CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2020

Lecture 5 – Specifications

Reminders

- HW2 (pt 2) due Monday

- HW3 due Wednesday
  - your git repository should exist now (check your email)
  - should be quick **unless** there are issues with the tools
    - make sure you leave time for that possibility
  - documentation about the tools on the web site
    - plan to spend some time reading...

CSE 331: Software Design & Implementation | Home | Calendar | Assignments | Lectures | Sections | Resources | Message Board

- Home
- Syllabus
- Academic Integrity
- Resources
- Zoom
- Canvas

Resources

Concepts

Class and Method Specifications
Writing Rep Invariants and Abstraction Functions
A Guide to Testing
How to Debug

Languages

Java Q&A
React Tips & Tricks

Tools

Project Software Setup
Editing, Compiling, Running, and Testing Java Programs
Version Control (Git) Reference
Assignment Submission

CSE 331 Infrastructure Videos

All project help info (8 videos)
Project software setup (direct links - creating SSH keys and IntelliJ git clone)
Project submission and repo management (direct links - repo clone, commit, tagging, etc.)

Recap + Q & A + Exercises

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 for

- **correctness**: part of our Hoare triple
- **changeability**: make clear what will/won't change
- **understandability**: clients only read spec, not code
- **modularity**: can work independently once spec is fixed

Formalizing specifications also rewards designs that are

- easy to describe clearly
- easy to describe concisely

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 good for job security

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

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
  - `@throws`: lists possible exceptions and conditions under which they are thrown (Javadoc uses this too)
  - `@effects`: gives guarantees on final state of modified objects
  - `@return`: describes return value (Javadoc uses this too)

Example 1

static <T> int changeFirst(List<T> lst, T oldelt, T newelt)

- requires lst, oldelt, and newelt are non-null
- 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 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
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

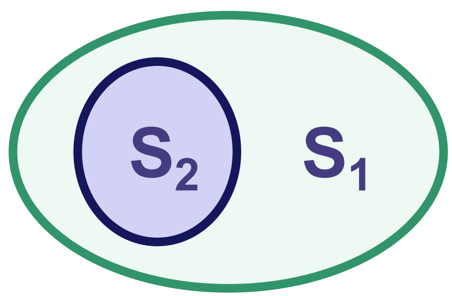
```
static void listAdd(
    List<Integer> lst1, List<Integer> lst2) {
  for(int i = 0; i < lst1.size(); i++) {
      lst1.set(i, lst1.get(i) + lst2.get(i));
  }
}
```

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: list has to be non-null → check
  - Example: list has to be sorted → skip
  - Be judicious if private / only called from your code

Stronger vs Weaker Specifications

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



(satisfying **implementations**)

- An implementation satisfying a stronger specification can be **used anywhere** that a weaker specification is required
  - can substitute a procedure satisfying a stronger spec

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

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`**

“Strange” case: @throws

Compare:

S1:

@throws FooException if x<0

@return x+3

S2:

@return x+3

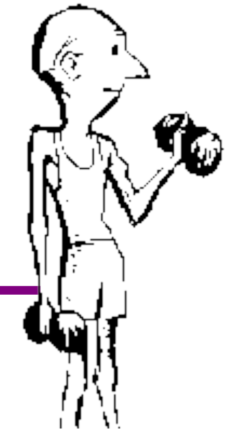
S3:

@requires x >= 0

@return x+3

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

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)

More Q & A