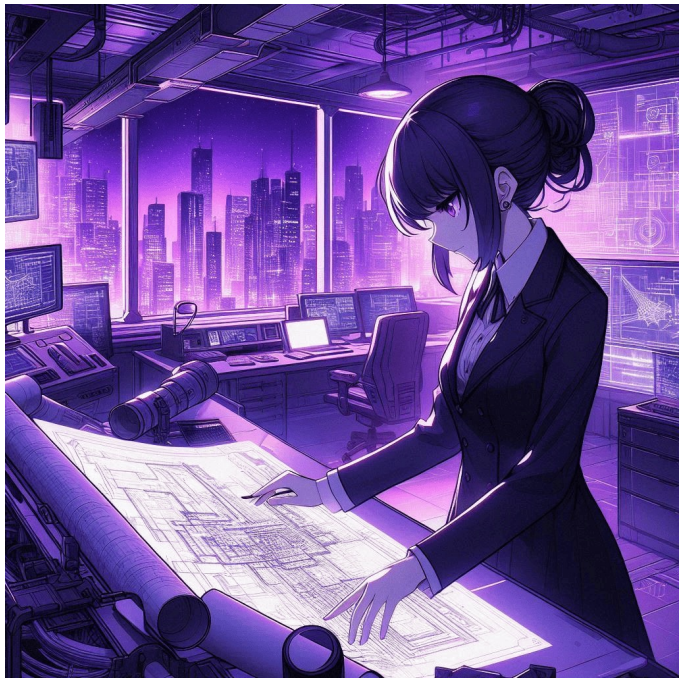


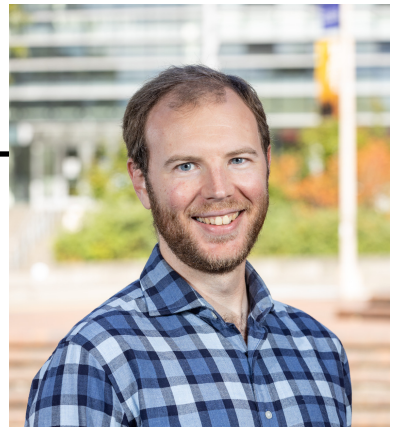
CSE 331

Specifications & Testing

James Wilcox

with thanks to Kevin Zatloukal for many slides

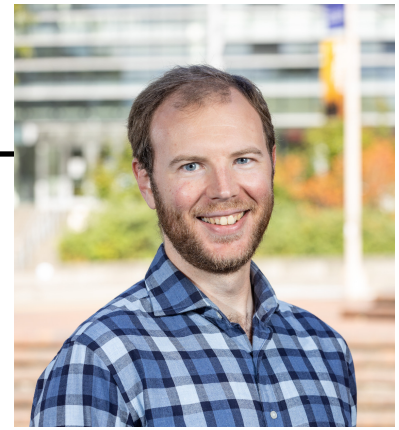
About Me



James Wilcox

- Asst Teaching Prof in CSE
  - did my PhD here 2013-2019
- Interested in:
  - programming languages
  - distributed systems
  - teaching programming
  - music (mostly choir)
  - running

About Me



James Wilcox

- Asst Teaching Prof in CSE
  - did my PhD here 2013-2019
- Built a wide range of systems and applications

Systems

- compilers
- operating systems
- distributed systems
- networking systems
- database systems
- graphics
- ...

Applications

- desktop apps
- web apps
- phone apps
- IDE
- games
- ...

About You

- Familiar with Java at the level of 123
  - understand primitive vs reference types
  - familiar with recursive functions
- Probably in your first/second year of the major

About This Class

- **Very little content change vs 10 years ago**
  - **maybe 10% changes based on who teaches it**

- **Substantial homework changes**
  - **made a big change ~3 years ago**

    typical student went from senior to sophomore
  - **making another change this year**

    need to consider the role of AI
  - **making another substantial change this quarter**

    integrating AI in programming

"Coding, or the translation of a precise design into software instructions, is dead. AI can do that."

— Magda Balazinska

About This Class

"Coding, or the translation of a precise design into software instructions, is dead. AI can do that."

— Magda Balazinska

Coding may be dead,

but software engineering is alive!

About This Class

"Coding, or the translation of a precise design into software instructions, is dead. AI can do that."

— Magda Balazinska

- We will focus what comes before and after
- Before/while coding, write a *precise* design
- After coding, check that it is correct

Before Coding

- A precise design requires a precise "specification"
  - says exactly what input/output behavior is expected
  - AI cannot read your mind!
- A correct specification should:
  - rule out every implementation you don't want
  - rule out no implementation you would accept
- Surprisingly difficult to do

Before Coding

- A precise design requires a precise "specification"
  - says exactly what input/output behavior is expected
- Also want our designs to lead to code that is
  - easy to **understand**
  - easy to **test**
  - easy to **change**
- Will look at ways to achieve this

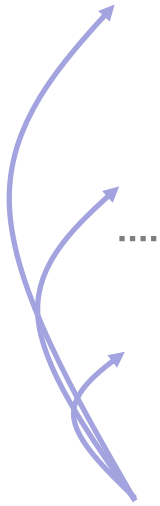
After Coding

- After coding, you must ensure it is correct
- How do you do that?
- Trying on a few examples is insufficient
  - if this is enough, then AI can do that too
  - Dijkstra did not just try his algorithm on a few graphs

After Coding

- After coding, you must ensure it is correct
- How do you do that?
- CS standard is a proof of correctness
  - only way to know the code is *fully* correct for *all* inputs
  - we will show you what it takes to do this

Software Development Process



1. Specification *what*
2. Plan Generation *how*

Design

3. Coding
4. Correctness

Implementation

Exams

- Midterm in class Friday, May 8

- Final exam
  - on Tuesday, June 9th at 2:30
  - in this room

A software engineer's job is to produce code that works

You must know that it works

Knowledge about programs

- What does it mean to know that code works?
  - what does it mean to know anything about code?

- We are interested in knowledge about *behavior*
  - could just run it and see what happens

What can you tell me about this program?

```
public void myProgram() {
    int x = 3;
    int y = 4;
    System.out.println(x + y);
}
```

Not much without seeing the rest of the program!

What can you tell me about this program? (1)

```
public static void main(String[] args) {
    int x = 3;
    int y = 4;
    System.out.println(x + y);
}
```

It's a program with exactly 1 behavior:
print 7 (and then exit)

What can you tell me about this program? (2)

```
public static void main(String[] args) {
    int x = 3;
    int y = 4;
    // (***)
    System.out.println(x + y);
}
```

What do we know at the line marked (***)?

x is 3 and y is 4

- **note how these are internal facts about variables, not user-observable/behavioral facts**

Abstraction

- An abstraction hides details in part of the code
  - "high level" description, avoiding "low level" details
- Makes the code more **understandable**
  - client does not need to learn those details
- Makes the code more **changeable**
  - implementer can change details hidden from client

client

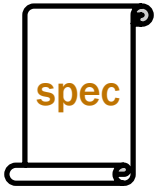
implementer

Procedural Abstraction

- Provided for a method via its specification
  - must describe the input/output behavior
  - need not describe every detail of how it works

- Usually called "procedural abstraction"
  - method aka function aka procedure aka routine
  - we will see other kinds in the future...

client



implementer

Abstraction Barrier

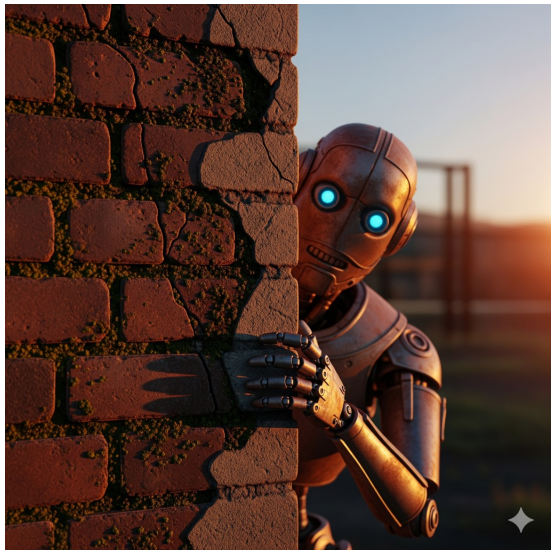
- "Abstraction barrier" between client & implementer
  - implementer promises code satisfies the specification
    free to choose any implementation within those constraints
  - client promises not to depend on hidden details
    will depend only on details included in the specification

Specifications

- A good specification is precise
  - cannot have confusion about required behavior
- A good specification…
  - hides details that may **change**
  - hides details that are hard to **understand**
  - provides enough detail to be **useful**
- Creating abstractions requires judgement
  - no sure-fire formula for how to do this
  - none of us can perfectly foresee future changes

AI and Abstraction

- At present, AI does not respect abstraction barriers
  - it should be possible to fix this in the future
  - for now, you have to police this yourself



- Writing specs is necessary
  - must distinguish details that are incidental vs essential
  - AI cannot read your mind!

What can you tell me about this program? (3)

```
public static void main(String[] args) {
    int x = 3;
    int y = 4;
    // (***)
    System.out.println(x + y);
}
```

What do we know at the line marked (***)?

x is 3 and y is 4

- **note how these are internal facts about variables, not user-observable/behavioral facts**

What can you tell me about this program? (4)

```
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    System.out.println("What is your name?");
    String name = s.nextLine();
    System.out.println("Hello, " + name + "!");
}
```

- It's a program with *infinitely many* behaviors!
- All can be described as:
  - read in the user's name from the keyboard
  - print the greeting "Hello, <NAME>!"

What can you tell me about this program? (4)

```
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    System.out.println("What is your name?");
    String name = s.nextLine();
    // (***)
    System.out.println("Hello, " + name + "!");
}
```

What do we know at the line marked (***)?

- **(must be true on *all* executions)**
- `name` **contains the user's input**

Assertions

Assertions

An *assertion* is:

- a true/false claim about the program state
- at a particular program point

An assertion *holds* if

claim is true at that point *on all executions*

A claim about *all* executions requires proof,

(not testing)

Simple Example Assertion

```
public static void main(String[] args) {
    int x = 3;
    int y = 4;
    {{x is 3 and y is 4}}
    System.out.println(x + y);
}
```

We write assertions in code like this:

```
{{x is 3 and y is 4}}
```

A More Complex Example

```
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    System.out.println("Enter some numbers.");
    List<Integer> list = new ArrayList<>();
    while (s.hasNextInt()) {
        list.add(s.nextInt());
    }
    System.out.println("Got " + list);
    int sum = 0;
    for (int i = 0; i < list.size(); i++) {
        sum += list.get(i);
    }
    System.out.println("The sum of your list was: " + sum);
}
```

What is this program's user-facing behavior?

Read in a list of numbers, print them and their sum.

A More Complex Example

```
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    System.out.println("Enter some numbers.");
    List<Integer> list = new ArrayList<>();
    while (s.hasNextInt()) {
        list.add(s.nextInt());
    }
    // (1)  {{list contains the user's input list of numbers}}
    System.out.println("Got " + list);
    int sum = 0;
    for (int i = 0; i < list.size(); i++) {
        sum += list.get(i);
    }
    // (2) {{sum contains the sum of the elements of list}}
    System.out.println("The sum of your list was: " + sum);
}
```

What assertions could you make at points (1) and (2)?

What assertions tell us

If the assertion holds, it means that

whenever the program reaches that point,

then the claim is true.

We can "reason between" assertions.

```
...
{{  P  }}
code C
{{  Q  }}
```

- If assertion P holds at its point, and
- whenever P is true,
  running C2 causes Q to be true
- Then assertion Q holds at its point
- (Note we don't need to know what "..." is)

Specifications

Methods and Assertions

```
private static List<Integer> readList() {
    Scanner s = new Scanner(System.in);
    System.out.println("Enter some numbers.");
    List<Integer> inputList = new ArrayList<>();
    while (s.hasNextInt()) {
        inputList.add(s.nextInt());
    }
    return inputList; }
private static int sumList(List<Integer> listToSum) {
    int sum = 0;
    for (int i = 0; i < listToSum.size(); i++) {
        sum += listToSum.get(i);
    }
    return sum; }
public static void main(String[] args) {
    List<Integer> list = readList();
    // (1)
    System.out.println("Got " + list);
    int sum = sumList(list);
    // (2)
    System.out.println("The sum of your list was: " + sum); }
```

What can you say at (1) and (2)?

Method Specifications in General

- Specification consists of two parts
  - precondition says what inputs are allowed
  - postcondition says what result for allowed inputs
- Client promises only to pass allowed inputs
  - inputs will satisfy the precondition
- Implementer promises results will be as expected
  - outputs will satisfy the postcondition
  - no promises if the precondition does not hold!

Writing Method Specifications in Java

- Java writes method specs in special comments
  - immediately before the method
  - using `/** .. */` comment format

- Produces HTML documentation from comments

contains

```
public boolean contains(CharSequence s)
```

Returns true if and only if this string contains the specified sequence of char values.

Parameters:

`s - the sequence to search for`

Returns:

`true if this string contains s, false otherwise`

Since:

`1.5`

incredibly important feature of Java

Writing Method Specifications in Java

- Each Javadoc comment includes
  - an overview sentence
  - explanations of each parameter
  - explanations of what is returned
  - any exceptions thrown and, if so, under what conditions

contains

```
public boolean contains(CharSequence s)
```

Returns true if and only if this string contains the specified sequence of char values.

Parameters:

`s - the sequence to search for`

Returns:

`true if this string contains s, false otherwise`

Since:

`1.5`

Writing Method Specifications in Java

- Overview comment is the first sentence
  - shown in the IDE when you hover over method name
  - followed by any amount of additional explanation

- Other parts are included with `@tag` syntax
  - **`@param`** `name` explains what name is for
  - **`@return`** explain in detail what return values mean
  - **`@throws`** `E` explain when exception `E` is thrown

- We will add some others...

Writing Method Specifications in Java

- **Preconditions** on individual parameters
  - often included in the `@param`

```
@param n Name to look for. Must be non-null
```

- **Preconditions** on multiple parameters
  - do not fit well in standard Javadoc tags
  - we will add our own `@requires`

```
@param i An index in the array
@param A The array to look in
@requires 0 <= i < A.length
```

Writing Method Specifications in Java

- **Method call either returns or throws an exception**
  - **one or the other, not both**
  - **describe in `@return` and `@throws`**

```
@param n The number to look for
@return an index i such that A[i] = n
@throws NotFound if n is not in A
```

- **Postcondition only specified for allowed inputs**
  - **contradictory to disallow and then say what happens**

```
@requires 0 <= i < A.length
@throws InvalidArg if 0 < i or A.length <= i
```

Writing Method Specifications in Java

- Every input falls in one of three cases:
  1. input is disallowed
  2. input is allowed and will return something
  3. input is allowed and will throw something

- Item 1 is the precondition
  - explained in `@param` and `@requires`

- Items 2-3 are the postcondition
  - explained in `@return` and `@throws`

Example 1

```
/**
 * Returns the index of a number in the list.
 * @param A The list to look in. Must be non-null
 * @param v The value to look for
 * @return an index i such that A[i] = v
 * @throws NotFound if no such index exists
 */
int indexOf(List<Integer> A, int v);
```

What is are other reasonable specifications?

Example 2

```
/**
 * Returns the concatenation of two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @return a list containing the elements of A
 *     followed by all the elements of B.
 */
List<Integer> concat(
    List<Integer> A, List<Integer> B);
```

What is another reasonable specification?

Example 3

```
/**
 * Adds the elements in two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @requires A.size() = B.size()
 * @return list C of the same length, where
 *    C's value is the sum of those at the
 *    corresponding indexes in A and B
 */
List<Integer> addLists(
    List<Integer> A, List<Integer> B);
```

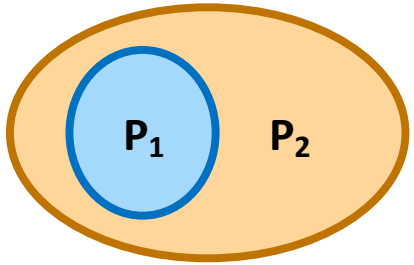
Comparing Specifications

Comparing Assertions

- We say that P_1 is stronger than P_2...
  - whenever P_1 is true, P_2 is also true
  - values satisfying P_1 are a subset of those for P_2

- We say that P_2 is weaker than P_1

311 alert: P_1 implies P_2



Comparing Specifications

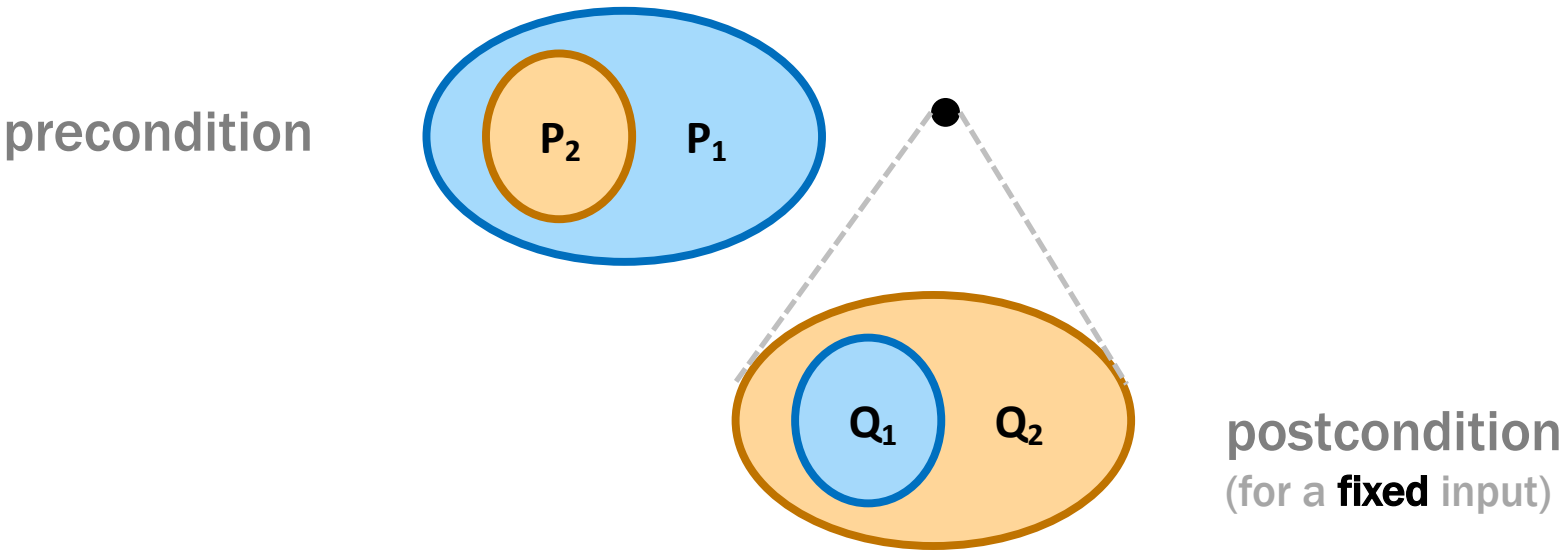
- Would like a similar definition for specifications

- Specification S_1 is stronger than S_2...
  - whenever is S_1 satisfied, S_2 is also satisfied
  - i.e., satisfying S_1 implies satisfying S_2

- But what does this mean?
  - specifications have a precondition and postcondition

Comparing Specifications

- Specification S_1 is stronger than S_2 if it has…
  - a stronger (or equal) postcondition, and
  - a weaker (or equal) precondition



Comparing Specifications in Java

- Specification S_1 is stronger than S_2 if it has...

- A stronger postcondition:
  - smaller subset of allowed outputs for a given input
  - e.g., return value "is between 1 and 100" is stronger than "is positive"

- A weaker precondition:
  - larger subset of allowed inputs
  - e.g., allowing all integer values instead of positive ones

Comparing Specifications

- Specification S_1 is **stronger** than S_2 if it has...
  - a **stronger** (or equal) postcondition, and
  - a **weaker** (or equal) precondition

- Not all specifications are stronger or weaker
  - all others are called "incomparable"
  - most pairs of specifications are incomparable

    stronger and **weaker** are special cases

Example 4

```
int indexOf(List<Integer> A, int v);

// Specification A
// @requires value v occurs somewhere in A
// @return an index i such that A[i] = v

// Specification B
// @requires value v occurs somewhere in A
// @return the smallest i such that A[i] = v
```

How does A relate to B?

Example 5

```
int indexOf(List<Integer> A, int v);

// Specification A
// @requires value v occurs somewhere in A
// @return an index i such that A[i] = v

// Specification C
// @return an index i such that A[i] = v
//     if v appears in A and otherwise -1
```

How does A relate to C?

Example 6

```
int indexOf(List<Integer> A, int v);

// Specification B
// @requires value v occurs somewhere in A
// @return the smallest i such that A[i] = v

// Specification C
// @return an index i such that A[i] = v
//      if v appears in A and otherwise -1
```

How does B relate to C?

Why Do We Care?

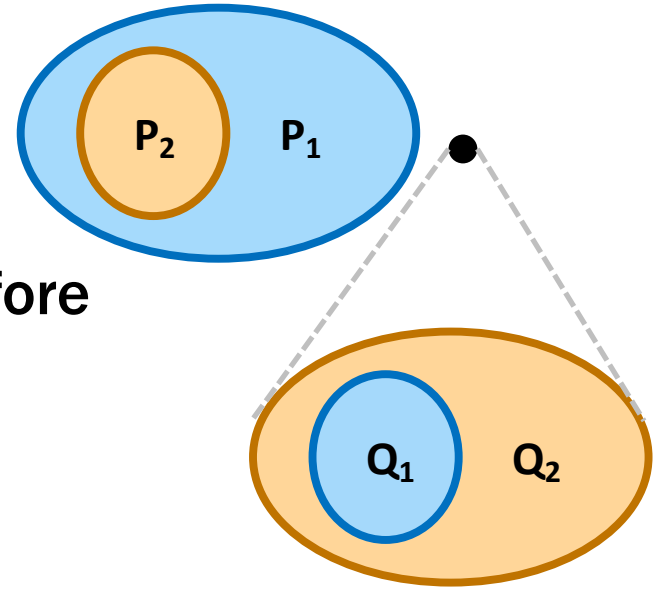
- Specification S_1 is stronger than S_2 if it has...
  - a weaker precondition
  - a stronger postcondition

- Changing from S_2 to S_1 (strengthening)...
  - cannot break any clients!
  - only allows more inputs

    old inputs in P_2 are still in P_1
  - all outputs were also allowed before

    clients' code was ready to handle all Q_2

    this includes all Q_1 also



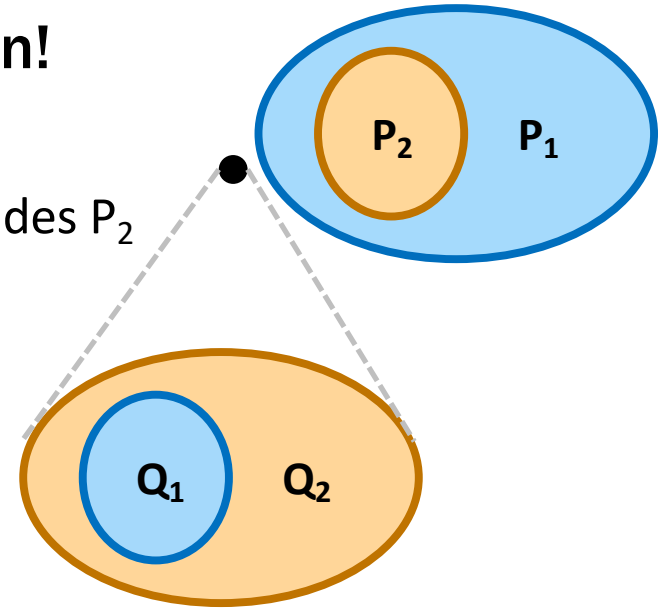
Why Do We Care?

- Specification S_1 is stronger than S_2 if it has…
  - a weaker precondition
  - a stronger postcondition

- Changing from S_1 to S_2 (weakening)…
  - cannot break the implementation!
  - allows fewer inputs

    current code handled all P_1, which includes P_2
  - allows more outputs

    current code returns values in Q_1, which is still inside of Q_2



Why Do We Care?

- Specification S_1 is stronger than S_2 if it has…
  - a weaker precondition
  - a stronger postcondition
- Changing from S_1 to S_2 (weakening)…
  - cannot break the implementation!
- Changing from S_2 to S_1 (strengthening)…
  - cannot break any clients
- Tells us what we need to fix with this change

Which Is Better?

- Changing from S_1 to S_2 (weakening)…
  - cannot break the implementation!
- Changing from S_2 to S_1 (strengthening)…
  - cannot break any clients!

- In principle, neither stronger nor weaker is better

- With few clients, weakening is easier
- With many clients, strengthening is easier
  - with 1,000 clients, weakening is *impossible*

Varieties of Testing

Unit vs Integration Tests

- A unit test checks one function
  - ideally, without testing anything else (not always possible)
- An integration test makes sure units work together
  - many (most?) bugs in practice are here
- An end-to-end test exercises almost all the code

Unit vs Integration Tests

- A unit test checks one function
- An integration test makes sure units work together
- An end-to-end test exercises almost all the code

- You will be expected to write unit tests in industry

- There will also be integration and end-to-end tests
  - someone will write them, but maybe not you
  - (requires understanding the whole system)

- Today we will focus on unit testing

Unit Testing

- **Even individual functions might be too big…**

```
int f(List<Integer> vals) {
  Map<Integer, List<Integer>> M = …;
  for (int v : vals) {
    …
  }

  int s = 0;
  for (int v : vals)  {
    for (int w : M.get(v))
      s += …
  }
}
```

Unit Testing

- Even individual functions might be too big…

```
int f(List<Integer> vals) {
  for (int v : vals)
    …
  for (int v : vals)
    …
}
```

- Multiple loops often should be multiple functions

- Purposefully design the code to be testable
  - important part of programming in practice

“Manual” vs Programmatic Tests

- Usually possible to run the code by hand (“manually”)
  - open the terminal and execute it
  - start the application and look at it (UI)
- No downside… unless the code changes
  - then, you need to do the tests again
- Programmatic tests are code that tests other code
  - easy to run them again whenever the code changes
  - these are generally preferred

“Manual” vs Programmatic Tests

- Usually possible to run the code by hand (“manually”)
  - open the terminal and execute it
  - start the application and look at it (UI)
- No downside… unless the code changes
  - then, you need to do the tests again
- For UI, manual testing is still common
  - written tests are hard to write and imperfect

    need to see it on the screen to be sure that it looks right
  - non-UI functions and all server code tested programmatically

Unit Testing

Writing a Programmatic Test

1. Choose an input
   - description of the inputs is the “test case”

2. Think through what the answer should be
   - look at the specification for the correct answer
   - if you run the code to get the answer, you are not testing

AI often does this
(and worse)

Writing a Programmatic Test

1. Choose an input / configuration
   - description of the inputs / configuration is the "test case"

2. Think through what the expected answer is
   - if you run the code to get the answer, you are not testing

3. Write code that
   a) calls the function that input
   b) compares the actual answer to the expected one
   c) throws an error if they do not match
   - useful libraries for doing this…

Writing a Programmatic Test

```
public class NumberUtil {

  // Determines whether n is a prime number.
  public static boolean isPrime(int n) {
    …
  }

  // Returns the greatest common divisor of a and b.
  public static int gcd(int a, int b) {
    …
  }
}
```

Writing a Programmatic Test with JUnit

```
import org.junit.*;
import static org.junit.Assert.*;

public class NumberUtilTest {
  @Test
  public void testIsPrime() {
    assertEquals(true, isPrime(2));
    assertEquals(true, isPrime(3));
    assertEquals(false, isPrime(4));
  }

  @Test
  public void testGcd() {
    assertEquals(1, gcd(3, 2));
    assertEquals(3, gcd(9, 3));
    assertEquals(4, gcd(12, 8));
  }
}
```

`assertEquals` will throw an exception and fail the test if the two are not equal

use **`gradlew test`** to run all the tests in the Java project

Ground Rules for Testing

1. Only need to test inputs allowed by the spec
   - there is no correct answer for other inputs

```
// Determines whether a positive integer is prime.
public static boolean isPrime(int n) {
  if (n <= 0)
    throw new IllegalArgumentException("negative n");
  …
}
```

good use of defensive programming
to check that the input is valid

Ground Rules for Testing

1. Only need to test inputs allowed by the spec
   - there is no correct answer for other inputs

2. Choose tests for each function individually
   - pick tests to do a good job of testing that one function

```
// Determines whether a positive integer is prime.
public static boolean isPrime(int n) {
  if (n <= 0)
    throw new IllegalArgumentException("negative n");

  int m = intSqrt(n);  // integer square root of n
  …
}
```

`intSqrt` has its own tests!

Ground Rules for Testing

1. Only need to test inputs allowed by the spec
   - there is no correct answer for other inputs

2. Test each function individually
   - assume anything it calls is correct (its own tests will check)

3. Test code should be **simple**
   - any loops in tests need their own tests!

How Many Tests are Necessary?

- Consider the following function:

```
// Allows inputs 1 <= a, b, c <= 10,000 …
public static int f(int a, int b, int c) {
  …
};
```

- How many tests needed guarantee correctness?
  - 1 trillion!
  - "just write a loop and …"

    the code in that loop could also be wrong
  - cannot **think** through even 1000 tests

    most code we write cannot be *exhaustively* tested

Ground Rules for Testing

1. Only need to test inputs allowed by the spec
   - there is no correct answer for other inputs

2. Test each function individually
   - assume anything it calls is correct (its own tests will check)

3. Test code should be **simple**
   - any loops in tests need their own tests!

4. If there are fewer than 10 allowed inputs, then test them all!
   - take advantage of the easy case

Choosing Test Cases

```
// Returns true iff n is a prime number
public static boolean isPrime(int n) { … }
```

- How about if we test 2, 3, 4, 7, 12, 97, 99?
  - seems okay?

Choosing Test Cases

```
// Returns true iff n is a prime number
public static boolean isPrime(int n) {
  if (n < 100) {
    return PRIME_CACHE[n];  // precomputed answers
  } else {
    for (int k = 2; k*k <= n; k++) {
      if (n % k === 0)
        return false;
    }
    return true;
  }
};
```

Cases 2 .. 100 are table lookups!

We didn't test the loop at all!

Impossible to know this without looking at the actual code.

Clear-Box Testing

- We need to look at the code to know what to test
  - this is called "clear-box testing"
  - it will be our **primary** heuristic
- In this class, I want a clear rule for how many tests
  - want homework and tests to have clear right/wrong answers
- Outside of class, these rules are also good
  - most programmers will be familiar with these concepts

Statement Coverage

- Simplest metric is "statement coverage"
  - what percentage of the statements in the code are executed by *at least one* test

    in our math notation, think of each **case** in the definition as a statement
  - this should be nearly 100%

```
// Determines whether a positive integer is prime.
public static boolean isPrime(int n) {
  if (n <= 0)
    throw new IllegalArgumentException("negative n");

  …
```

- The "`throw`" is not executed by any *allowed* input
  - we only test the allowed inputs

Statement Coverage

- Simplest metric is "statement coverage"
  - what percentage of the statements in the code are executed by *at least one* test
- Must test 100% of code reachable on allowed inputs
  - cannot send code to users that you didn't even try!
  - we will refer to this as having "full statement coverage"
- Are we done?

Statement Coverage

- Consider the following function:

```
// Returns the smaller of a and b.
public static int min(int a, int b) {
  int m = a;
  if (a <= b)
    m = a;
  return m;
};
```

  - testing on a=1 b=2 gives full statement coverage
  - what is the bug?

    gives the wrong answer whenever a > b
  - we never tested the case where the "`if`" doesn't execute

Conditionals

Conditionals are "`if`" statements

```
if (n > 0) {
  x = 2*(n - 1);
} else {
  x = 0;
}
```

Every conditional has two branches (“then” and “else”)

Conditionals

Conditionals are "`if`" statements

```
if (n > 0) {          =     if (n > 0) {
  x = 2*(n - 1);      =       x = 2*(n - 1);
}                     =     } else {
                      =     }
```

Every conditional has two branches ("then" and "else")

- missing "`else`" still has an empty else branch

Branch Coverage

- Next metric is "branch coverage"
  - for what percentage of the conditionals, are both branches executed by some test

- Must test all branches reachable on allowed inputs
  - can ignore branches that are unreachable

    i.e., the ones that `throw new Exception` on bad inputs

Branch Coverage

- Consider the following function:

```
/** Returns the smaller of a and b. */
public static int min(int a, int b) {
  int m = a;
  if (a <= b)
    m = a;
  return a;
};
```

  - problem only arises when "`if`" falls through to code after
  - if every branch ends with `return` / `throw`,
    then statement coverage = branch coverage

    always true for code without mutation of local variables

Branch Coverage

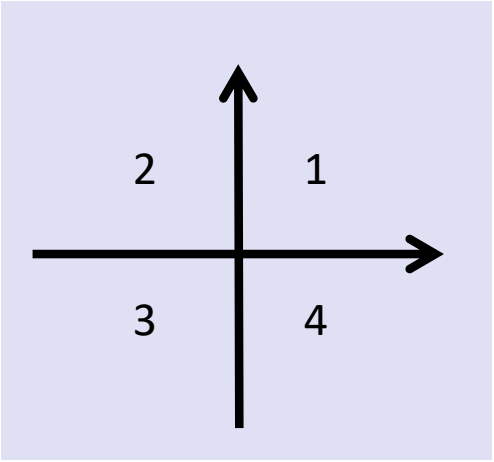
- Next metric is "branch coverage"
  - for what percentage of the conditionals, are both branches executed by some test
- Must test all branches reachable on allowed inputs
  - can ignore branches that are unreachable

    i.e., the ones that `throw new Error` on bad inputs
- Are we done?

Branch Coverage

- Consider the following function:

```
// Returns quadrant containing (x, y).
public static int quad(float x, float y) {
  int answer;
  if (x >= 0) {
    answer = 4;
  } else {
    answer = 3;
  }
  if (y >= 0)
    answer = 1;
  return answer;
};
```



  - testing on $(1, 1)$ and $(-1, -1)$ gives full branch coverage
  - this code is wrong... it never returns 2!

How Many Tests Are Required?

- More advanced metrics could fix this
  - "path coverage" would require 4 tests
  - #paths can grow exponentially in #branches
- For straight-line code and conditionals, we will only require branch coverage
- What about loops / recursion?

How Many Tests Are Required?

- Consider the following function:

```
public static int binarySearch(String s, String[] A) {
  int lo = 0;
  int hi = A.length;
  while (lo < hi) {  // s could be in A[lo .. hi-1]
    int m = (lo + hi) / 2;
    if (s < A[m]) {
      hi = m - 1;
    } else if (s > A[m]) {
      lo = m + 1;
    } else {
      return m;
    }
  }
  return hi;
};
```


Testing on s="a"/"b"/"c" A=["b"] gives full statement coverage

But the code is wrong.

In general, values written inside the loop are not read until the next time around, so you need 2+ iterations to test them.

How Many Tests Are Required?

- Our last metric is "loop coverage" (not 100% standardized)
  - what percent of loops are executed 0, 1, and many (2+) times by some test case

- Same idea applies to recursion
  - some arguments passed to recursive calls may not be read until the second recursive call
  - full loop coverage means every recursive call is executed 0, 1, and many times by some test

    will need this for **specifications** written in our math notation

Summary of testing requirements

- At least two tests for any function (non-UI)

- Must have full coverage of *reachable*
  - statements: must be executed
  - branches: must execute both branches
  - loops / recursion: must run 0, 1, & many times

Example 12

$$f: (\mathbb{N}) \rightarrow \mathbb{R}$$

$$f(0) := 0$$

$$f(n+1) := \sin\left(\left(n + \tfrac{1}{2}\right)\pi\right)$$

How many tests? Which ones?

- 0 (top branch) and 1 (bottom branch)

  statement coverage = branch coverage in functional code

Example 13

```
// n must be a non-negative integer
public static int f(int n) {
  if (n < 3) {
    return 0;
  } else if (n < 10) {
    return (n - 3) / 10;
  } else {
    return 1;
  }
}
```

How many tests? Which ones?

- 2 (top), 6 (middle), and 10 (bottom)

Example 14

```
// m and n must be a non-negative
public static int f(int m, int n) {
  if (m > n)
    m = n;
  return Math.abs(m);
}
```

How many tests? Which ones?

- $m=2, n=1$ gives full statement coverage
- adding $m=1, n=2$ gives branch coverage

Example 15

```
// n must be a non-negative integer
public static int f(int n) {
  if (n <= 1) {
    return 0;
  } else {
    return 1 + f(n / 2);
  }
}
```

How many tests? Which ones?

- 1 (0 recursive calls)
- 2 (1 recursive call)
- 5 (2 recursive calls)

Example 16

$f: (\mathbb{Z}) \to \mathbb{N}$

$$
\begin{aligned}
f(1) &:= 0 \\
f(n+1) &:= 1 + 2\, f(n) \qquad \textbf{if } 1 \le n \le 9
\end{aligned}
$$

- only defined on 1, 2, ..., 10

  we have $1 \le n \le 9$ exactly when $2 \le n+1 \le 10$

How many tests? Which ones?

- only 10 inputs, so... all of them

Other Heuristics

Not mandatory for 331 but useful in practice:

- Make sure every argument value is changed
- Look at special values
  - null, undefined, NaN, empty array, etc. often have bugs
- Look at the specification for branches
  - maybe the code doesn't split inputs where it should!
  - e.g., spec splits into "$\text{if } x \geq 0$" but code is "`if (x > 0)`"

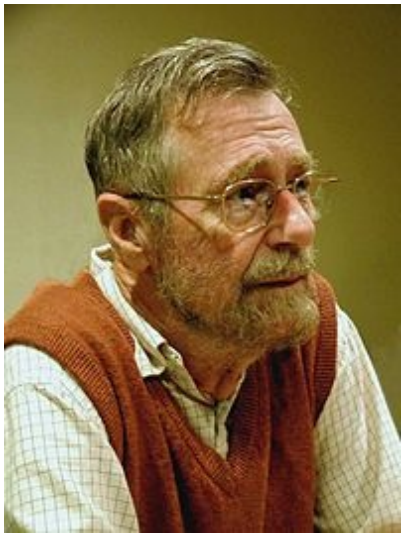
Summary of testing requirements

- At least two tests for any function (non-UI)
- Must have full coverage of *reachable*
  - statements: must be executed
  - branches: must execute both branches
  - loops / recursion: must run 0, 1, & many times
- Are we done?
  - no!

What Can We Learn From Testing?

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsgar Dijkstra
Notes on Structured Programming, 1970



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

Donald Knuth, 1977