

---

# CSE 331

# Software Design & Implementation

Hal Perkins  
Winter 2021  
Specifications

# Administrivia 1

---

- Next two assignments out today
  - HW2: Written problems on loops, due Tue. night 11 pm
  - HW3: Java warmup & project logistics (posted tonight)
    - Due next Thur. night, 11 pm
    - You should get gitlab email later today when repo created. Feel free to ignore until section tomorrow.
    - Shouldn't take much time, but start early so we can chase down any configuration problems
      - & read and follow instructions carefully!
    - Warning: Stackoverflow and Google are probably *not* your friends for configuration problems. Our setup is intended to work, not to require random web searches and tinkering. If something isn't working right, check handouts, use office hours, and/or post on the discussion board.

# Administrivia 2

---

- Sections tomorrow on hw3 & project logistics
  - If you plan to use your own laptop, bring it
    - Install OpenJDK Java 11 JDK (not JRE), IntelliJ Ultimate, and git before section
      - See the “Project Software Setup” handout for details
      - (Java 12, ... also ok, but we use Java 11 for testing)
    - Windows users: Best practice: remove all existing Java JDKs and JREs before installing current one
    - Everyone: be sure you have a clean IntelliJ with no strange plugins, customized options, etc.
      - Students can get a free Ultimate version license

# New Administrivia (added Friday)

---

- HW2 due Tuesday night **11PM** (not 11:30, not 11:59)
- HW3 due Thursday night, 11 PM
  - But please be sure you've got things set up and have been able to clone your gitlab repo way before then so we have time to fix problems. Email `cse331-staff[at]cs` if individual setup bugs
- UW closed Monday (MLK day) – no class
  - But we will have regular office hours (thanks to our great TAs!)

# 2 Goals of Software System Building

---

- Building the *right system*
  - Does the program meet the user's needs?
  - Determining this is usually called *validation*
- Building the *system right*
  - Does the program meet the specification?
  - Determining this is usually called *verification*
- CSE 331: the second goal is the focus – creating a correctly functioning artifact
  - Surprisingly hard to specify, design, implement, test, and debug even simple programs

# Where we are

---

- We've started to see how to reason about code
- We'll build on those skills in many places:
  - *Specification*: What are we supposed to build?
  - *Design*: How do we decompose the job into manageable pieces? Which designs are “better”?
  - *Implementation*: Building code that meets the specification
  - *Testing*: Systematically finding problems
  - *Debugging*: Systematically fixing problems
  - *Maintenance*: How does the artifact adapt over time?
  - *Documentation*: What do we need to know to do these things? How/where do we write that down?

# The challenge of scaling software

---

- Small programs are simple and malleable
  - Easy to write
  - Easy to change
- Big programs are (often) complex and inflexible
  - Hard to write
  - Hard to change
- Why does this happen?
  - Because *interactions* become unmanageable
- How do we keep things simple and malleable?
  - Divide and conquer!

# A discipline of modularity

---

- Two ways to view a program:
  - The client's view (how to use it)
  - The implementer's view (how to build it)
- Apply implementer and client views to system parts:
  - While implementing one part, consider yourself a client of any other parts it depends on
  - Ignore the implementation of those other parts
  - Minimizes interactions between parts
- Formalized through the idea of a *specification*

# A specification is a contract

---



- A set of requirements agreed to by the user and the manufacturer of the product
  - Describes their expectations of each other
- Facilitates simplicity via *two-way* isolation
  - Isolate client from implementation details
  - Isolate implementer from how the part is used
  - Discourages implicit, unwritten expectations
- Facilitates change
  - reduces the “Medusa effect”: the specification, rather than the code, gets “turned to stone” by client dependencies



# Importance of Specifications

---

Specifications are essential to **correctness**

They are also essential to **changeability**

- need to know what changes will break code using it

They are also essential to **understandability**

- need to tell readers what it is supposed to do

They are also essential to **modularity**...

# Method Specifications

---

To prove correctness of a method, we need

- precondition
- postcondition

Correctness =  
Validity of  
 $\{ P \} S \{ Q \}$

Without these, we can't say whether the code is correct

These tell us what it means to be correct

They are the *specification* for the method

# Isn't the interface sufficient?

---

An interface defines the boundary between implementers and users:

```
public class BadList<E> implements List<E> {
    public E get(int x) { return null; }
    public void set(int x, E y){}
    public void add(E x) {}
    public void add(int x, E y){}
    ...
    public static <T> boolean isSub(List<T>, List<T>){
        return false;
    }
}
```

Interface provides the *syntax and types*

But nothing about the *behavior and effects*

- *Provides too little information to clients*

*Note: Code above is right concept but might not be (completely) legal Java*

- slides will often gloss over details to get main ideas to fit

# Why not just read code?

---

```
static <T> boolean sub(List<T> src, List<T> part) {
    int part_index = 0;
    for (T elt : src) {
        if (elt.equals(part.get(part_index))) {
            part_index++;
            if (part_index == part.size()) {
                return true;
            }
        } else {
            part_index = 0;
        }
    }
    return false;
}
```

Why are you better off with a specification?

# Code is complicated

---

- Code gives more detail than needed by client
- Understanding or even reading every line of code is an excessive burden
  - Suppose you had to read source code of Java libraries to use them
  - Same applies to developers of different parts of the libraries
- Client cares only about *what* the code does, not *how* it does it

# Code is ambiguous

---

- Code seems unambiguous and concrete
  - But which details of code's behavior are **essential**, and which are **incidental**?
- Code invariably gets rewritten
  - Clients need to know what they can rely on
    - What properties will be maintained over time?
    - What properties might be changed by future optimization, improved algorithms, or bug fixes?
  - Implementer needs to know what features the client depends on, and which can be changed

# Comments are essential

---

Typical comments often convey only an informal, general idea of what that the code does:

```
// This method checks if "part" appears as a
// sub-sequence in "src"
static <T> boolean sub(List<T> src, List<T> part) {
    ...
}
```

Problem: ambiguity remains

- What if `src` and `part` are both empty lists?
- When does the function return `true`?

# From vague comments to specifications

---

- Roles of a specification:
  - Client agrees to rely *only* on information in the description in their use of the part
  - Implementer of the part promises to support everything in the description
    - Otherwise is perfectly at liberty
- Sadly, much code lacks a specification
  - Clients often work out what a method/class does in ambiguous cases by running it and depending on the results
  - Leads to bugs and programs with unclear dependencies, reducing simplicity and flexibility

# Recall the sublist example

---

```
static <T> boolean sub(List<T> src, List<T> part) {
    int part_index = 0;
    for (T elt : src) {
        if (elt.equals(part.get(part_index))) {
            part_index++;
            if (part_index == part.size()) {
                return true;
            }
        } else {
            part_index = 0;
        }
    }
    return false;
}
```

# A more careful description of `sub`

---

***// Check whether “part” appears as a sub-sequence in “src”***

needs to be given some caveats (why?):

*// \* src and part cannot be null*

*// \* If src is empty list, always returns false*

*// \* Results may be unexpected if partial matches*

*// can happen right before a real match; e.g.,*

*// list (1,2,1,3) will not be identified as a*

*// sub sequence of (1,2,1,2,1,3).*

or replaced with a more detailed description:

*// This method scans the “src” list from beginning*

*// to end, building up a match for “part”, and*

*// resetting that match every time that...*

# It's better to *simplify* than to *describe* complexity!

---

A complicated description suggests poor design

- Rewrite `sub` to be more sensible, and easier to describe

*// returns true iff possibly empty sequences A, B exist such that*

*// src = A + part + B*

*// where “+” is sequence concatenation*

```
static <T> boolean sub(List<T> src, List<T> part) {
```

- Mathematical flavor not always necessary, but often helps avoid ambiguity
- “Declarative” style is important
  - Avoid reciting or depending on implementation details

# Sneaky fringe benefit of specs #1

---

- The discipline of writing specifications changes the **incentive structure** of coding
  - Rewards code that is easy to describe and understand
  - Punishes code that is hard to describe and understand
    - Even if it is shorter or easier to write
- If you find yourself writing complicated specifications, it is an incentive to redesign
  - In **sub**, code that does exactly the right thing may be slightly slower than a hack that assumes no partial matches before true matches, but cost of forcing client to understand the details is too high

# Writing specifications with Javadoc

---

- Javadoc
  - Sometimes can be daunting; get used to using it
- Javadoc convention for writing specifications
  - Method signature (prototype – name, parameters, result type)
  - Text description of method
  - **@param**: description of what gets passed in
  - **@return**: description of what gets returned
  - **@throws**: exceptions that may occur

# Example: Javadoc for `String.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

Throws:

`NullPointerException` – if `s` is null

Since:

1.5

# CSE 331 specifications

---

**Note:** slides are abbreviated. In your code, it must be `@spec.requires`, `@spec.modifies`, etc. but `@throws`, `@returns` for standard Javadoc tags

- The *precondition*: constraints that hold before the method is called (if not, all bets are off – no guarantees about what method will do)
  - **@requires**: spells out any obligations on client
- The *postcondition*: constraints that hold after the method is called (if the precondition held)
  - **@modifies**: lists objects that may be affected by method; any object not listed is guaranteed to be unchanged
  - **@effects**: gives guarantees on final state of modified objects
  - **@throws**: lists possible exceptions and conditions under which they are thrown (Javadoc uses this too)
  - **@returns**: describes return value (Javadoc uses this too)

# Example 1

---

static <T> int **change**(List<T> **lst**, T **oldelt**, T **newelt**)  
    **requires**     **lst**, **oldelt**, and **newelt** are non-null.  
                  **oldelt** occurs in **lst**.  
  
    **modifies**    **lst**  
  
    **effects**     change the first occurrence of **oldelt** in **lst** to **newelt**  
                  & makes no other changes to **lst**  
  
    **returns**    the position of the element in **lst** that was **oldelt** and  
                  is now **newelt**

---

```
static <T> int change(List<T> lst,  
                      T oldelt, T newelt) {  
    int i = 0;  
    for (T curr : lst) {  
        if (curr == oldelt) {  
            lst.set(newelt, i);  
            return i;  
        }  
        i = i + 1;  
    }  
    return -1;  
}
```

# Example 2

---

static List<Integer> zipSum(List<Integer> lst1, List<Integer> lst2)

requires lst1 and lst2 are non-null.  
lst1 and lst2 are the same size.

modifies none

effects none

returns a list of same size where the ith element is  
the sum of the ith elements of lst1 and lst2

---

```
static List<Integer> zipSum(List<Integer> lst1
                             List<Integer> lst2) {
    List<Integer> res = new ArrayList<Integer>();
    for(int i = 0; i < lst1.size(); i++) {
        res.add(lst1.get(i) + lst2.get(i));
    }
    return res;
}
```

# Example 3

---

static void `listAdd`(List<Integer> `lst1`, List<Integer> `lst2`)

`requires`    `lst1` and `lst2` are non-null.

`lst1` and `lst2` are the same size.

`modifies`   `lst1`

`effects`     `ith` element of `lst2` is added to the `ith` element of `lst1`

`returns`     none

---

```
static void listAdd(List<Integer> lst1,
                   List<Integer> lst2) {
    for(int i = 0; i < lst1.size(); i++) {
        lst1.set(i, lst1.get(i) + lst2.get(i));
    }
}
```

# A method should do only one thing

---

- Either return a value, or have some *side effect* (modification of program state)
  - Poor style to have *both* `@effects` and `@returns`
  - Exception: methods like `HashMap.put` that return an old value as part of an update

# Should requires clause be checked?

---

- If the client calls a method without meeting the precondition, the code is free to do *anything*
  - Including pass corrupted data back
  - It is polite, nevertheless, to *fail fast*: to provide an immediate error, rather than permitting mysterious/silent bad behavior
- Preconditions are common in “helper” methods/classes
  - In public libraries, it’s friendlier to deal with all possible input
  - *But: binary search would normally impose a pre-condition rather than simply failing if list is not sorted. Why?*
- Rule of thumb: Check if cheap to do so
  - *Example: list has to be non-null → check*
  - *Example: list has to be sorted → skip*
  - A quality implementation will check preconditions whenever it is inexpensive and convenient to do so
    - Defensive programming

# @throws vs @requires

---

- Require a precondition or throw an exception if it's cheap enough to check? Which is better?
  - @requires size > 0
  - @throws IllegalArgumentException if size <= 0
- Tradeoffs
  - @throws describes exactly what will happen – it is part of the spec.
  - @requires says “if this precondition isn't met, who knows what might happen?”
- **Must** choose one or the other – can't include both.
  - Can't specify “who knows what might happen” and “this is guaranteed to happen” for the same input!

# Sneaky fringe benefit of specs #2

---

- Specification means that client doesn't need to look at implementation
  - So the code might not even exist yet!
- Write specifications first, make sure system will fit together, and then assign separate implementers to different modules
  - Allows teamwork and parallel development
  - Also helps with testing (future topic)

# Upgrading a library

---

- Your program uses a library
- Can you use a different library?
- Can you use a new version?

We want an objective test

You can upgrade if the specification of the new version is **stronger**

- It makes at least as many promises and doesn't require more from your program
- Example:
  - Weaker spec: returns the elements
  - Stronger spec: returns the elements in sorted order

# Satisfaction of a specification

---

Let  $M$  be an implementation and  $S$  a specification

$M$  satisfies  $S$  if and only if

- for every input allowed by the spec precondition,  
M produces an output allowed by the spec postcondition

If  $M$  does not satisfy  $S$ , either  $M$  or  $S$  (or both!) could be “wrong”

- “*one person’s feature is another person’s bug.*”
- usually better to change the implementation than the spec

# Stronger vs Weaker Specifications

---

- **Definition 1:** specification  $S_2$  is stronger than  $S_1$  iff
  - for any implementation  $M$ :  $M$  satisfies  $S_2 \Rightarrow M$  satisfies  $S_1$
  - i.e.,  $S_2$  is harder to satisfy

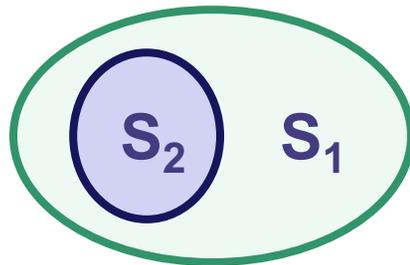


- Two specifications may be *incomparable*
  - but we are usually choosing between stronger vs weaker

# Stronger vs Weaker Specifications

---

- An implementation satisfying a stronger specification can be **used anywhere** that a weaker specification is required
  - can **use** a method satisfying  $S_2$  anywhere  $S_1$  is expected



Making changes to a specification...

- changing from  $S_1$  to  $S_2$  should not break clients
  - but it could break implementation
- changing from  $S_2$  to  $S_1$  should not break implementation
  - but it could break clients!

# Stronger vs Weaker Specifications

---

- **Definition 2:** specification  $S_2$  is stronger than  $S_1$  iff
  - postcondition of  $S_2$  is stronger than that of  $S_1$   
(on all inputs allowed by both)
  - precondition of  $S_2$  is weaker than that of  $S_1$
- A **stronger** specification:
  - is harder to satisfy
  - gives more guarantees to the caller
- A **weaker** specification:
  - is easier to satisfy
  - gives more freedom to the implementer

# Two specifications for find which is stronger?

---

```
int find(int[] a, int value) {
    for (int i=0; i<a.length; i++) {
        if (a[i]==value)
            return i;
    }
    return -1;
}
```

- Specification A
  - requires: value occurs in **a**
  - returns: **i** such that **a[i] = value**
- Specification B
  - requires: value occurs in **a**
  - returns: *smallest* **i** such that **a[i] = value**

# Two specifications for find

## Which is stronger?

---

```
int find(int[] a, int value) {
    for (int i=0; i<a.length; i++) {
        if (a[i]==value)
            return i;
    }
    return -1;
}
```

- Specification A
  - requires: value occurs in **a**
  - returns: **i** such that **a[i] = value**
- Specification C
  - returns: **i** such that **a[i] = value**, or **-1** if value is not in **a**

# Why compare specifications?

---

We wish to relate **procedures to specifications**

- Does the procedure satisfy the specification?
- Has the implementer succeeded?

We wish to compare **specifications to one another**

- Which specification (if either) is stronger?
- A procedure satisfying a stronger specification can be used anywhere that a weaker specification is required
  - Substitutability principle

Given two specifications, they may be *incomparable*

- Neither is weaker/stronger than the other
- *Some* implementations might still satisfy them both

# Substitutability

---

- Suppose that
  - $I_1$  and  $I_2$  satisfy specification  $S$
  - $P$  uses  $I_1$  as a component (and relies only on  $S$ )
- Then  $P$  can use  $I_2$
  
- Further, suppose that
  - $I_3$  satisfies  $S_3$  which is stronger than  $S$
- Then  $P$  can use  $I_3$
  
- Fact: If specification  $S_3$  is stronger than  $S_1$ , then for any implementation  $I$ ,  $I$  satisfies  $S_3 \Rightarrow I$  satisfies  $S_1$

# “Strange” case: @throws

---

[Prior versions of course, including old exams, were clumsy/wrong about this]

Compare:

S1:

@throws FooException if  $x < 0$

@return  $x + 3$

S2:

@return  $x + 3$

- These are *incomparable* because they promise different, incomparable things when  $x < 0$
- Both are *stronger* than @requires  $x \geq 0$ ; @return  $x + 3$

# Which is better?

---

- Stronger does not always mean better!
- Weaker does not always mean better!
- Strength of specification trades off:
  - Usefulness to client
  - Ease of simple, efficient, correct implementation
  - Promotion of reuse and modularity
  - Clarity of specification itself
- “It depends”

# Review:

## Ways to compare specifications

---

- A *stronger* specification is satisfied by fewer implementations
- A stronger specification has
  - *weaker* preconditions (note contravariance)
  - stronger postcondition
  - fewer modificationsCan be checked by hand
- A stronger specification has a (logically) stronger formula – can be checked by tools

# Specification style

---

- The point of a specification is to be helpful
  - Formalism helps, excessive formalism doesn't
- A specification should be
  - coherent: not too many cases
  - informative: (bad example in Java library, `HashMap.get`; what does result of `null` mean?)
  - strong enough: to do something useful, to provide guarantees
  - weak enough: to permit (efficient) implementation

# Warnings on Specifications

---

Specifications are also the products of human design, so...

- They will contain **bugs**
  - (recall the central dogma of this course)
  - harder to fix the more people that have seen it
    - “turns to stone” a bit more with each viewer
- Creating them requires **judgement**
  - no “turn the crank” way to produce good specs (or invariants)
  - harder but that’s why it’s interesting work