

Procedural Abstraction and Specification

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Homework 0 and Reading 1: due tonight, 11:59pm
- Homework 1: due Friday, October 9, 11:59pm
- First section: Tuesday, October 6

Last time

- **Specification:** what code must do on every run
 - As seen from outside it
- **Assertion:** a claim about the program's state at one point
 - It **holds** if it is true there on every run
- **Reasoning:** from P , through the code, to Q

Sum, in pieces

One method reads the numbers:

```
13  static List<Integer> readList() {  
14      Scanner in = new Scanner(System.in);  
15      List<Integer> list = new ArrayList<>();  
16      while (in.hasNextInt()) {  
17          list.add(in.nextInt());  
18      }  
19      return list;  
20  }
```

Sum, in pieces

And another adds them up:

```
22  static int sumList(List<Integer> list) {  
23      int sum = 0;  
24      for (int i = 0; i < list.size(); i++) {  
25          sum += list.get(i);  
26      }  
27      return sum;  
28  }
```

Sum, in pieces

Then `main` calls them both:

```
6  public static void main(String[] args) {  
7      System.out.println("Enter some numbers.");  
8      List<Integer> list = readList();  
9      System.out.println("Got " + list);  
10     System.out.println("The sum: " + sumList(list));  
11 }
```

- Same output as last lecture's `Sum`, on every input
 - So it meets the same specification

Does main work?

```
7  System.out.println("Enter some numbers.");
8  List<Integer> list = readList();
9  System.out.println("Got " + list);
10 System.out.println("The sum: " + sumList(list));
```

- To argue that `Sum` meets its specification, we need:
 - what `readList` returns
 - what `sumList` returns for that list
- `main` does not say

Read every method?

- We could read both, as we read the loops last lecture
- Not once a program has a thousand methods
- And `main` may rely on how `sumList` happens to be written
 - Then a rewrite of `sumList` can break `main`
- **Instead:** write down each method's **promises**
 - What the code that calls it can count on

A specification for a method

- A method's runs are its calls
- So its specification says what every call must do
 - As seen from the code that makes the call
- **Client:** code that calls a method
 - For example, `main` is a client of `sumList`

Client and user

- A program's **user** owes it nothing
 - They may type anything at all
 - So the user manual covers every input
- A method's **client** can be held to a promise
 - It is code, by a programmer who can read the specification
- So a method's specification has two parts
 - What it asks of its client, and what it promises in return

Method specification

- **Precondition:** an assertion where the method is called
 - The client must make it true
- **Postcondition:** an assertion where it returns or throws
 - The implementer must make it true
 - The implementer may assume the precondition

Meeting a specification

A method **meets** its specification if every call that begins with the precondition true ends with the postcondition true.

- A call that breaks the precondition: no promise
 - Any value, any exception, or never returning
 - The method still meets its specification

Homework 1 says *satisfies* where we say *meets*; the two mean the same.

Assertions become promises

- The claim at (1): what `readList` promises
 - About the list it returns
- The claim at (2) about `sum`: what `sumList` promises
 - About the number it returns
- Preconditions: short
 - `readList` asks nothing of its client
 - `sumList` asks only for a list that is not `null`

A specification in Java

```
/**
 * Returns the element in the middle of a list.
 *
 * @param list the list to look in; must be non-null
 * @requires list has an odd number of elements
 * @return the element with as many elements before it as after it
 */
static int middle(List<Integer> list) {
    return list.get(list.size() / 2);
}
```

Writing it in Java

- **Where:** just above the method, in a comment starting `/**`
- **First:** a sentence saying what the method does
- **Then:** each part of the specification, behind a **tag**
 - A word starting with `@`
- **javadoc:** turns these comments into web pages
 - Like the pages for `ArrayList` and `Scanner`

The six tags

Tag	Says	Part of the
@param	what a parameter is for	precondition
@requires	a precondition too long for one @param	precondition
@return	what the method returns	postcondition
@throws	which exception it throws, and when	postcondition
@modifies	the parameters the method may change	postcondition
@effects	how those parameters change	postcondition

@requires, @modifies, and @effects are this course's, from Liskov and Guttag.

Two specifications

First, `readList`:

```
/**
 * Reads whole numbers typed by the user.
 *
 * @return a list holding, in order, every number the user typed
 *         before the input ended or something that was not an int appeared
 */
static List<Integer> readList()
```

Two specifications

Then `sumList`:

```
/**
 * Adds up a list of numbers.
 *
 * @param list the numbers to add; must be non-null
 * @return the sum of the elements of list, wrapped into the int range
 *         if the true sum falls outside it
 */
static int sumList(List<Integer> list)
```

What they leave out

- **Scanner**, loops, indices: a client does not need them
- **@modifies**: `sumList` has none
 - So it promises not to change `list`
 - The second claim last lecture needed at (2)
 - The list the user was shown is the list that got summed

When a method changes something

```
/**
 * Reads whole numbers typed by the user into a list.
 *
 * @param list the list to add to; must be non-null
 * @modifies list
 * @effects adds to the end of list, in order, every number the user
 *          typed before the input ended or something that was not an int
 *          appeared
 */
static void readInto(List<Integer> list)
```

The parameters it may change: here, `list`

How `list` changes: the new list in terms of the old

Reasoning across a call

```
7  System.out.println("Enter some numbers.");
8  List<Integer> list = readList();
   {{ list holds, in order, every number the user typed before
      the input ended or something that was not an int appeared }}
9  System.out.println("Got " + list);
10 System.out.println("The sum: " + sumList(list));
```

`readList`: no precondition, and a promise about the list it returns

Line 8 stores that list in `list`, so the claim holds

A call as the code

```
{{ P }}  
a call  
{{ Q }}
```

- **P**: the method's precondition
- **The code**: the call
 - Takes any **P** state to a **Q** state, as the specification promises
- **Q**: the method's postcondition

The call on line 10

- `readList` promised a list, so `list` is not `null`
 - As the precondition of `sumList` requires
- The call returns the wrapped sum of the elements of `list`
- `sumList` promises not to change `list`
 - So its elements are still the numbers line 9 printed
- So line 10 prints what `Sum`'s specification says
- Each step relied on nothing but the promises

Specification or source code?

Half sheet, question 2

Procedural abstraction

Procedural abstraction is using a method through its specification alone.

- **Clients:** rely on nothing it does not promise
- **Implementer:** may change anything it does not promise
- **Abstraction barrier:** the specification, as the line between them

The abstraction barrier

- On one side, `main` relies on the promises and nothing else
- On the other, `sumList` may add up the list from the end
 - Same promises: `main` needs no change and no new argument
- A large program is argued correct one method at a time
 - Each against the specifications of the methods it calls

Choosing a specification

- A method can often have more than one reasonable specification

```
/**
 * Searches for a value in part of an array.
 *
 * @param a the array to search; must be non-null
 * @param lo the index where the range to search begins
 * @param hi the index just past where the range to search ends
 * @param x the value to search for
 * ... and one of the specifications below ...
 */
static int indexOf(int[] a, int lo, int hi, int x)
```

Five specifications

Spec	Requires	Returns	If there is no match
A	a match	a match	no promise
B		the smallest match	-1
C		the largest match	-1
D		some match	-1
E		the smallest match	NoSuchElementException

- A **match**: an index i with $lo \leq i < hi$ and $a[i] == x$
- All five also require $0 \leq lo \leq hi \leq a.length$

Four implementations

```
// Implementation 1
for (int i = lo; i < hi; i++) {
    if (a[i] == x) {
        return i;
    }
}
return -1;
```

```
// Implementation 2
for (int i = lo; i < hi; i++) {
    if (a[i] == x) {
        return i;
    }
}
throw new NoSuchElementException();
```

```
// Implementation 3
int i = lo;
while (a[i] != x) {
    i++;
}
return i;
```

```
// Implementation 4
for (int i = hi - 1; i >= lo; i--) {
    if (a[i] == x) {
        return i;
    }
}
return -1;
```

Which meet which?

Half sheet, question 1

The answers

	Spec A	Spec B	Spec C	Spec D	Spec E
Implementation 1	yes	yes	no	yes	no
Implementation 2	yes	no	no	no	yes
Implementation 3	yes	no	no	no	no
Implementation 4	yes	no	yes	yes	no

Implementation 1: the first match, or -1

Implementation 2: the first match, or an exception

Implementation 3: never looks at hi

Implementation 4: the last match, or -1

Read it by specification

	Spec A	Spec B	Spec C	Spec D	Spec E
Implementation 1	yes	yes	no	yes	no
Implementation 2	yes	no	no	no	yes
Implementation 3	yes	no	no	no	no
Implementation 4	yes	no	yes	yes	no
Met by	4	1	1	2	1

- A specification that promises more is met by fewer implementations

Stronger and weaker

- **Stronger**, for assertions: P is stronger than Q
 - If Q is true whenever P is
- **Stronger**, for specifications: $S1$ is stronger than $S2$
 - If every implementation that meets $S1$ also meets $S2$
- **Weaker**: the reverse; Q is weaker than P , $S2$ than $S1$
- **Incomparable**: neither stronger nor weaker than the other

Same precondition

B, C, D, and E share one, so we compare only their postconditions:

- **B and C are each stronger than D**
 - The smallest or largest match is in particular some match
- **B and C are incomparable**
 - On an array with x twice, the smallest and largest differ
- **E is incomparable with B, C, and D**
 - Where x is absent, they promise -1 and E an exception

A asks more

- Where x occurs, A and D promise the same: a match
- Where x does not, a call breaks A 's precondition
 - A promises nothing about that call
- An implementation that meets D handles every call
 - So it meets A too: D is stronger than A

The rule

S1 is stronger than S2 when
S1's precondition is weaker than or the same as S2's,
and, on every call where S2's precondition is true,
S1's postcondition is stronger than or the same as S2's.

- So each of B, C, D, and E is stronger than A

All five compared

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

The row against the column: **S** stronger, **W** weaker, --- incomparable.

Which is better?

- **Stronger:** promises the client more
 - B obliges `indexOf` to find the first match
- **Weaker:** leaves the implementer more choices
 - D lets `indexOf` search from either end
 - Each choice is a case the client must handle

A good specification rules out exactly the implementations its clients cannot use.

Changing a specification

Change	Cannot break	Can break
Strengthening	a client	the implementation
Weakening	the implementation	a client

- **Strengthening** keeps every promise a client relied on
- **Weakening** makes no promise the old specification did not
- With only your own code as clients, fix both sides at once
- With a thousand clients, a specification can only grow stronger

Writing a client

A client is **correct** if it works with every implementation that meets the specifications it relies on.

- Working with the implementation it happens to run with is not enough

Is count correct?

```
static int count(int[] a, int x) {  
    int c = 0;  
    int i = indexOf(a, 0, a.length, x);  
    while (i != -1) {  
        c++;  
        i = indexOf(a, i + 1, a.length, x);  
    }  
    return c;  
}
```

The first search starts at 0; each later one, just past the last match

Against B: correct, since no search skips a match

Against D: a search may skip matches, so count is wrong

count on [5, 1, 5, 2, 5]

Implementations 1 and 4 both meet D.

```
// With implementation 1
indexOf(a, 0, 5, 5) // 0
indexOf(a, 1, 5, 5) // 2
indexOf(a, 3, 5, 5) // 4
indexOf(a, 5, 5, 5) // -1
count(a, 5)         // 3
```

```
// With implementation 4
indexOf(a, 0, 5, 5) // 4
indexOf(a, 5, 5, 5) // -1
count(a, 5)         // 1
```

Against D, count is wrong, though it may pass every test you give it

Does x occur?

```
indexOf(a, 0, a.length, x) != -1
```

- **B, C, and D** promise -1 exactly when x is absent: correct
- **A**: when x is absent, the call breaks A's precondition
- **E**: the same call throws where the client expects -1

Promises about parameters

- So far: promises about what `indexOf` returns or throws
- A method can also promise things about its parameters
 - `@effects`: how the parameters in `@modifies` change
 - `@modifies`: any parameter it does not list stays unchanged
- Both are part of the postcondition, so the rule covers them

Comparing @effects

Two versions of `sort`'s `@effects`, both with `@modifies list`:

Original	<code>@effects</code> rearranges the elements of <code>list</code> into ascending order
New	<code>@effects</code> after the call, <code>list</code> is in ascending order

- Same precondition and `@modifies`: compare only `@effects`
- **The original is stronger than the new one**
 - Rearranged into ascending order is in particular in ascending order

The original `@effects` follows Liskov and Guttag's `sort`.

Does the new @effects promise as much?

Try an implementation that throws the elements away:

```
list.clear();
```

- On `[3, 1, 2]`, it leaves `[]`; the original asks for `[1, 2, 3]`
- **It meets the new one!** Every call leaves `[]`, in ascending order
- **It fails the original** on every list that is not empty
- **So the new one promises less than the original**
 - The original relates the list after the call to the list before

Comparing @modifies

`sumList` has no `@modifies`, so it promises not to change `list`.

Suppose its specification added two lines:

```
@modifies list  
@effects  rearranges the elements of list into some order
```

- Same precondition and `@return`: compare promises about `list`
- **The original is stronger than the new one**
 - Unchanged is in particular rearranged into some order: the old one

Does the new `@modifies` promise as much?

Try an implementation that sorts `list` and then adds it up.

- The order of the numbers does not change their sum: the right sum
- Sorting rearranges `list`, so it meets the new specification
- On `[3, 1, 2]`, it leaves `[1, 2, 3]`, so it fails the original
- So the new one promises less than the original
 - Listing `list` in `@modifies` drops the promise not to change it

Knowing a program in pieces

- **One method at a time:** each meets its own specification
 - The specifications of what it calls stand in for their code
- **Each client:** works with every implementation the specifications allow
- **What procedural abstraction is for:**
 - No one has to hold the whole program in their head
 - Clients stay as they are when a rewrite keeps the promises

Coming up

- Monday: Reading 1 discussion, and batch processing
- Tuesday: section, on assertions and specifications
- Homework 1, due Friday, October 9:
 - Tasks 1 and 2: matching and comparing specifications, as today
 - Task 4: a client that works with any correct implementation
- Hand in your half sheet at the door