

# 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!**

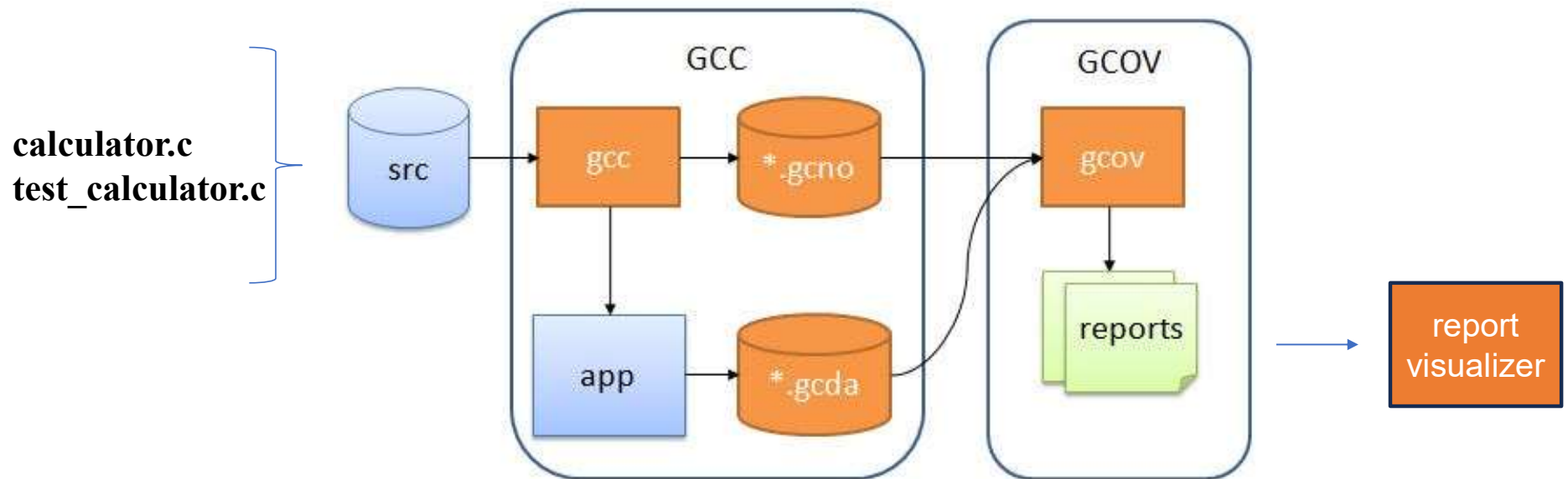
# Jumping into a demo – calculator module

## 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



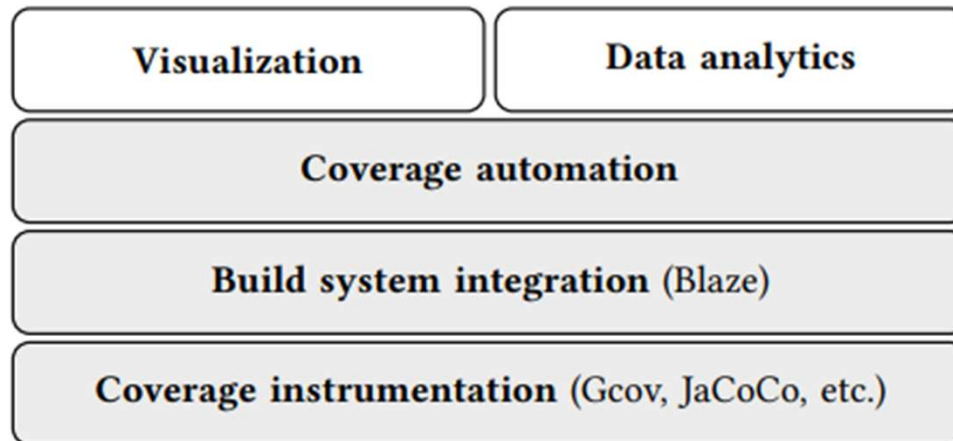
## Intro to gcov demo

Link it to your CI automation

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

# Code coverage at Google

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](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?

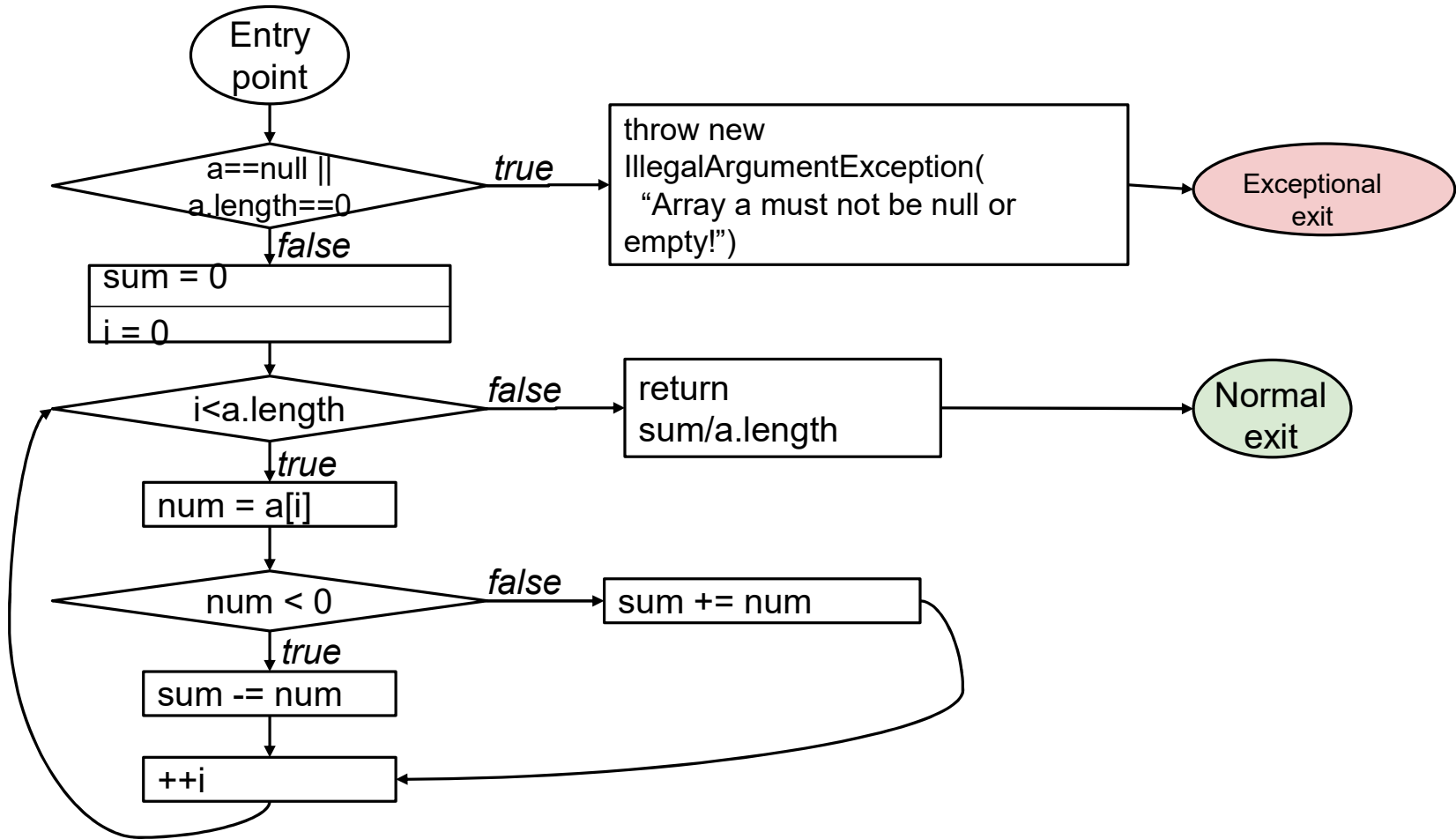
# Code coverage: the basics

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;  
}
```

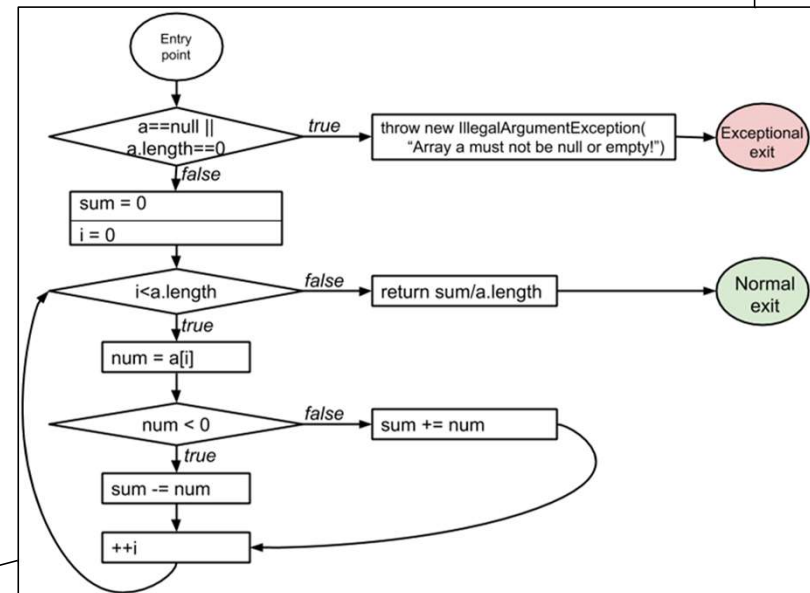


# Create the control flow graph



# And align the two to help identify tests

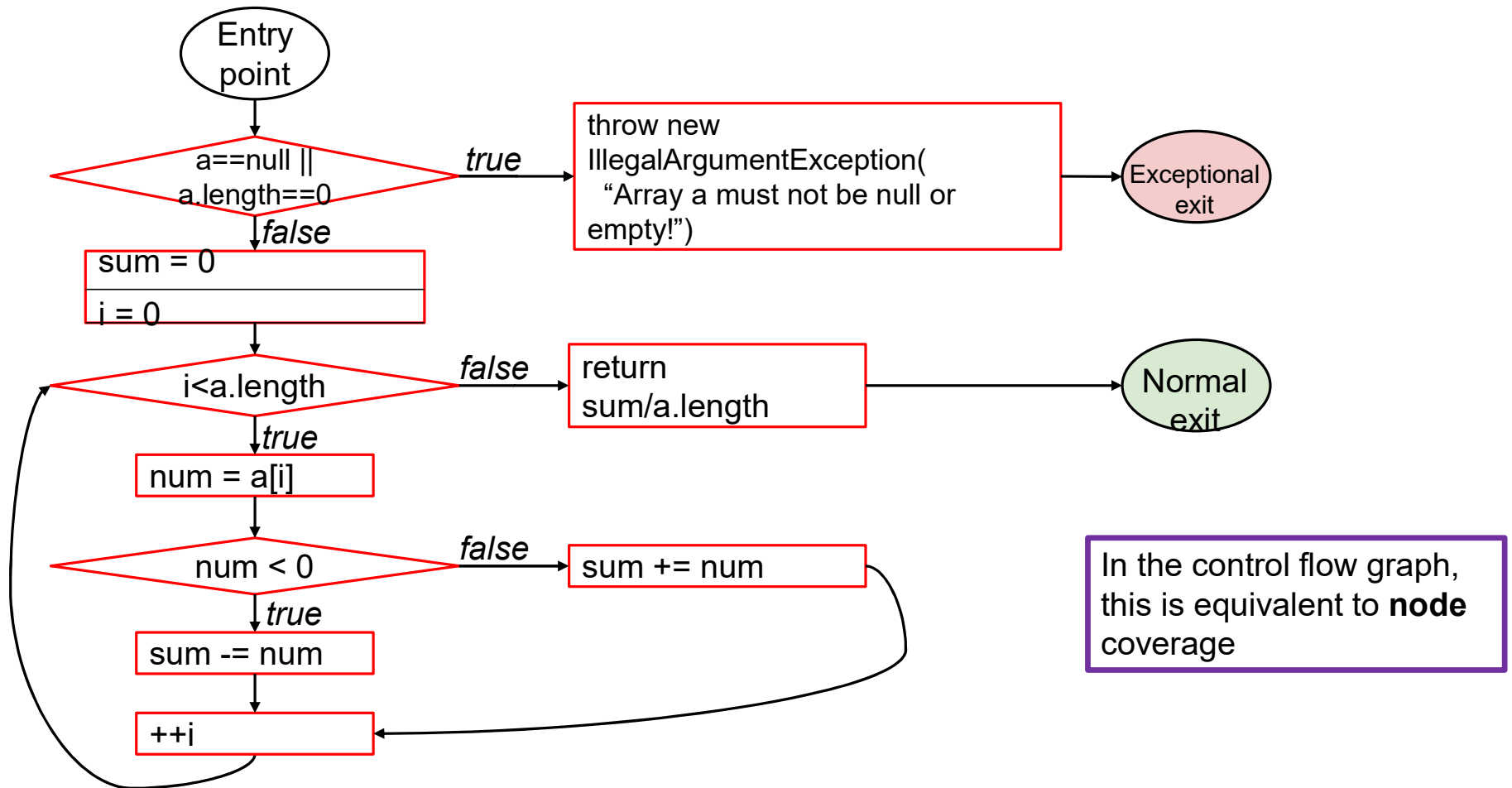
```
public double avgAbs(double ... numbers) {  
  
    // We expect the array to be non-null and non-empty  
    if (numbers == null || numbers.length == 0) {  
        throw new IllegalArgumentException("Numbers must not be null or empty!");  
    }  
  
    double sum = 0;  
    for (int i=0; i<numbers.length; ++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



# 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?

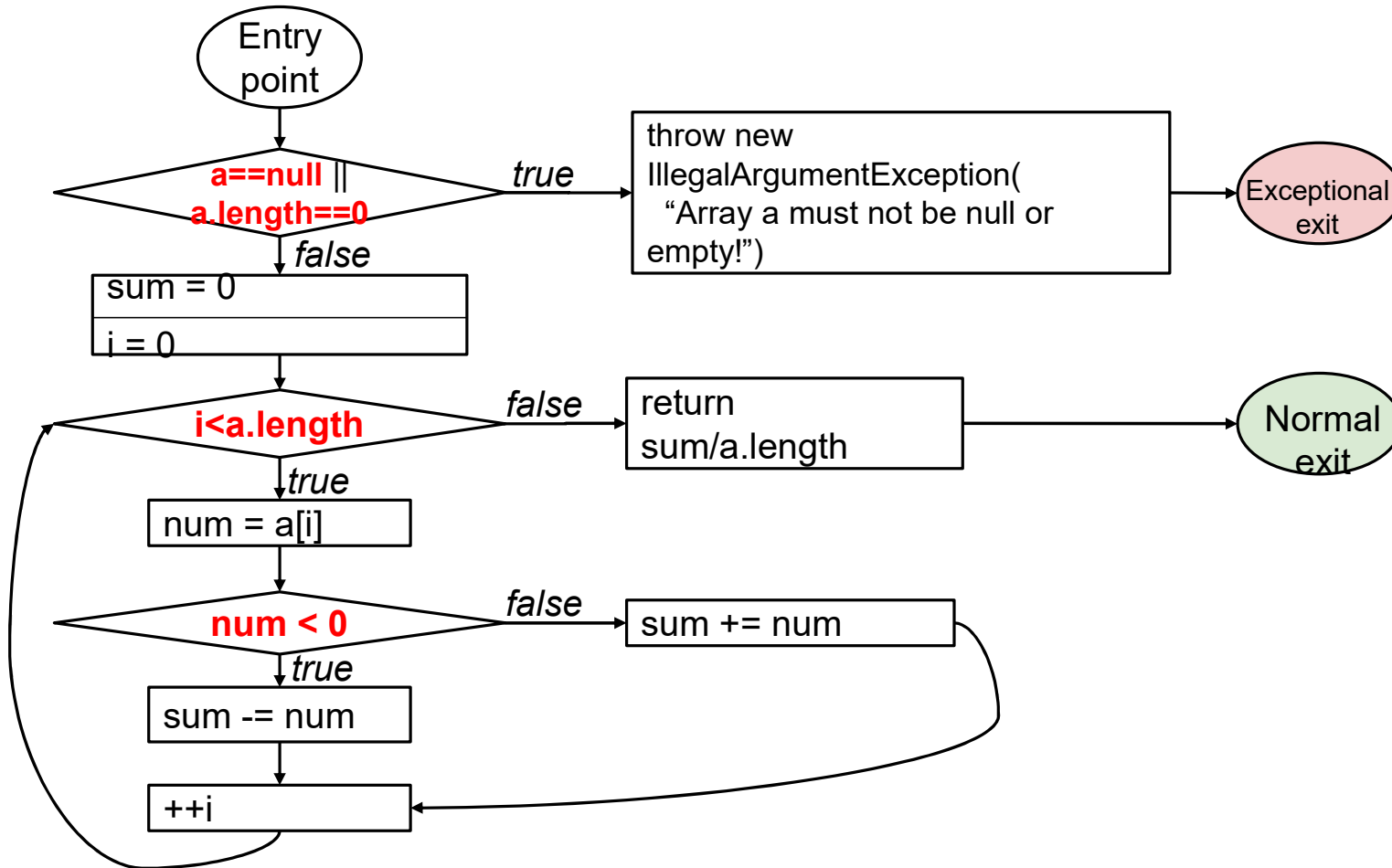
What is the boolean expression ( a | b )?

# 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



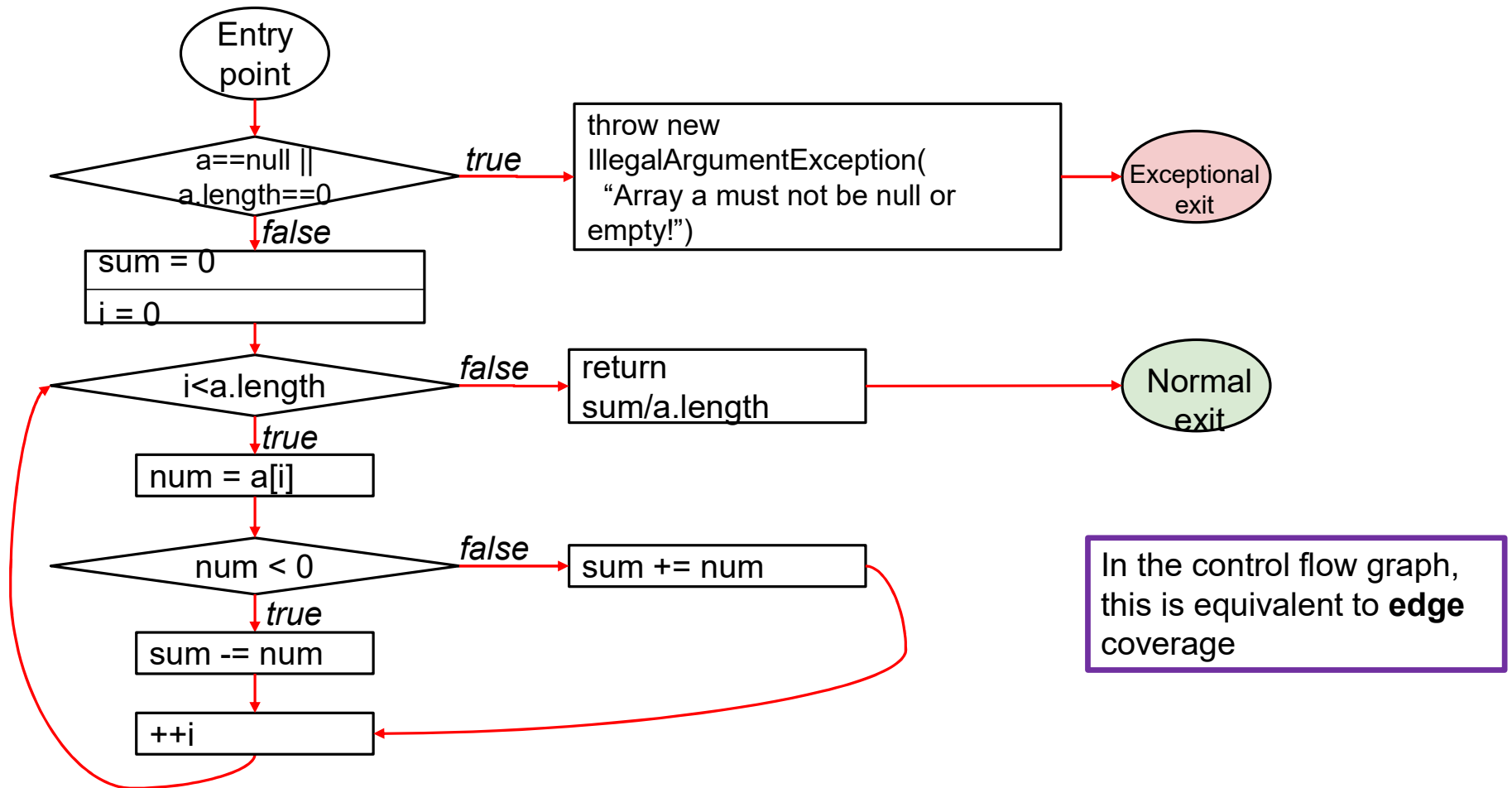
# Decision coverage

**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



# 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

## Does statement coverage subsume decision coverage?

Yes

No

## Does decision coverage subsume statement coverage?

Yes

No

## Does decision coverage subsume condition coverage?

Yes

No

## Does condition coverage subsume decision coverage?

Yes

No

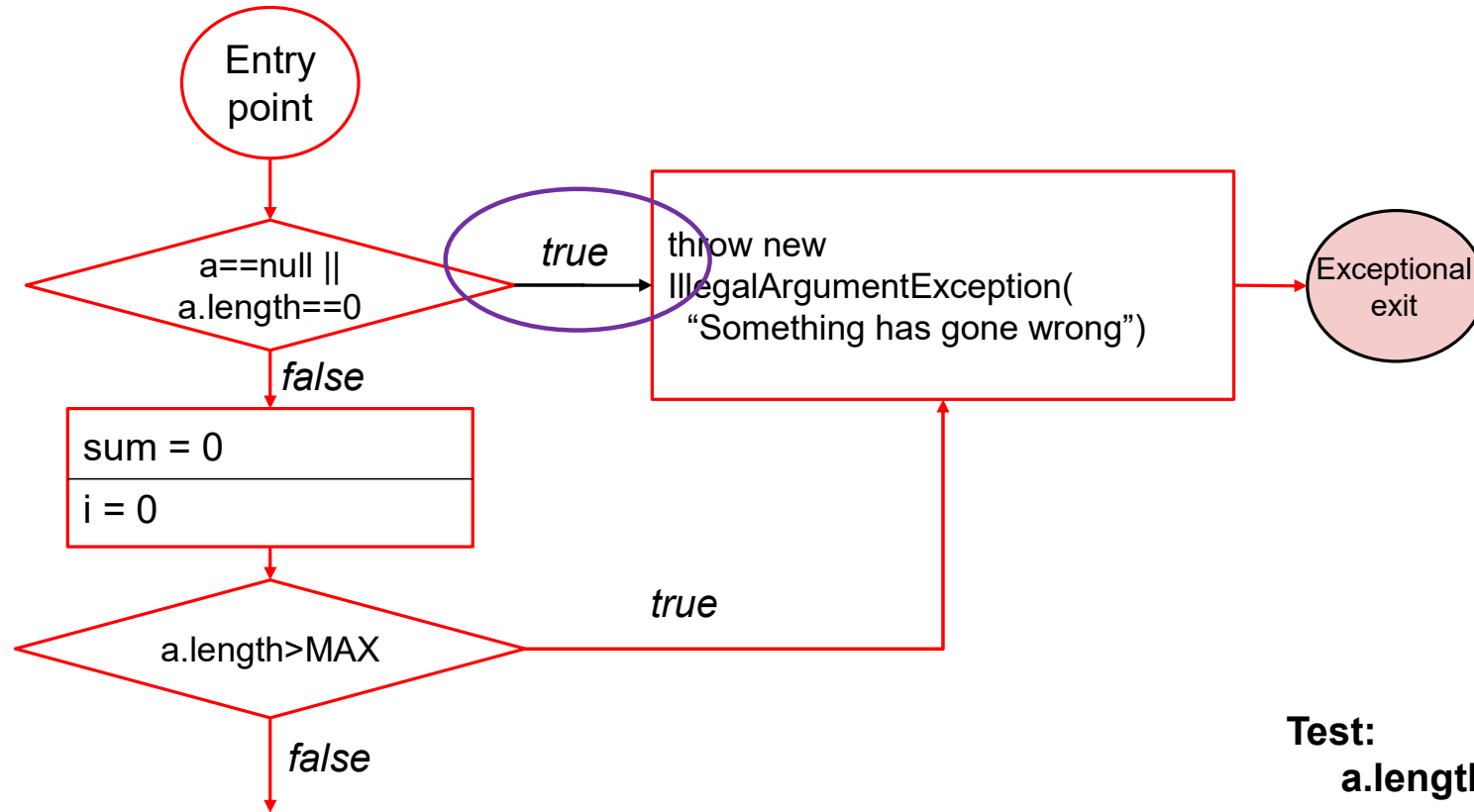
# And the experts say...

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

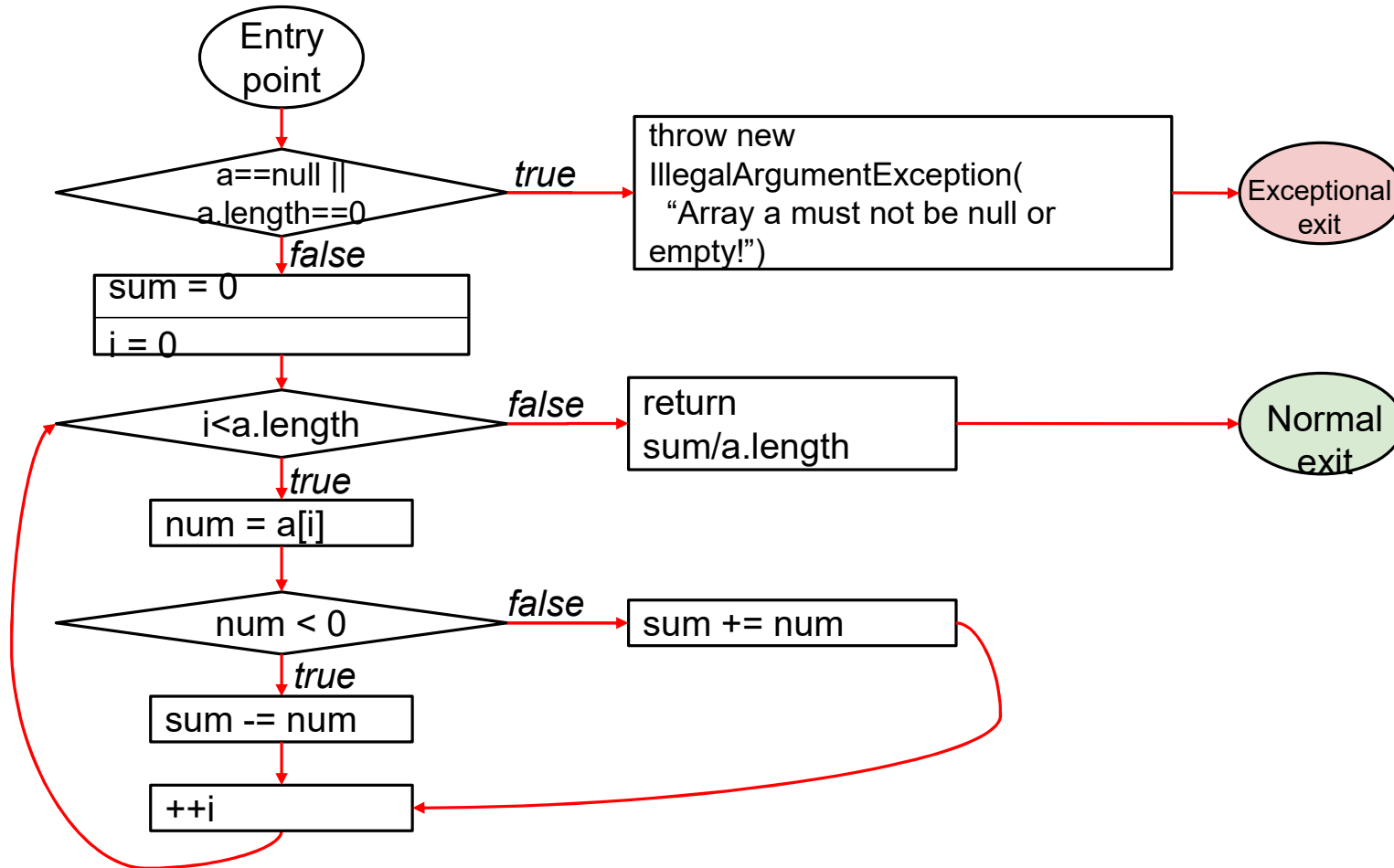


# Statement does not subsume Decision coverage



**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%



## 2. What percentage of coverage should you aim for?

There's no silver bullet in code coverage, and a high percentage of coverage could still be problematic if critical parts of the application are not being tested, or if the existing tests are not robust enough to properly capture failures upfront. With that being said it is generally accepted that 80% coverage is a good goal to aim for. Trying to reach a higher coverage might turn out to be costly, while not necessary producing enough benefit.

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

```
/**
 * @author ThomasHartmann
 */
public class Program {
    private int condition;
    public void doSomething() {
        condition = 0;
    }
    public void doSomethingElse() {
        condition = 1;
    }
    public void doSomethingThird() {
        condition = 2;
    }
    public void doSomethingFourth() {
        condition = 3;
    }
    public void doSomethingFifth() {
        condition = 4;
    }
    public void doSomethingSixth() {
        condition = 5;
    }
    public void doSomethingSeventh() {
        condition = 6;
    }
    public void doSomethingEighth() {
        condition = 7;
    }
    public void doSomethingNinth() {
        condition = 8;
    }
    public void doSomethingTenth() {
        condition = 9;
    }
    public void doSomethingEleventh() {
        condition = 10;
    }
    public void doSomethingTwelfth() {
        condition = 11;
    }
    public void doSomethingThirteenth() {
        condition = 12;
    }
    public void doSomethingFourteenth() {
        condition = 13;
    }
    public void doSomethingFifteenth() {
        condition = 14;
    }
    public void doSomethingSixteenth() {
        condition = 15;
    }
    public void doSomethingSeventeenth() {
        condition = 16;
    }
    public void doSomethingEighteenth() {
        condition = 17;
    }
    public void doSomethingNineteenth() {
        condition = 18;
    }
    public void doSomethingTwentieth() {
        condition = 19;
    }
    public void doSomethingTwentyFirst() {
        condition = 20;
    }
    public void doSomethingTwentySecond() {
        condition = 21;
    }
    public void doSomethingTwentyThird() {
        condition = 22;
    }
    public void doSomethingTwentyFourth() {
        condition = 23;
    }
    public void doSomethingTwentyFifth() {
        condition = 24;
    }
    public void doSomethingTwentySixth() {
        condition = 25;
    }
    public void doSomethingTwentySeventh() {
        condition = 26;
    }
    public void doSomethingTwentyEighth() {
        condition = 27;
    }
    public void doSomethingTwentyNinth() {
        condition = 28;
    }
    public void doSomethingThirtieth() {
        condition = 29;
    }
    public void doSomethingThirtyFirst() {
        condition = 30;
    }
    public void doSomethingThirtySecond() {
        condition = 31;
    }
    public void doSomethingThirtyThird() {
        condition = 32;
    }
    public void doSomethingThirtyFourth() {
        condition = 33;
    }
    public void doSomethingThirtyFifth() {
        condition = 34;
    }
    public void doSomethingThirtySixth() {
        condition = 35;
    }
    public void doSomethingThirtySeventh() {
        condition = 36;
    }
    public void doSomethingThirtyEighth() {
        condition = 37;
    }
    public void doSomethingThirtyNinth() {
        condition = 38;
    }
    public void doSomethingFortieth() {
        condition = 39;
    }
    public void doSomethingFortyFirst() {
        condition = 40;
    }
    public void doSomethingFortySecond() {
        condition = 41;
    }
    public void doSomethingFortyThird() {
        condition = 42;
    }
    public void doSomethingFortyFourth() {
        condition = 43;
    }
    public void doSomethingFortyFifth() {
        condition = 44;
    }
    public void doSomethingFortySixth() {
        condition = 45;
    }
    public void doSomethingFortySeventh() {
        condition = 46;
    }
    public void doSomethingFortyEighth() {
        condition = 47;
    }
    public void doSomethingFortyNinth() {
        condition = 48;
    }
    public void doSomethingFiftieth() {
        condition = 49;
    }
    public void doSomethingFiftyFirst() {
        condition = 50;
    }
    public void doSomethingFiftySecond() {
        condition = 51;
    }
    public void doSomethingFiftyThird() {
        condition = 52;
    }
    public void doSomethingFiftyFourth() {
        condition = 53;
    }
    public void doSomethingFiftyFifth() {
        condition = 54;
    }
    public void doSomethingFiftySixth() {
        condition = 55;
    }
    public void doSomethingFiftySeventh() {
        condition = 56;
    }
    public void doSomethingFiftyEighth() {
        condition = 57;
    }
    public void doSomethingFiftyNinth() {
        condition = 58;
    }
    public void doSomethingSixtieth() {
        condition = 59;
    }
    public void doSomethingSixtyFirst() {
        condition = 60;
    }
    public void doSomethingSixtySecond() {
        condition = 61;
    }
    public void doSomethingSixtyThird() {
        condition = 62;
    }
    public void doSomethingSixtyFourth() {
        condition = 63;
    }
    public void doSomethingSixtyFifth() {
        condition = 64;
    }
    public void doSomethingSixtySixth() {
        condition = 65;
    }
    public void doSomethingSixtySeventh() {
        condition = 66;
    }
    public void doSomethingSixtyEighth() {
        condition = 67;
    }
    public void doSomethingSixtyNinth() {
        condition = 68;
    }
    public void doSomethingSeventieth() {
        condition = 69;
    }
    public void doSomethingSeventyFirst() {
        condition = 70;
    }
    public void doSomethingSeventySecond() {
        condition = 71;
    }
    public void doSomethingSeventyThird() {
        condition = 72;
    }
    public void doSomethingSeventyFourth() {
        condition = 73;
    }
    public void doSomethingSeventyFifth() {
        condition = 74;
    }
    public void doSomethingSeventySixth() {
        condition = 75;
    }
    public void doSomethingSeventySeventh() {
        condition = 76;
    }
    public void doSomethingSeventyEighth() {
        condition = 77;
    }
    public void doSomethingSeventyNinth() {
        condition = 78;
    }
    public void doSomethingEightieth() {
        condition = 79;
    }
    public void doSomethingEightyFirst() {
        condition = 80;
    }
    public void doSomethingEightySecond() {
        condition = 81;
    }
    public void doSomethingEightyThird() {
        condition = 82;
    }
    public void doSomethingEightyFourth() {
        condition = 83;
    }
    public void doSomethingEightyFifth() {
        condition = 84;
    }
    public void doSomethingEightySixth() {
        condition = 85;
    }
    public void doSomethingEightySeventh() {
        condition = 86;
    }
    public void doSomethingEightyEighth() {
        condition = 87;
    }
    public void doSomethingEightyNinth() {
        condition = 88;
    }
    public void doSomethingNinetieth() {
        condition = 89;
    }
    public void doSomethingNinetyFirst() {
        condition = 90;
    }
    public void doSomethingNinetySecond() {
        condition = 91;
    }
    public void doSomethingNinetyThird() {
        condition = 92;
    }
    public void doSomethingNinetyFourth() {
        condition = 93;
    }
    public void doSomethingNinetyFifth() {
        condition = 94;
    }
    public void doSomethingNinetySixth() {
        condition = 95;
    }
    public void doSomethingNinetySeventh() {
        condition = 96;
    }
    public void doSomethingNinetyEighth() {
        condition = 97;
    }
    public void doSomethingNinetyNinth() {
        condition = 98;
    }
    public void doSomethingHundredth() {
        condition = 99;
    }
}
```

Program



**Mutation testing**

# Mutation testing: mutant generation

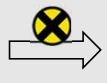
```
1 // ...
2 // ...
3 // ...
4 // ...
5 // ...
6 // ...
7 // ...
8 // ...
9 // ...
10 // ...
11 // ...
12 // ...
13 // ...
14 // ...
15 // ...
16 // ...
17 // ...
18 // ...
19 // ...
20 // ...
21 // ...
22 // ...
23 // ...
24 // ...
25 // ...
26 // ...
27 // ...
28 // ...
29 // ...
30 // ...
31 // ...
32 // ...
33 // ...
34 // ...
35 // ...
36 // ...
37 // ...
38 // ...
39 // ...
40 // ...
41 // ...
42 // ...
43 // ...
44 // ...
45 // ...
46 // ...
47 // ...
48 // ...
49 // ...
50 // ...
51 // ...
52 // ...
53 // ...
54 // ...
55 // ...
56 // ...
57 // ...
58 // ...
59 // ...
60 // ...
61 // ...
62 // ...
63 // ...
64 // ...
65 // ...
66 // ...
67 // ...
68 // ...
69 // ...
70 // ...
71 // ...
72 // ...
73 // ...
74 // ...
75 // ...
76 // ...
77 // ...
78 // ...
79 // ...
80 // ...
81 // ...
82 // ...
83 // ...
84 // ...
85 // ...
86 // ...
87 // ...
88 // ...
89 // ...
90 // ...
91 // ...
92 // ...
93 // ...
94 // ...
95 // ...
96 // ...
97 // ...
98 // ...
99 // ...
100 // ...
```

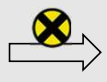
Program

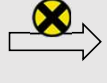


**Mutation testing**



*Lhs < rhs*  *Lhs <= rhs*

*Lhs < rhs*  *Lhs != rhs*

*stmt*  *no-op*

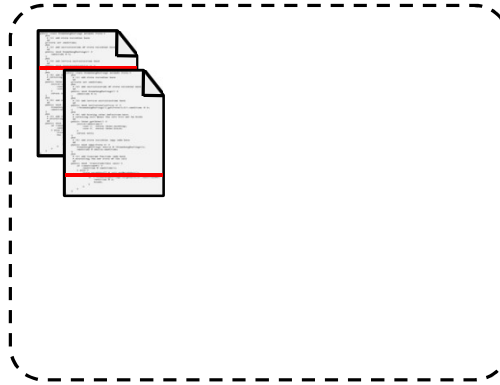
Mutation operators



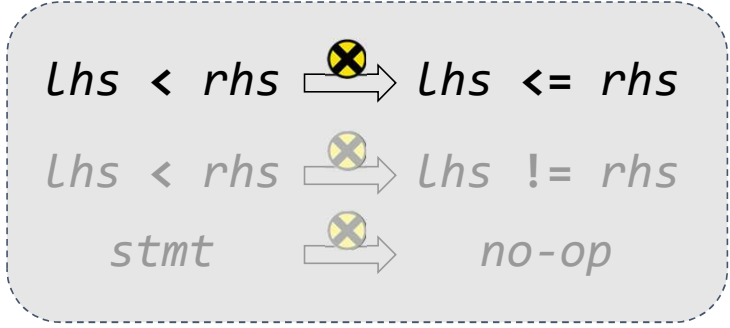
# Mutation testing: mutant generation

```
1 // class Stromungsbetrag mit einem Betrag *
2 //
3 // @author
4 //
5 // @see
6 //
7 // @since
8 //
9 // @version
10 //
11 //
12 //
13 //
14 //
15 //
16 //
17 //
18 //
19 //
20 //
21 //
22 //
23 //
24 //
25 //
26 //
27 //
28 //
29 //
30 //
31 //
32 //
33 //
34 //
35 //
36 //
37 //
38 //
39 //
40 //
41 //
42 //
43 //
44 //
45 //
46 //
47 //
48 //
49 //
50 //
51 //
52 //
53 //
54 //
55 //
56 //
57 //
58 //
59 //
60 //
61 //
62 //
63 //
64 //
65 //
66 //
67 //
68 //
69 //
70 //
71 //
72 //
73 //
74 //
75 //
76 //
77 //
78 //
79 //
80 //
81 //
82 //
83 //
84 //
85 //
86 //
87 //
88 //
89 //
90 //
91 //
92 //
93 //
94 //
95 //
96 //
97 //
98 //
99 //
100 //
```

Program



Mutants

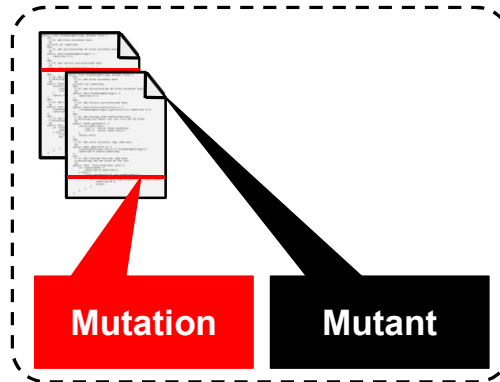


Mutation operators

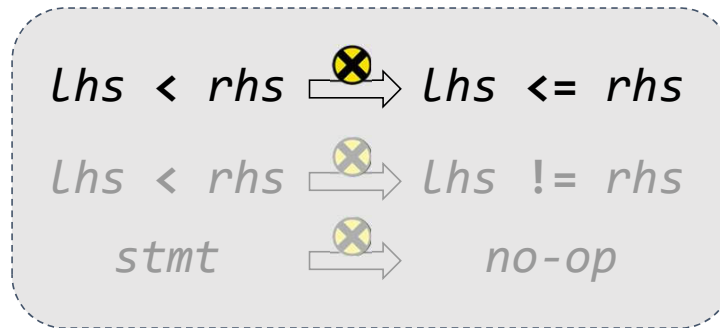
# Mutation testing: mutant generation

```
1 // ...
2 // ...
3 // ...
4 // ...
5 // ...
6 // ...
7 // ...
8 // ...
9 // ...
10 // ...
11 // ...
12 // ...
13 // ...
14 // ...
15 // ...
16 // ...
17 // ...
18 // ...
19 // ...
20 // ...
21 // ...
22 // ...
23 // ...
24 // ...
25 // ...
26 // ...
27 // ...
28 // ...
29 // ...
30 // ...
31 // ...
32 // ...
33 // ...
34 // ...
35 // ...
36 // ...
37 // ...
38 // ...
39 // ...
40 // ...
41 // ...
42 // ...
43 // ...
44 // ...
45 // ...
46 // ...
47 // ...
48 // ...
49 // ...
50 // ...
51 // ...
52 // ...
53 // ...
54 // ...
55 // ...
56 // ...
57 // ...
58 // ...
59 // ...
60 // ...
61 // ...
62 // ...
63 // ...
64 // ...
65 // ...
66 // ...
67 // ...
68 // ...
69 // ...
70 // ...
71 // ...
72 // ...
73 // ...
74 // ...
75 // ...
76 // ...
77 // ...
78 // ...
79 // ...
80 // ...
81 // ...
82 // ...
83 // ...
84 // ...
85 // ...
86 // ...
87 // ...
88 // ...
89 // ...
90 // ...
91 // ...
92 // ...
93 // ...
94 // ...
95 // ...
96 // ...
97 // ...
98 // ...
99 // ...
100 // ...
```

Program



Mutants



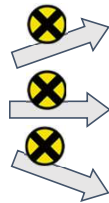
Mutation operators



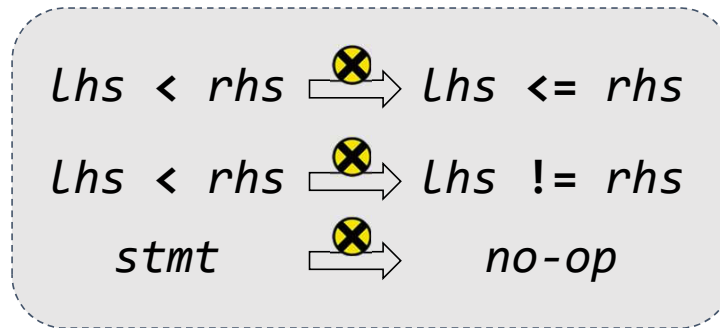
# Mutation testing: mutant generation



Program

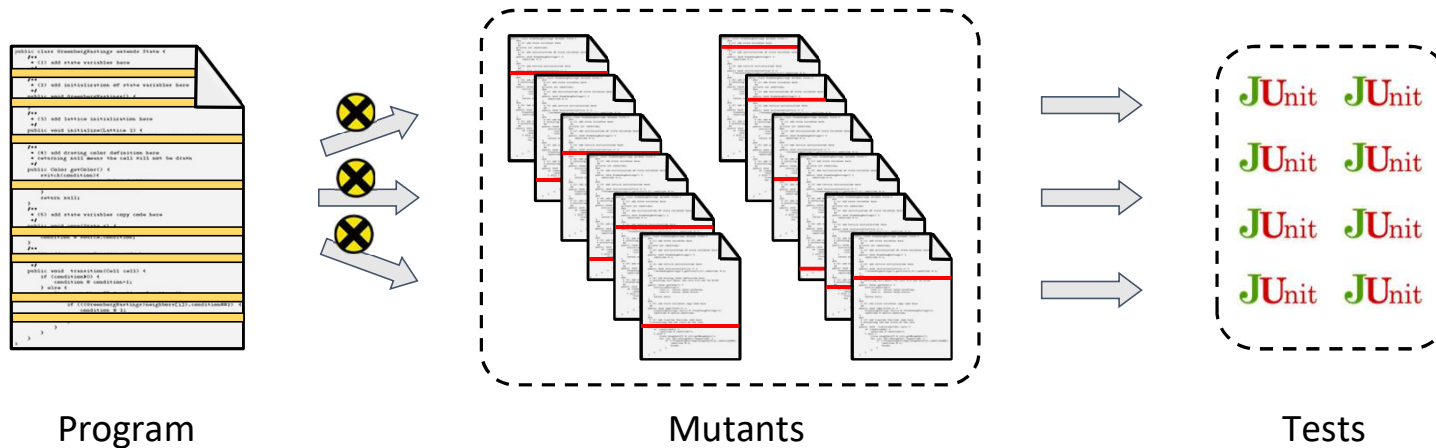


Mutants



Mutation operators

# Mutation testing: test creation



## 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/mutants_real_faults_fse_2014.pdf),

[https://homes.cs.washington.edu/~rjust/publ/mutation\\_testing\\_practices\\_icse\\_2021.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;

<b><i>a</i></b>	<b><i>b</i></b>	<b>Original</b>	<b>M1</b>	<b>M2</b>	<b>M3</b>	<b>M4</b>
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;

<b>a</b>	<b>b</b>	<b>Original</b>	<b>M1</b>	<b>M2</b>	<b>M3</b>	<b>M4</b>
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;

<i>a</i>	<i>b</i>	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;
```

<i>a</i>	<i>b</i>	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)**

<b>a</b>	<b>b</b>	<b>Original</b>	<b>M1</b>	<b>M2</b>	<b>M3</b>	<b>M4</b>
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?

<i>a</i>	<i>b</i>	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

<i>a</i>	<i>b</i>	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

<i>a</i>	<i>b</i>	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

Redundant

Equivalent

# 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

```
int RunMe(int a, int b) {  
  if (a == b || b == 1) {  
    ~  
    7  
    8
```

▼ Mutants 14:25, 28 Mar

Changing this 1 line to

```
    if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#) [Not useful](#)

*Practical Mutation Testing at Scale: A view from Google* ([Reading](#))

# Productive mutants: mutation testing at Google

```
int RunMe(int a, int b) {  
  if (a == b || b == 1) {  
    ~  
    8
```

▼ Mutants  
14:25, 28 Mar

Changing this 1 line to

```
if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#) [Not useful](#)

*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?

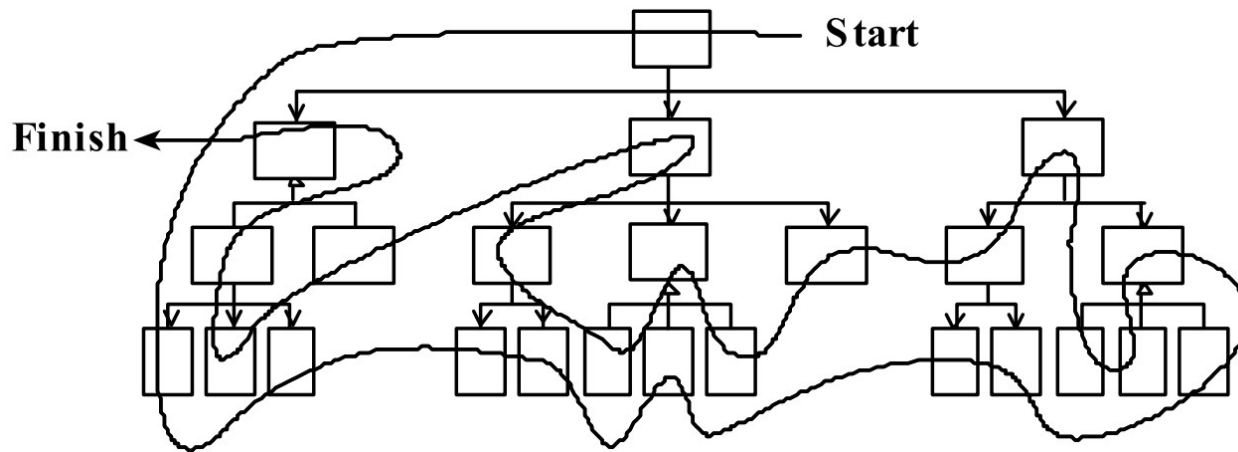
- **Incremental integration** benefits:
  - Errors easier to isolate, find, fix
    - reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - good for customer, developer morale
- But it isn't without challenges:
  - May need to create "**stub**" versions of some 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

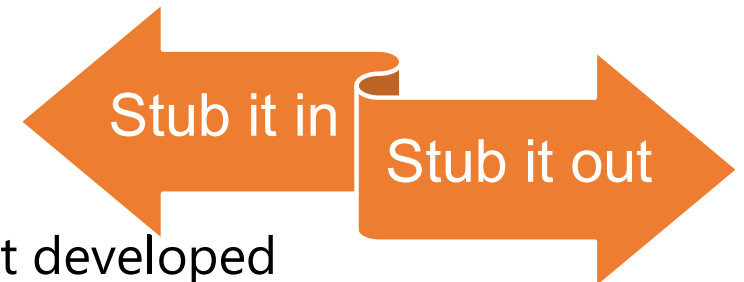
# Additional reference material



# What's a stub?

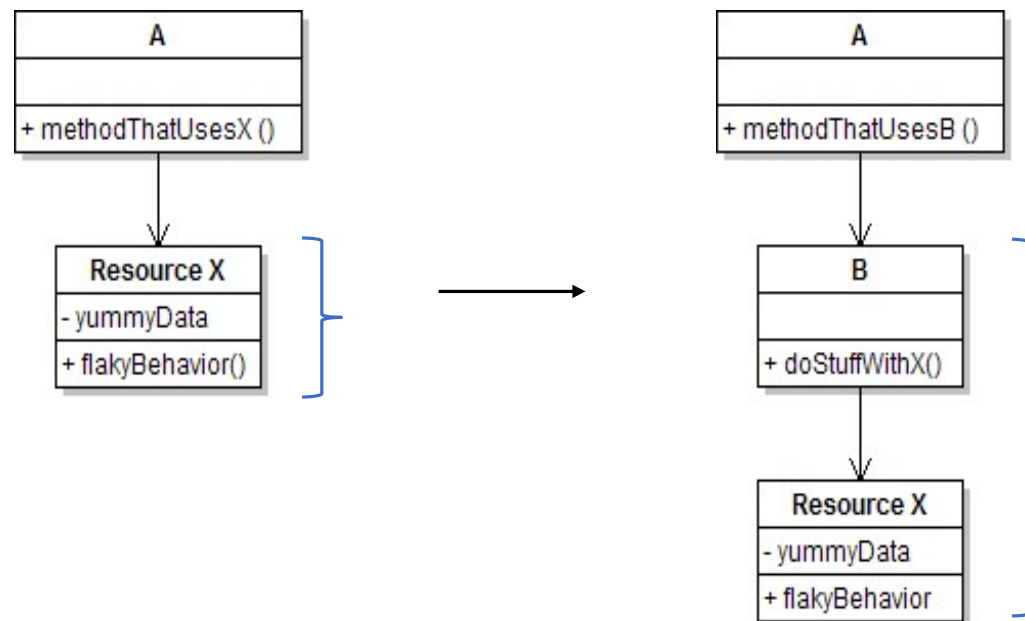
**Stub:** a controllable replacement for a software unit

- 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

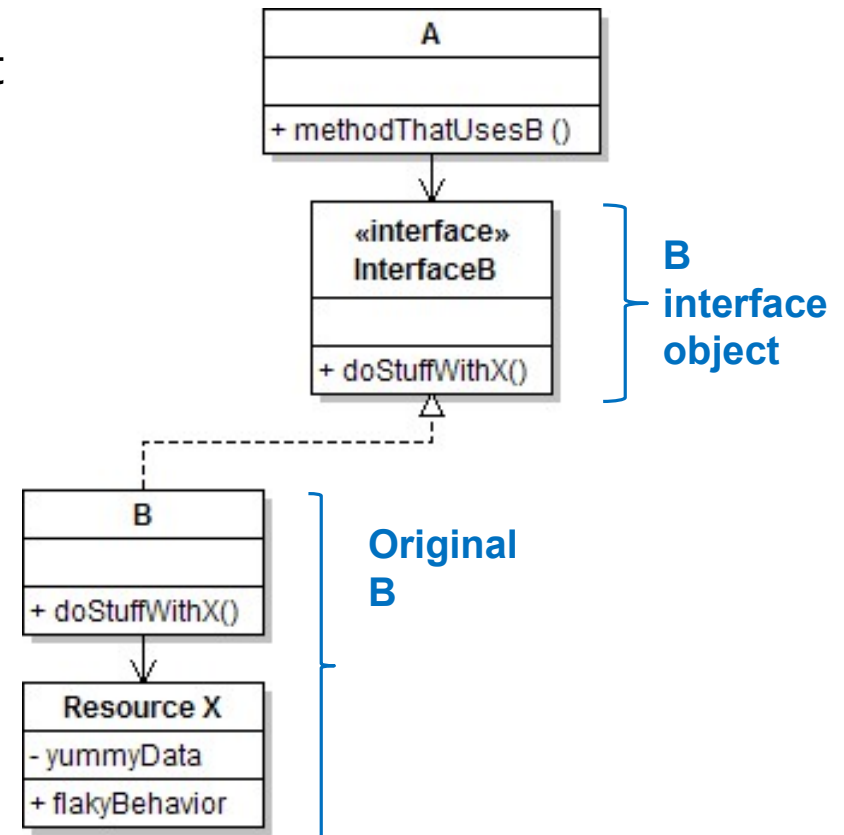
Create Class B to represent the challenging/missing dependency (as needed)

# 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

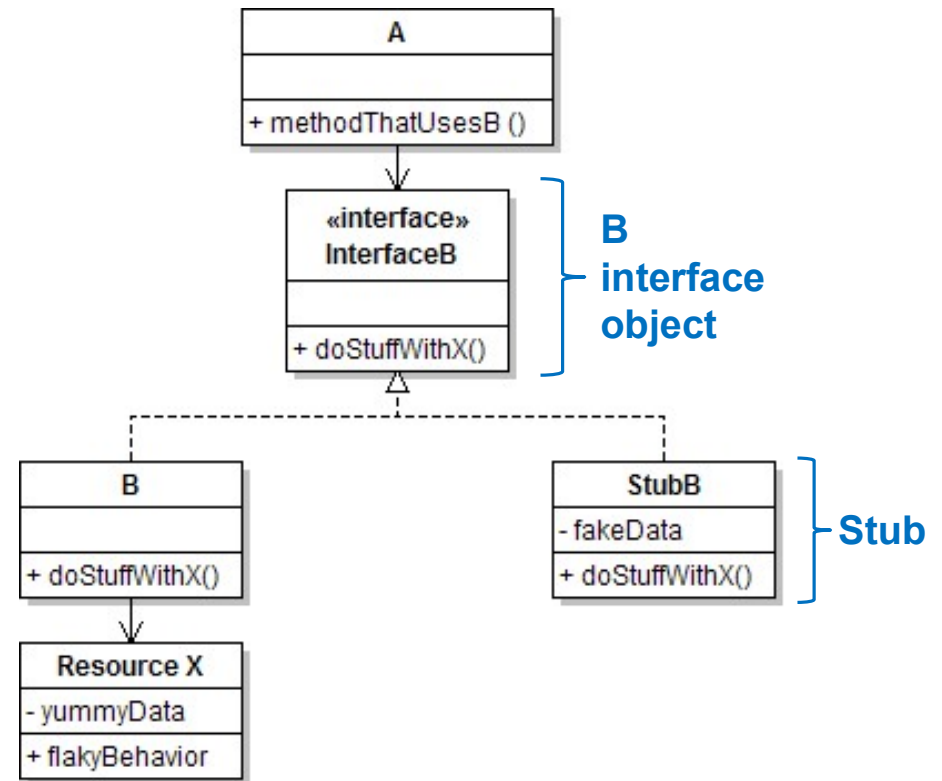


# Create a stub, step 3

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()** );

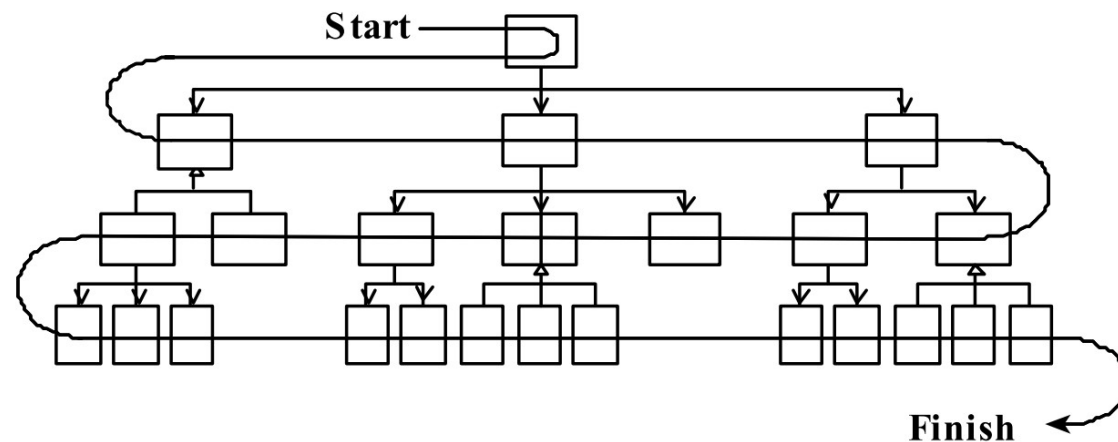
Think about how to minimize code changes when you no longer depend on the stub

# 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?)

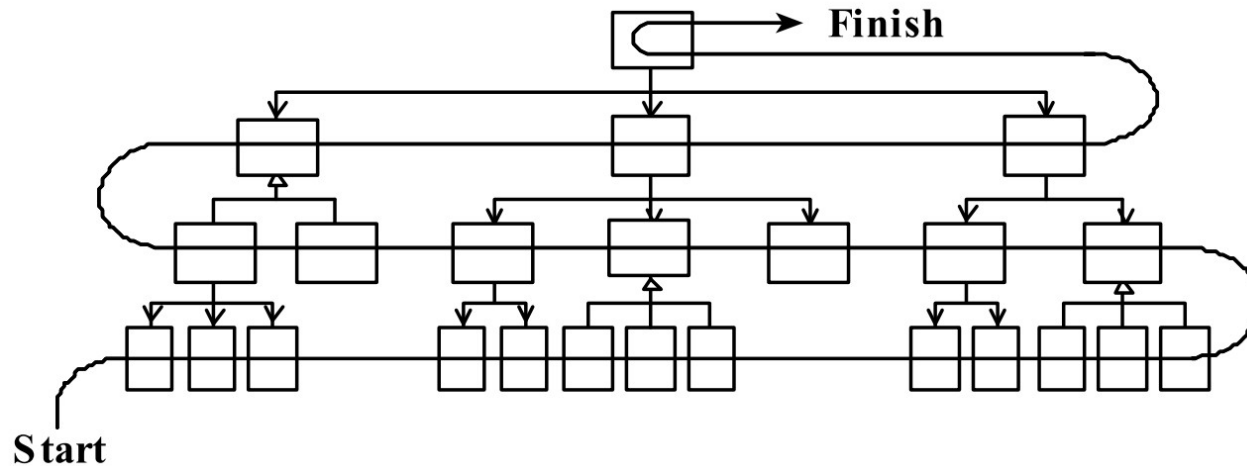


# Or bottom-up

## 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**