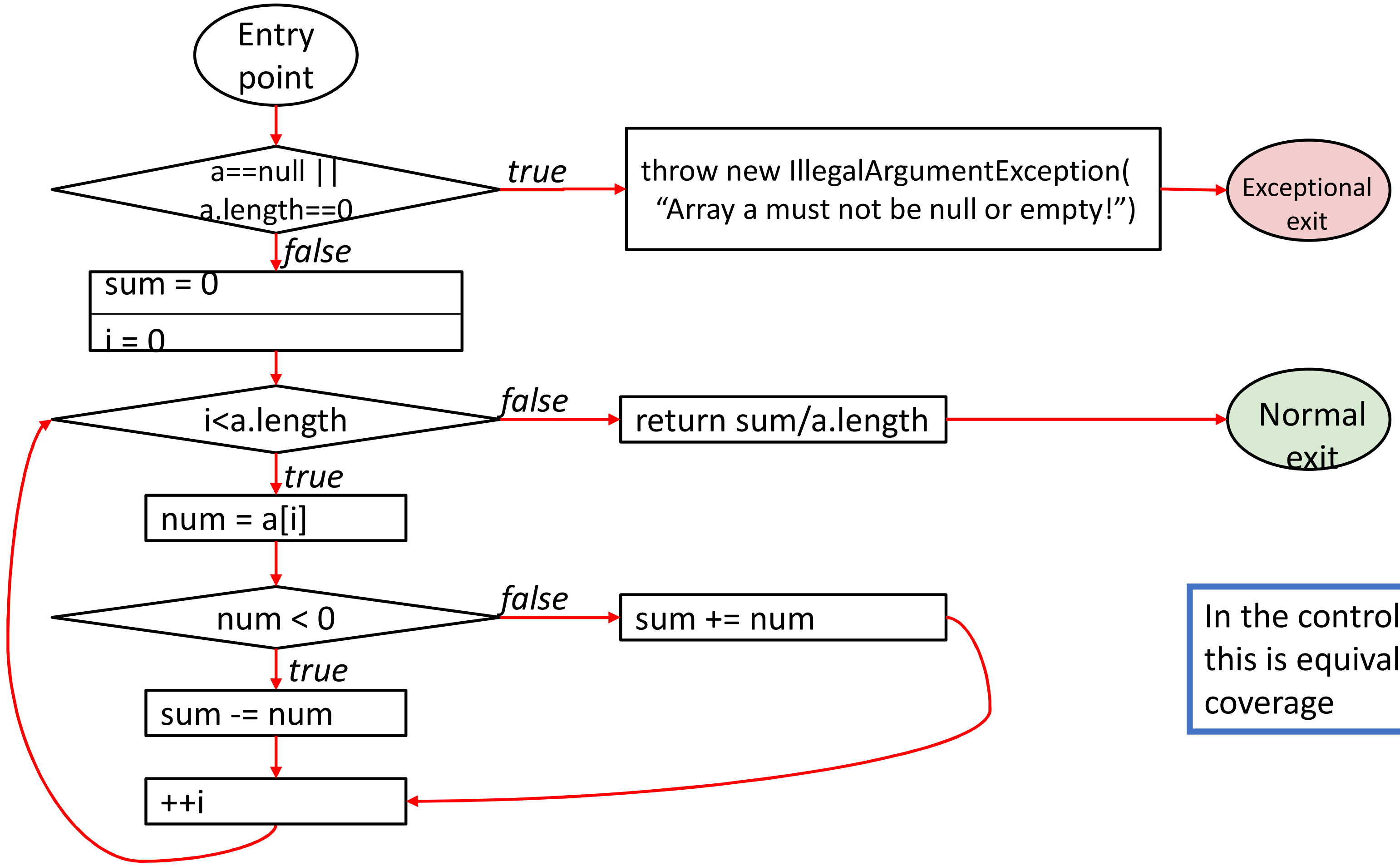


Decision coverage

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



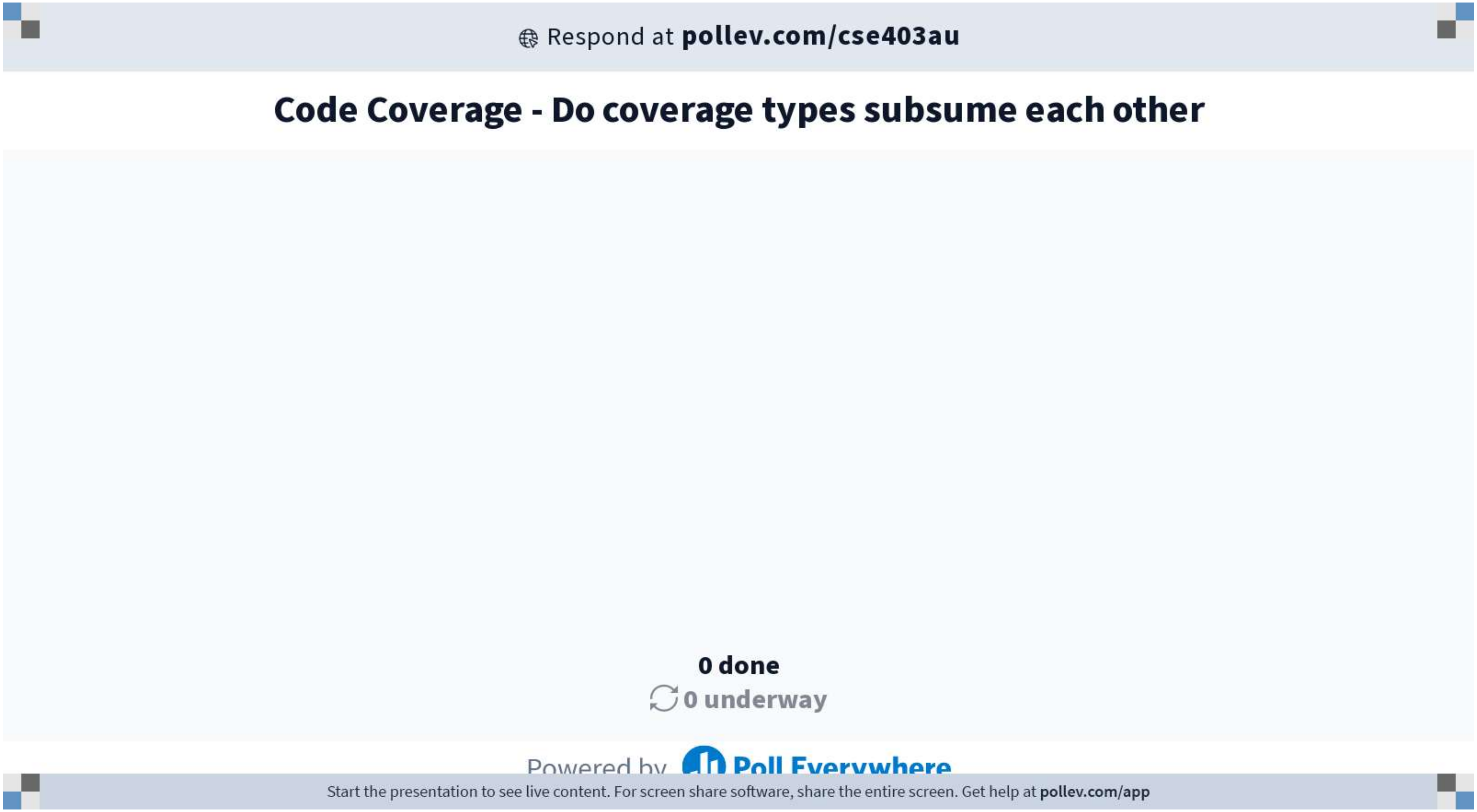
In the control flow graph, this is equivalent to **edge** coverage

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/cse403au



Respond at **pollev.com/cse403au**

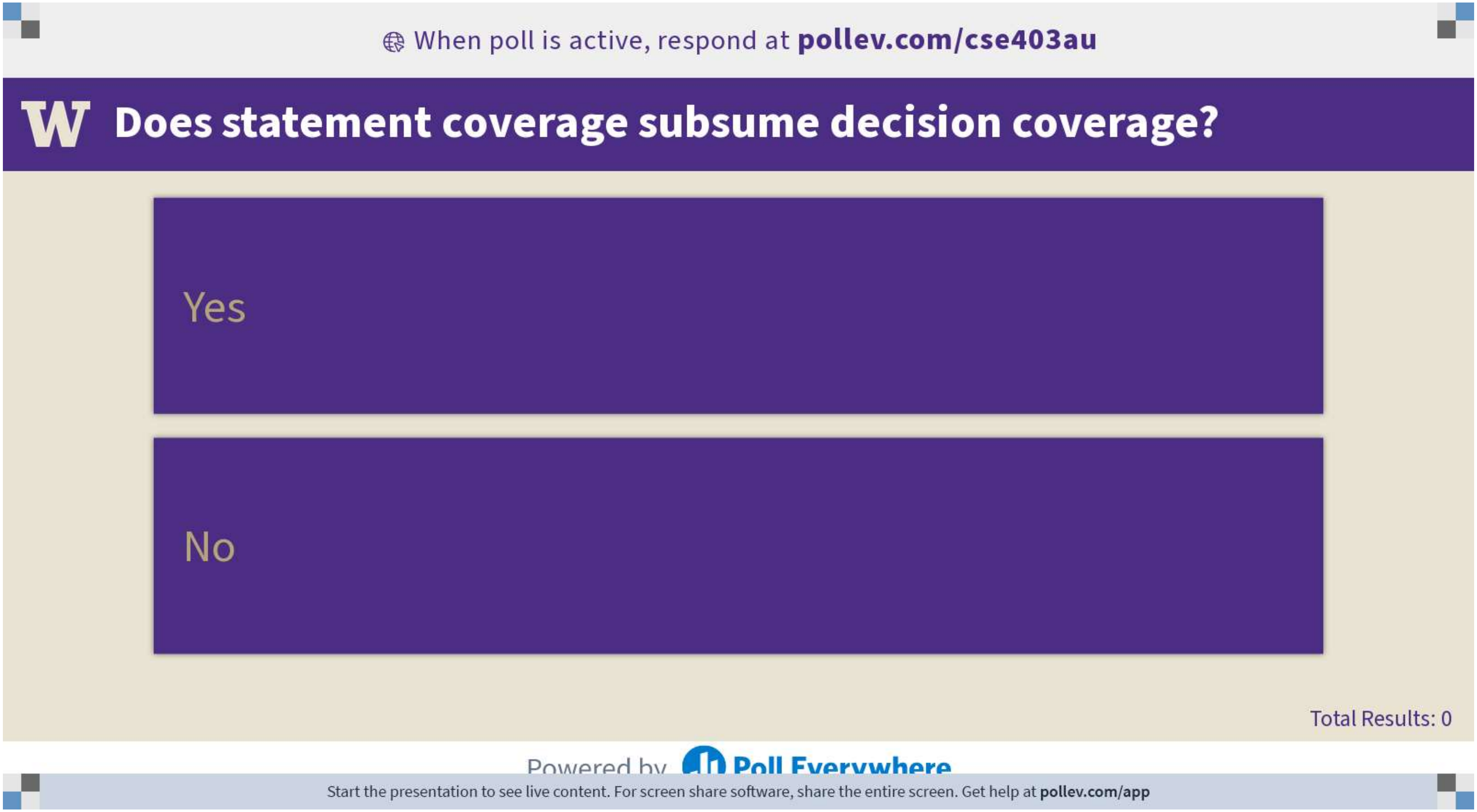
Code Coverage - Do coverage types subsume each other

0 done

0 underway

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**



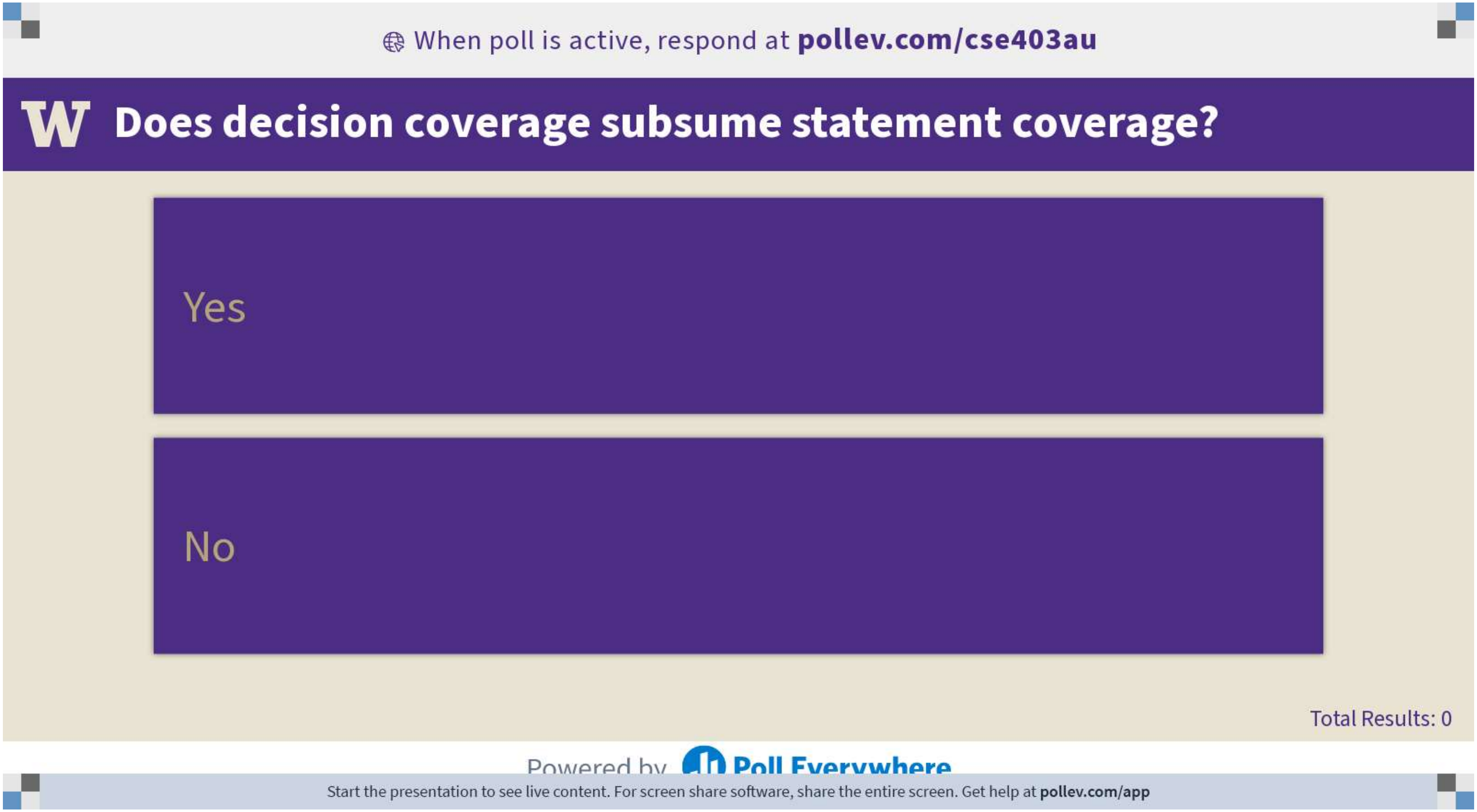
When poll is active, respond at **pollev.com/cse403au**

Does statement coverage subsume decision coverage?

Yes

No

Total Results: 0



When poll is active, respond at **pollev.com/cse403au**

Does decision coverage subsume statement coverage?

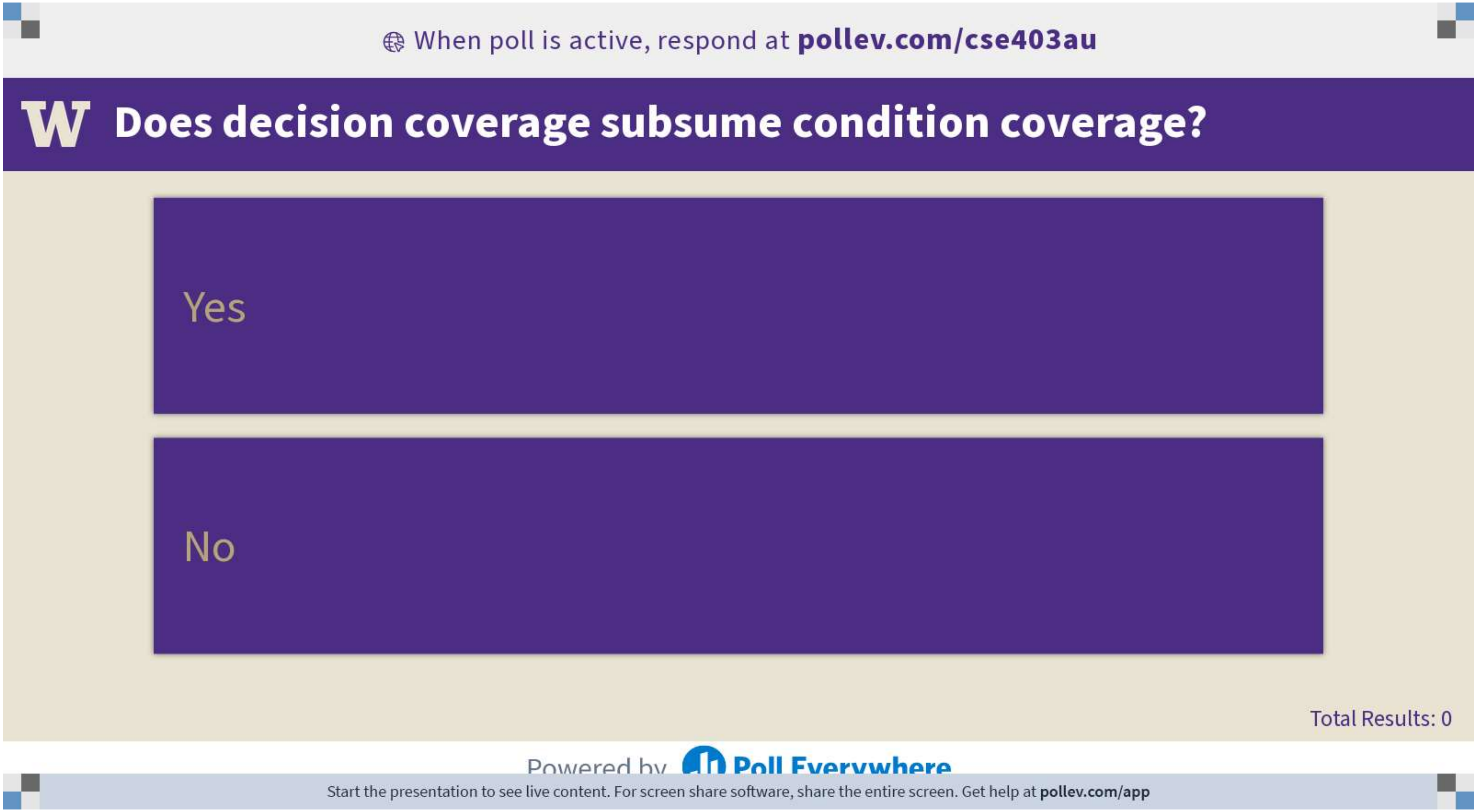
Yes

No

Total Results: 0

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**



When poll is active, respond at **pollev.com/cse403au**

Does decision coverage subsume condition coverage?

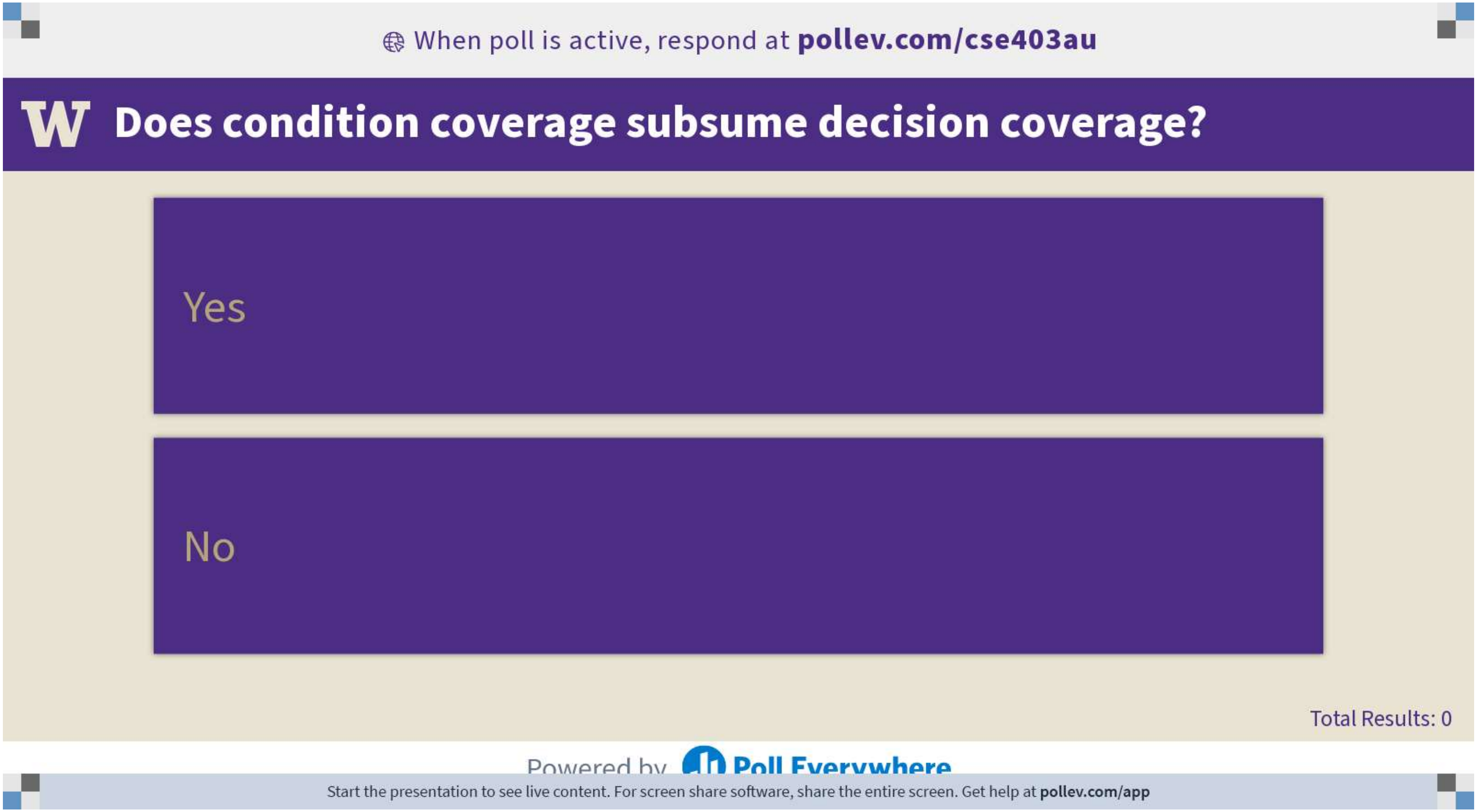
Yes

No

Total Results: 0

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**



When poll is active, respond at **pollev.com/cse403au**

Does condition coverage subsume decision coverage?

Yes

No

Total Results: 0

Powered by Poll Everywhere

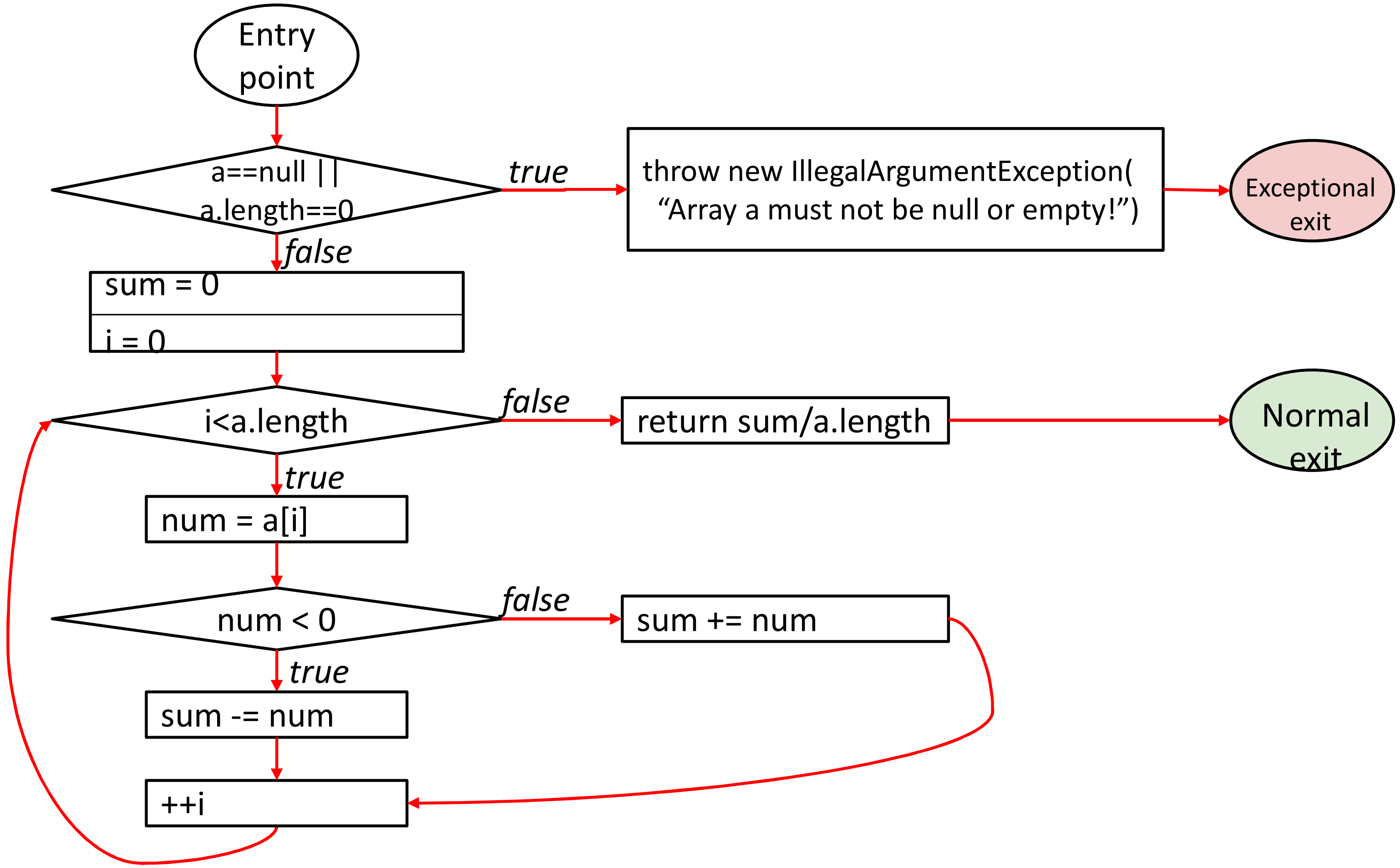
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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

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**

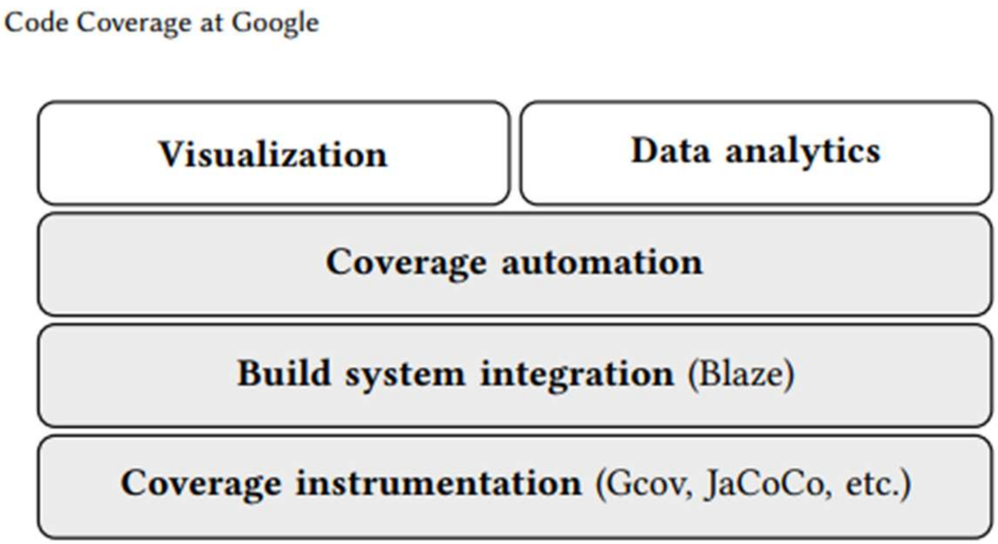
a	b	a | b
0	**0**	**0**
0	**1**	**1**
1	0	1
1	1	1

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

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

Code coverage at Google



Code Coverage at Google

Visualization
Data analytics
Coverage automation
Build system integration (Blaze)
Coverage instrumentation (Gcov, JaCoCo, etc.)

Layered architecture!

Visualization tools are built on top of code instrumentation tools

[More details: https://homes.cs.washington.edu/~rjust/publ/google_coverage_fse_2019.pdf](https://homes.cs.washington.edu/~rjust/publ/google_coverage_fse_2019.pdf)

Back to our four categories of testing

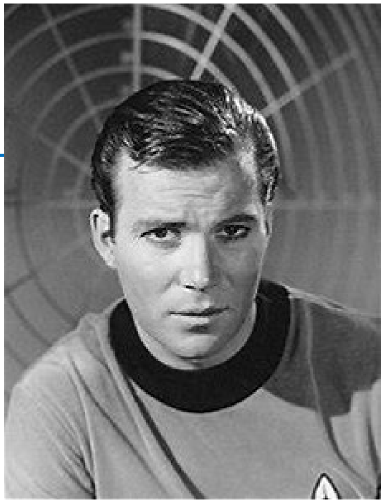
1. Unit Testing
   - Does each module do what it is supposed to do in isolation?
2. **Integration Testing**
   - **Do you get the expected results when the parts are put together?**
3. Validation Testing
   - Does the program satisfy the requirements?
4. System Testing
   - Does the program work as a whole and within the overall environment? (includes full integration, performance, scale, etc.)

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



“To go where no man has gone before…”

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

This Photo by Unknown Author is licensed under CC BY-NC-ND

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?

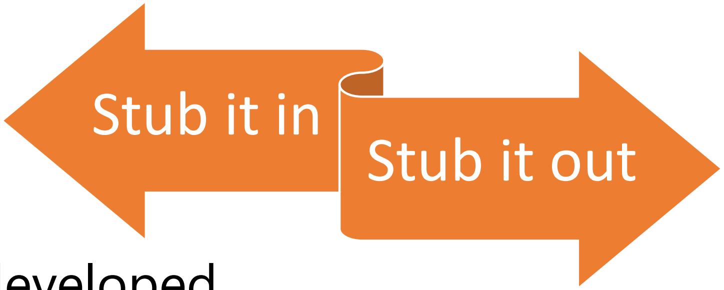
- **Incremental integration** benefits:
  - Errors easier to isolate, find, fix
    - reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - good for customer, developer morale

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

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



Stub it in

Stub it out

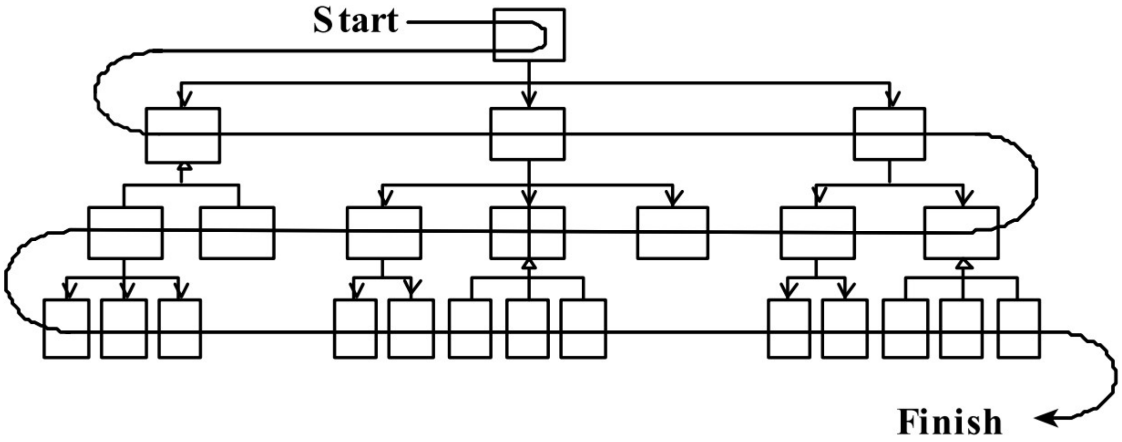
- Useful for simulating components not yet developed

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?)



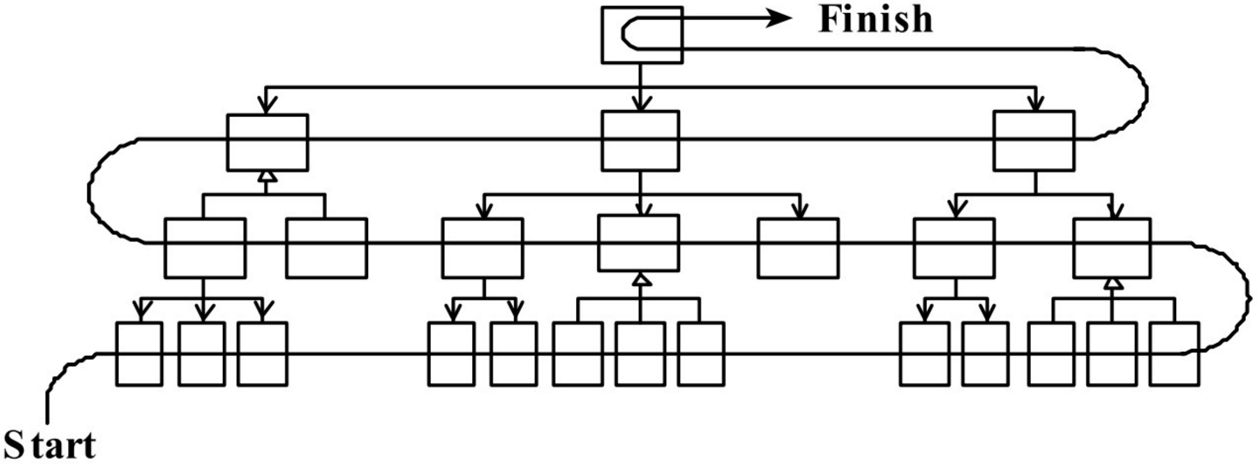
Steve McConnel, Code Complete 2

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



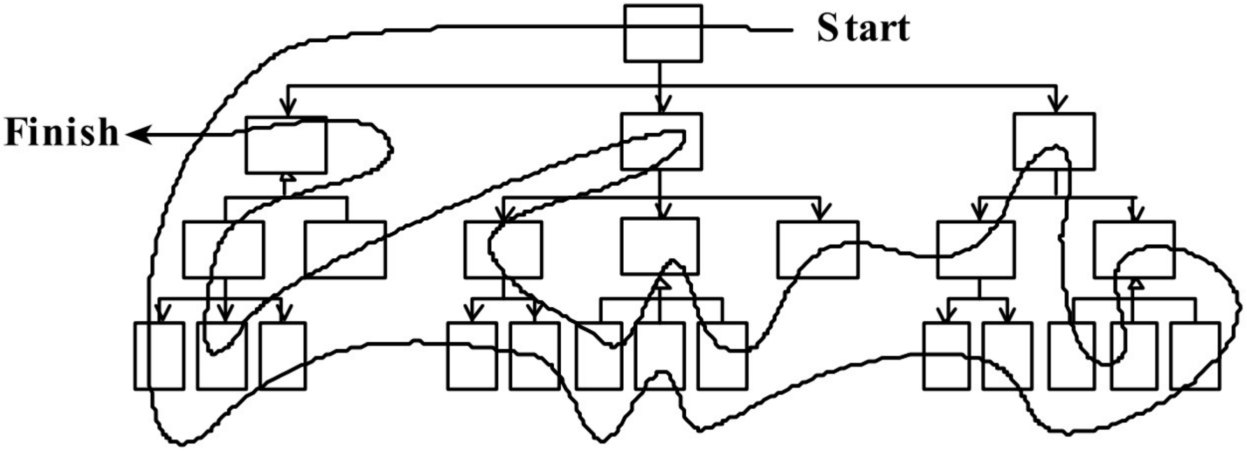
Top down, bottom up or “sandwich" integration?

“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

Consider starting with a skeleton implementation for your project



Onto integration testing

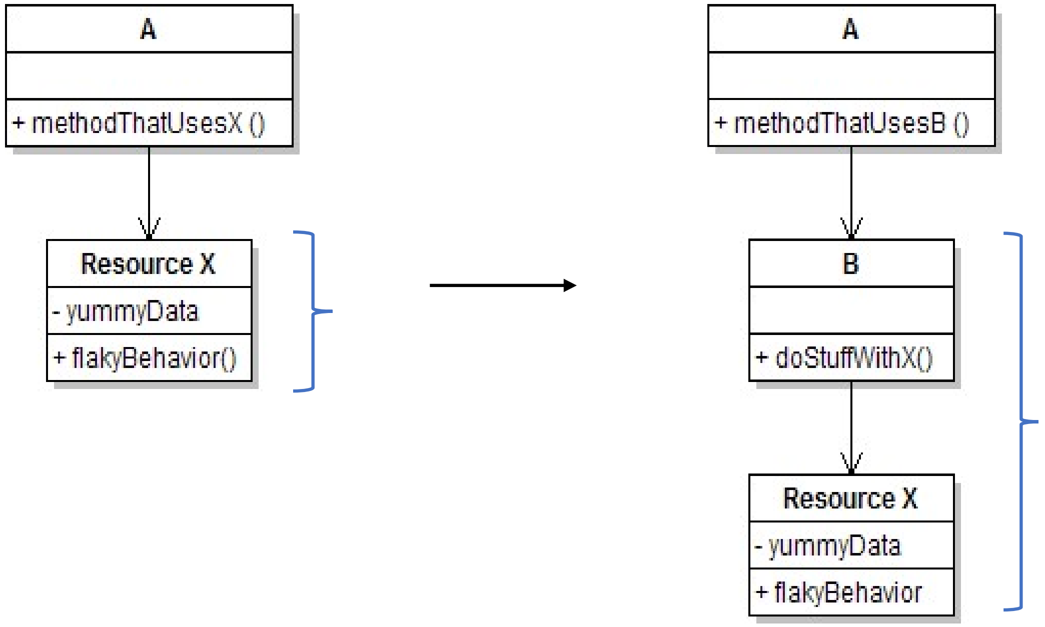
Integration testing: verifying software quality by testing two or more dependent software modules as a group

Can be quite challenging as:

- Combined units can fail in more places and in more complicated ways
- Must use **stubs** to "rig" behavior if not all pieces yet exist OR
    - if you want to simplify problematic components to debug more gradually

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)

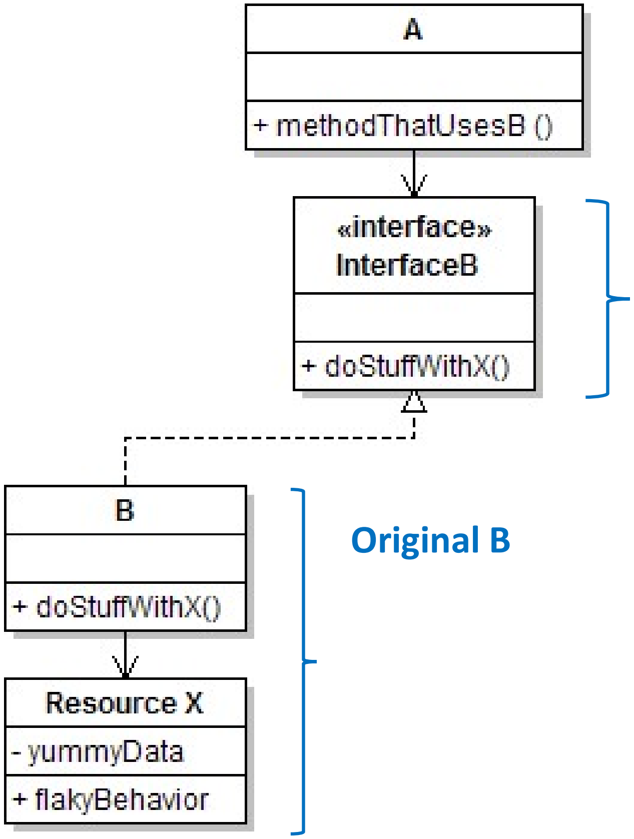
Class A depends on Class B

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



B interface object

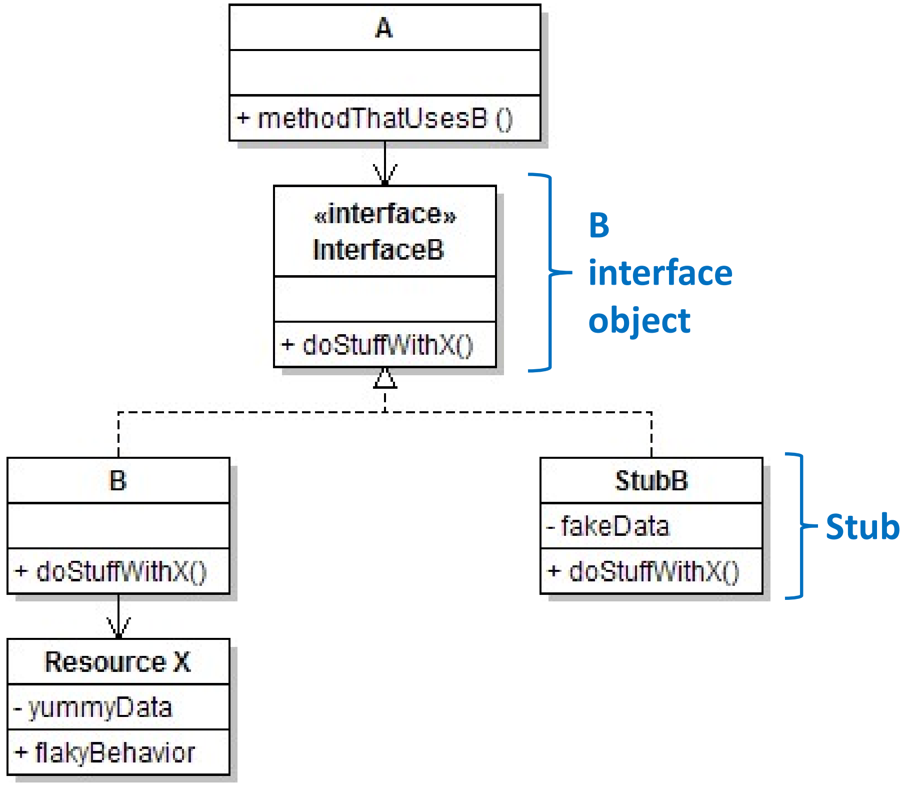
Original B

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

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

Appendix – Mock objects for integration testing

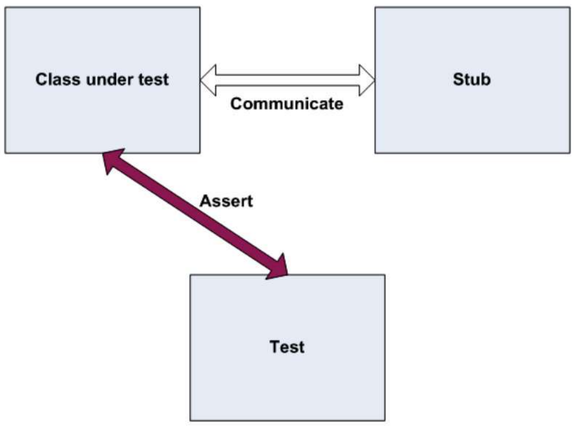
Mock objects
Mock vs stub objects

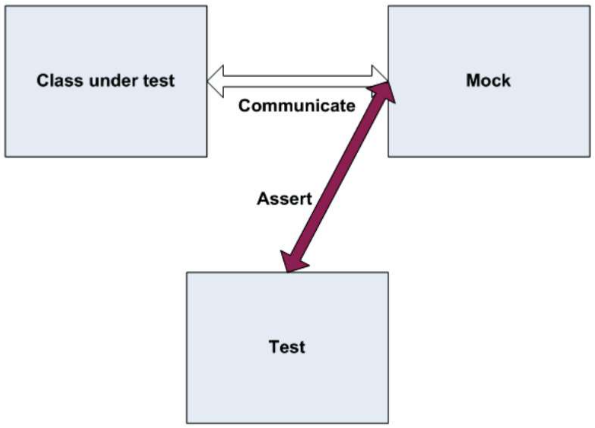
Thanks to Marty Stepp, previous UW CSE 403 instructor, for providing this and an earlier version of the integration testing material

"Mock" objects

mock object: a fake object that decides whether a unit test has passed or failed by watching interactions between objects

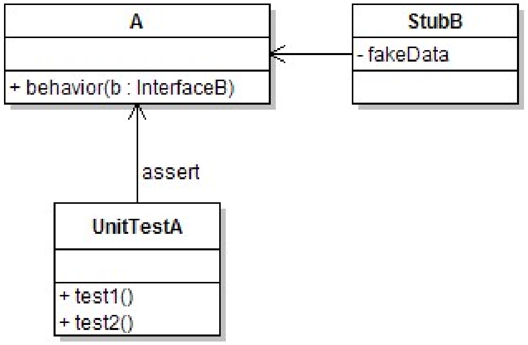
- useful for **interaction testing** (as opposed to **state testing**)



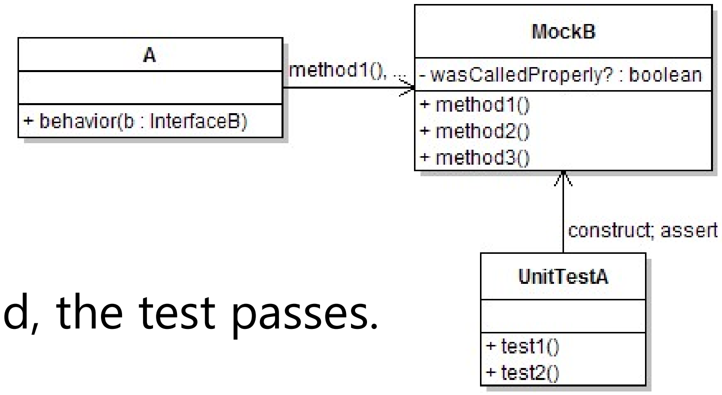


Stubs vs. mocks

- A **stub** gives out data that goes to the object/class under test.
- The unit test directly asserts against class under test, to make sure it gives the right result when fed this data.



- A **mock** waits to be called by the class under test (A).
  - Maybe it has several methods it expects that A should call.
- It makes sure that it was contacted in exactly the right way.
  - If A interacts with B the way it should, the test passes.



Mock object frameworks

- Stubs are often best created by hand/IDE.
  Mocks are tedious to create manually.

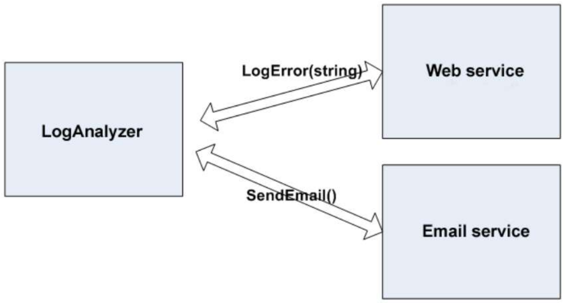
- Mock object frameworks help with the process.
  - android-mock, EasyMock, jMock (Java)
  - FlexMock / Mocha (Ruby)
  - SimpleTest / PHPUnit (PHP)
  - ...

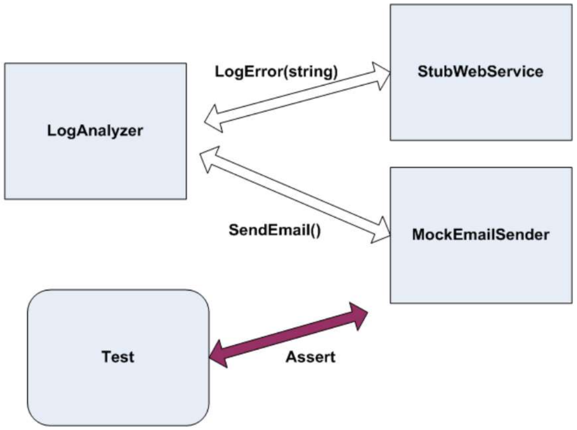
- Frameworks provide the following:
  - auto-generation of mock objects that implement a given interface
  - logging of what calls are performed on the mock objects
  - methods/primitives for declaring and asserting your expectations

Using stubs/mocks together

- Suppose a log analyzer reads from a web service.
  If the web fails to log an error, the analyzer must send email.
  - How to test to ensure that this behavior is occurring?

- Set up a *stub* for the web service that intentionally fails.
- Set up a *mock* for the email service that checks to see whether the ar send an e


LogAnalyzer
LogError(string)
Web service
SendEmail()
Email service


LogAnalyzer
LogError(string)
StubWebService
SendEmail()
MockEmailSender
Test
Assert