
CSE 331

Software Design & Implementation

Kevin Zatloukal
Spring 2022
Lecture 5 – Specifications

Administrivia

- HW2 released Wednesday
 - starts and ends with reasoning problems
 - middle parts are about learning to use the tools
- Pushed a small update to the code early Thursday
 - pull that change as soon as you can
 - make sure you *merge* with any changes you made

Goals

We want our code to be:

1. Correct
 - everything else is secondary
2. Easy to change
 - most code written is changing existing systems
3. Easy to understand
 - corollary of previous two
4. Easy to scale
 - modular

Specifications

To prove correctness of our method, 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

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**

- need to tell clients what it will do so they can start building their own parts of the system

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 (modularity)
 - 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



Isn't the interface sufficient?

The interface defines the boundary between implementers and users:

```
public class MyList implements List<E> {  
    public E get(int x) { return null; }  
    public void set(int x, E y){}  
    public void add(E elem) {}  
    public void add(int index, E elem){}  
    ...  
}
```

Interface provides the *syntax and types*

But nothing about the *behavior and effects*

- provides **too little** information to clients

Why not just read code?

```
static <T> boolean ???(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;
}
```

How long does it take you to figure out what this does?

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;
}
```

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
 - would make it impossible to build million-line programs
- 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
 - client needs 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

Most comments convey only a vague idea of what that the code does:

```
// Returns the location of the largest value  
// in the first n elements of the array arr  
int maxLoc(int[] arr, int n) {
```

Ambiguity remains:

- what if $n = 0$
- what if `arr.length < n`?
- what if there are two maximums?

Comments are essential

Most comments convey only a vague idea of what that the code does:

```
// This method checks if "part" appears as a  
// subsequence in "src"  
static <T> boolean sub(List<T> src, List<T> part)
```

Ambiguity remains:

- should be True if **part** is empty and False if **src** is empty
- what if **src** and **part** are both empty?

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

A more careful description of sub

// Check whether "part" appears as a subsequence in "src"

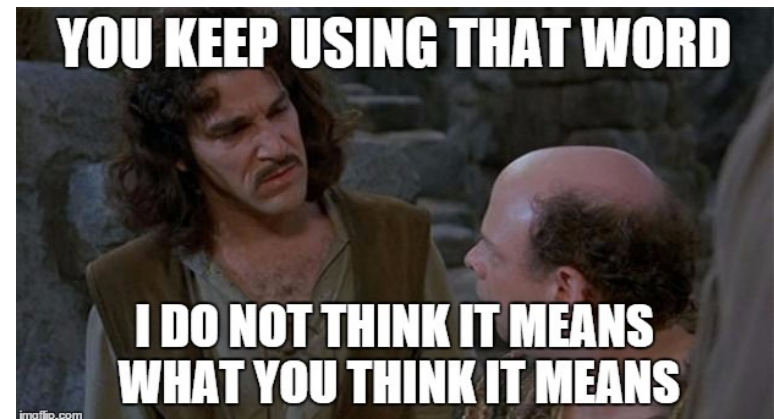
needs to be given some caveats:

*// * src and part cannot be null*

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

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 subsequence in "src"

needs to be given some caveats:

```
// * 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...
```

A better approach

It's better to simplify than to describe complexity!

Complicated description suggests poor design

- rewrite **sub** to be more sensible, and easier to describe

```
// Returns true iff there exist sequences A and B (possibly  
// empty) such that src = A + part + B, where + means concat  
static <T> boolean sub(List<T> src, List<T> part) {
```

- Mathematical flavour not always necessary, but avoids ambiguity
- “Declarative” style is important: avoids reciting or depending on operational/implementation details

Sneaky fringe benefit of specs

- 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
 - Very important feature of Java (copied by others)
- Javadoc convention for writing specifications
 - Method signature
 - 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: these are abbreviated.
In your code, it must be
`@spec.requires,`
`@spec.modifies, etc.`

- The *precondition*: constraints that hold before the method is called (if not, all bets are off)
 - **@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 untouched
 - **@effects**: gives guarantees on final state of modified objects
 - **@throws**: lists possible exceptions and conditions under which they are thrown (Javadoc uses this too)
 - **@return**: describes return value (Javadoc uses this too)

Example 1

static <T> int **changeFirst**(List<T> **lst**, T **oldelt**, T **newelt**)
 requires **lst** is non-null
 modifies **lst**
 effects change the first occurrence of **oldelt** in **lst** to **newelt**
 (making no other changes to **lst**)
 returns the position of the element in **lst** that was **oldelt** and
 is now **newelt** or -1 if not in **oldelt**

```
static <T> int changeFirst(
    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 (or leave these off)
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` `i`th element of `lst2` is added to the `i`th element of `lst1`

`returns` none (or leave this off)

```
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));  
    }  
}
```

Should requires clause be checked?

- Preconditions are common in ordinary classes
 - in public libraries, necessary to deal with all possible inputs
- If the client calls a method without meeting the precondition, the code is free to do *anything*
 - including pass corrupted data back
 - it is a good idea to *fail fast*: to provide an immediate error, rather than permitting mysterious bad behavior
- Rule of thumb: Check if cheap to do so
 - Example: number has to be positive → check
 - Example: list has to be sorted → skip
 - Be judicious if private / only called from your code

Comparing specifications

- Occasionally, we need to compare different specification:
 - comparing potential specifications of a new class
 - comparing new version of a specification with old
 - recall: most work is making changes to existing code
- For that, we often consider *stronger* and *weaker* specifications...

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.”*

Stronger vs Weaker Specifications

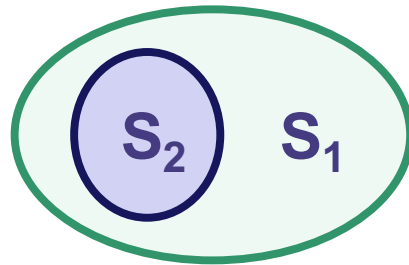
- **Definition 1:** specification S_2 is stronger than S_1 iff
 - for any implementation M : M satisfies $S_2 \Rightarrow M$ satisfies S_1
 - i.e., S_2 is harder to satisfy



- 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

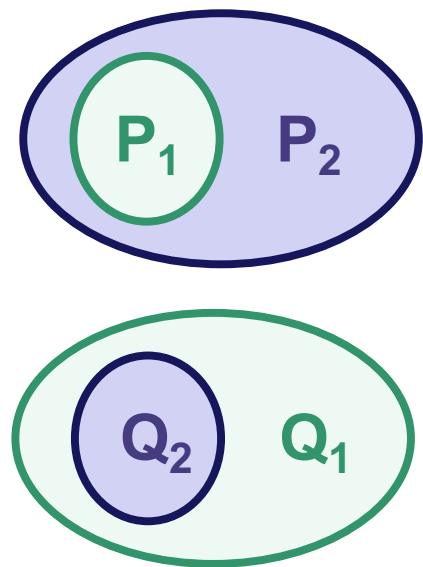


Making changes to a specification...

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

Stronger vs Weaker Specifications

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



Example 1 (stronger postcondition)

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

Which is stronger?

- 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**

Example 2 (weaker precondition)

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

Which is stronger?

- 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**

Example 3

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

Which is stronger?

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

Strengthening a specification



- Strengthen a specification by:
 - Promising more (stronger postcondition):
 - returns clause harder to satisfy
 - effects clause harder to satisfy
 - fewer objects in modifies clause
 - more specific exceptions (subclasses)
 - Asking less of client (weaker precondition)
 - requires clause easier to satisfy
- Weaken a specification by:
 - (Opposite of everything above)

“Strange” case: @throws

Compare:

S1:

@throws FooException if $x < 0$

@return $x + 3$

S2:

@return $x + 3$

S3:

@requires $x \geq 0$

@return $x + 3$

- S1 & S2 are *stronger* than S3
- S1 & S2 are *incomparable* because they promise different, incomparable things when $x < 0$

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”

Warnings on Specifications

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

- They will contain **bugs**
 - (recall the central dogma of this course)
- Creating them requires **judgement**
 - no mechanical way to produce good specs (or invariants)
 - harder but good for job security
- Harder to fix the more people that use it
 - Medusa effect: “turns to stone” a bit more with each look
 - widely used parts become impossible to change

XKCD 1172



Back to Correctness...

Correctness Toolkit

- Learned forward and backward reasoning for
 - assignment
 - if statement
 - while loop
- One missing element: function calls
 - we needed specifications for that
 - now we have them

Reasoning about Function Calls

```
static int f(int a, int b) { ... }
```

requires $P(a,b)$ -- some assertion about a & b

returns $R(a,b,c)$ -- some assertion about a, b, & c (returned)

Forward

```
{{ P1 }}
```

```
  c = f(a, b);
```

Reasoning about Function Calls

```
static int f(int a, int b) { ... }
```


requires $P(a,b)$ -- some assertion about a & b

returns $R(a,b,c)$ -- some assertion about a, b, & c (returned)

Forward

```
{{ P1 }}  
  c = f(a, b);  
{{ P1 and R(a,b,c) }}
```

if $P1$ implies $P(a,b)$



Reasoning about Function Calls

```
static int f(int a, int b) { ... }
```

requires $P(a,b)$ -- some assertion about a & b

returns $R(a,b,c)$ -- some assertion about a, b, & c (returned)

Backward

```
c = f(a, b);  
{ { Q } }
```

Reasoning about Function Calls

```
static int f(int a, int b) { ... }
```

requires $P(a,b)$ -- some assertion about a & b

returns $R(a,b,c)$ -- some assertion about a , b , & c (returned)

Backward

solve $R(a,b,c)$ for c
substitute c appears in Q

↑
 $\{ \{ Q[c / f(a,b)] \text{ and } P(a,b) \} \}$
 $c = f(a, b);$
 $\{ \{ Q \} \}$