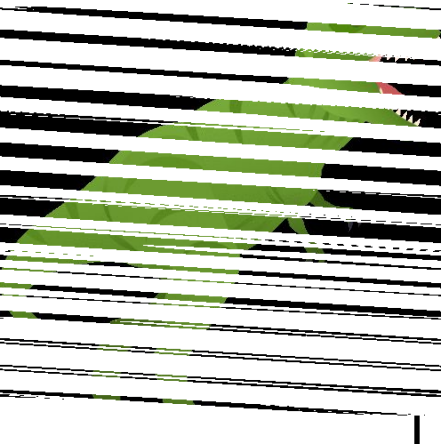
CSE 331
Software Design & Implementation

Autumn 2025

Section 1 – Specifications

Welcome!

- Let’s all introduce ourselves:
  - Name and pronouns
  - Year
  - What other classes you are taking this quarter
  - Would you rather be a dinosaur or a unicorn and why?



Homework Cycle

- Tuesdays: Section
  - content review & homework problems
  - Graded on participation
  - (If sick): Can submit on gradescope **@ 6pm Wednesday**
- Wednesdays - Fridays: At-Home Work
  - Submit remaining homework problems **@ 6pm Friday**
  - TAs will assess and give feedback
  - Graded on participation
- Mondays: In-Class Work
  - Complete problems during lecture time
  - **Graded submissions**

Review - Abstraction

- Hides unnecessary details from clients via **specification**
- Creates an *Abstraction Barrier* between clients and implementers
  - Implementers promise that the code follows the specification
  - Clients promise not to rely on details outside the specification

Poll - Abstraction

What is one downside of AI regarding abstraction?

AI does not respect abstraction barriers. It relies on implementation details when writing code rather than solely specifications.

This leads to dependent code, where a change in implementation of a function can break the rest of the code.

Review – Specification Types

- **Imperative** specification says how to calculate the answer
  - Gives the exact steps to get the answer
  - Just have to translate math to code
  - Ex: Absolute value: $|x| = x$ if $x \geq 0$ and $-x$ otherwise

- **Declarative** specification says what the answer looks like
  - Does not say how to calculate it
  - Up to us to ensure that our code satisfies the spec
  - Ex: Subtraction $(a - b)$: return x such that $b + x = a$

Question - Specification Types

What type is each of these specs?:

- Steps to calculate the square of a number n: multiply n by itself, n * n.
  - Declarative
  - Imperative

- Return a number, such that this number is the same value as the square of n
  - Declarative
  - Imperative

Question - Specification Types

What type is each of these specs?:

- Steps to calculate the square of a number n: multiply n by itself, n * n.
  - Declarative
  - **Imperative**

- Return a number, such that this number is the same value as the square of n
  - Declarative
  - Imperative

Question - Specification Types

What type is each of these specs?:

- Steps to calculate the square of a number n: multiply n by itself, n * n.
  - Declarative
  - **Imperative**

- Return a number, such that this number is the same value as the square of n
  - **Declarative**
  - Imperative

Review – Specifications

A specification consists of two parts:

Precondition: Allowed inputs

Postcondition: Allowed outputs

A specification is **stronger** when it has *less restrictive inputs* and *more restrictive outputs*. In other words, strong specifications have more guarantees for a larger set of inputs.

Question - Specifications

Which of the two specs is stronger?:

```
/** Returns the square root of a given number x
 * @requires x >= 0
 * @return the integer y such that y^2 = x
```

Or

```
/** Returns the square root of a given number x
 * @requires x >= 0
 * @return the integer y such that y^2 = x and y >= 0
```

Question - Specifications

Which of the two specs is stronger?:

```
/** Returns the square root of a given number x
 * @requires x >= 0
 * @return the integer y such that y^2 = x
```

Or

```
/** Returns the square root of a given number x
 * @requires x >= 0
 * @return the integer y such that y^2 = x and y >= 0
```

Review – Math Notation

Standard notations

Made up for this class

$\mathbb{N}$	all non-negative integers ("natural" numbers)
$\mathbb{Z}$	all integers
$\mathbb{R}$	all real numbers
$\mathbb{B}$	the boolean values (T and F)
$\mathbb{S}$	any character
$\mathbb{S}^*$	any sequence of characters ("strings")

- **Union**: $A \cup B$ set including everything in A *and* B
- **Tuple**: $A \times B$ all pairs (a, b) where a $\in A$ and b $\in B$
- **Record**: $\{x: A, y: B\}$ all records with fields x, y of types A, B

Review – Math Notation

- **Side Conditions**: limiting / specifying input in right column
  - ex: abs : $\mathbb{R} \rightarrow \mathbb{R}$

    abs(x) := x if $x \geq 0$

    abs(x) := –x if $x < 0$
  - conditions must be **exclusive** and **exhaustive**
- **Pattern Matching**: defining function based on input cases
  - Exactly **one** rule for every valid input

    ex: $f : \mathbb{N} \rightarrow \mathbb{N}$

    $f(0) := 0$

    $f(n+1) := n$
  - "n + 1" signifies that input must be > 0 since smallest $\mathbb{N}$ would be 0
  - Preferred over side conditions in most cases
- Course Website > Topics > Math Notation Notes

Review - Math Notation Example

Consider the following function, which calculates half when given an **even** number but also accepts other inputs (though it doesn't perform the same behavior in those cases):

$$\text{half} : (\text{undefined} \cup \mathbb{N}) \to \mathbb{Z}$$

$$\begin{aligned}
\text{half}(\text{undefined}) &:= 0 \\
\text{half}(n) &:= n/2 && \text{if } n \text{ is even} \\
\text{half}(n) &:= -(n+1)/2 && \text{if } n \text{ is odd}
\end{aligned}$$

Task 1a

Fill in the following table explaining the relationships between each pair of specifications. Write an "S" if the spec on left (the row) is stronger than the spec on top (the column), a "W" if the left spec is weaker, and "—" if the specs are incomparable.

	A	B	C	D	E
A	X				
B		X			
C			X		
D				X	
E					X

Task 1a

Fill in the following table explaining the relationships between each pair of specifications. Write an "S" if the spec on left (the row) is stronger than the spec on top (the column), a "W" if the left spec is weaker, and "—" if the specs are incomparable.

	A	B	C	D	E
A	X	W	—	—	W
B					
C					
D					
E					

Task 1a

Fill in the following table explaining the relationships between each pair of specifications. Write an "S" if the spec on left (the row) is stronger than the spec on top (the column), a "W" if the left spec is weaker, and "—" if the specs are incomparable.

	A	B	C	D	E
A	X	W	—	—	W
B	S	X	—	—	—
C					
D					
E					

Task 1a

Fill in the following table explaining the relationships between each pair of specifications. Write an "S" if the spec on left (the row) is stronger than the spec on top (the column), a "W" if the left spec is weaker, and "—" if the specs are incomparable.

	A	B	C	D	E
A	X	W	—	—	W
B	S	X	—	—	—
C	—	—	X	W	—
D					
E					

Task 1a

Fill in the following table explaining the relationships between each pair of specifications. Write an "S" if the spec on left (the row) is stronger than the spec on top (the column), a "W" if the left spec is weaker, and "—" if the specs are incomparable.

	A	B	C	D	E
A	X	W	—	—	W
B	S	X	—	—	—
C	—	—	X	W	—
D	—	—	S	X	—
E					

Task 1a

Fill in the following table explaining the relationships between each pair of specifications. Write an "S" if the spec on left (the row) is stronger than the spec on top (the column), a "W" if the left spec is weaker, and "—" if the specs are incomparable.

	A	B	C	D	E
A	X	W	—	—	W
B	S	X	—	—	—
C	—	—	X	W	—
D	—	—	S	X	—
E	S	—	—	—	X

Task 1b

Now consider a new pair of precondition and postcondition behaviors:

```
@requires price > 0
@return an item T in items with T.price <= price <= T.price + 0.5
```

This specification is not sensible, what is wrong with it? Why shouldn't we use it?

- The specification does not describe what should happen if/when items is null.
- This is a problem because a client cannot effectively use a method if they do not know what happens in some cases.

Task 2a

For each of the following implementations, state which of the specifications it satisfies. If it does not satisfy some specification, explain (in as few words as possible) why it does not.

```
public static MenuItem findBest(int price, MenuItem[] items) {
  if (price <= 0)
    throw new IllegalArgumentException("bad price");
  for (int i = 0; i < items.length; i++) {
    if (items[i].price <= price && price <= items[i].price + 0.5)
      return items[i];
  }
  return null;
}
```

A & B

C & D are not satisfied since the item returned may not be the one nearest below in price.

E is not satisfied since a NullPointerException is thrown instead of returning null when items is null.

Task 2b

For each of the following implementations, state which of the specifications it satisfies. If it does not satisfy some specification, explain (in as few words as possible) why it does not.

```
public static MenuItem findBest(int price, MenuItem[] items) {
  items.sort();  // puts items in order by increasing price
  MenuItem best = null;
  for (int i = 0; i < items.length; i++) {
    if (items[i].price <= price)
      best = items[i];
  }
  return best;
}
```

C & D

A & B & E are not valid since the return may not be within the required price range

Task 2c

For each of the following implementations, state which of the specifications it satisfies. If it does not satisfy some specification, explain (in as few words as possible) why it does not.

```
public static MenuItem findBest(int price, MenuItem[] items) {
  if (items == null)
    return null;
  if (price <= 0)
    throw new IllegalArgumentException("bad price");
  for (int i = 0; i < items.length; i++) {
    if (items[i].price <= price && price <= items[i].price + 0.5)
      return items[i];
  }
  return null;
}
```

A & B & E

C & D are not satisfied since the item returned may not be the one nearest below in price.

Review - Testing

- **Statement Coverage**
  - Test every executable statement reachable by an allowed input
- **Branch Coverage**
  - For every conditional, test all branches for allowed inputs
- **Loop Coverage**
  - Every loop/recursive call must be tested on 0, 1, any 2+ iterations for allowed inputs
- **Exhaustive Testing**
  - Test all possible inputs for functions with <= 10 allowed inputs

Notes on Testing Requirements

Task 3

For each of the following functions, state the number of tests required to meet our coverage requirements and explain why that is the required number.

Then, describe a specific set of tests to use (with the same number of tests you as said before). Describe each test by giving the input (identify a specific input rather than saying, e.g., "some positive number"), saying what portion of the function it tests, and explaining why our rules require that test.

a)

```
public static int f(int n) {
  if (n < 0) {
    return -2 * n;
  } else {
    return 3 * n;
  }
```

2 test cases (n < 0 and n >= 0)

- Achieves statement coverage
- Statement coverage also yields branch coverage
- Loop coverage vacuously true
- Ex: f(-1) = 2, f(2) = 6

b)

```
public static int h(int n) {
  if (n <= 0) {
    return 1;
  } else {
    return 2 + h(n / 3);
  }
}
```

3 test cases (0, 1, 2+ iterations)

- Achieves loop coverage
- Also yields Statement & branch coverage
- Ex: h(0) = 1, h(1) = 3, h(3) = 5