

---

# CSE 331

# Software Design & Implementation

Hal Perkins

Fall 2015

Lecture 4 – Specifications

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins)

---

# Administrivia

---

- Next two assignments out now
  - HW2: Written problems on loops, due Tue. night
  - HW3: Java warmup & project logistics
    - Due Thur. night
    - Should go quickly, but please start early so we can fix setup problems before the last minute
- Lots of new readings related to next few lectures – dig in if you haven't already
  - Quizzes coming soon 😊

# 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?

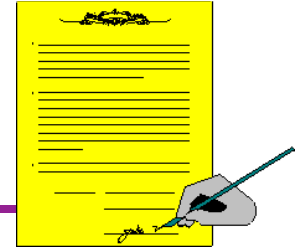
# A discipline of modularity

---

- Two ways to view a program:
  - The implementer's view (how to build it)
  - The client's view (how to use it)
- It helps to apply these views to program parts:
  - While implementing one part, consider yourself a client of any other parts it depends on
  - Try *not* to look at those other parts through an implementer's eyes
  - Helps dampen 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



# Isn't the interface sufficient?

---

The interface defines the boundary between implementers and users:

```
public class List<E> {  
    public E get(int x) { return null; }  
    public void set(int x, E y){}  
    public void add(E) {}  
    public void add(int, E){}  
    ...  
    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 is not (completely) legal Java*

- *Parameters need names; no static interface methods before Java 8*



# 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
  - 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 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...*

# 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 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: avoids reciting or depending on operational/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
  - 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

---

- 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 **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));
    }
}
```

# Example 4 (Watch out for bugs!)

---

static void `uniquify`(List<Integer> `lst`)

`requires` ???

???

`modifies` ???

`effects` ???

`returns` ???

---

```
static void uniquify(List<Integer> lst) {  
    for (int i=0; i < lst.size()-1; i++)  
        if (lst.get(i) == lst.get(i+1))  
            lst.remove(i);  
}
```



# 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 bad behavior
- Preconditions are common in “helper” methods/classes
  - In public libraries, it’s friendlier to deal with all possible input
  - *Example: binary search would normally impose a precondition 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*

# Satisfaction of a specification

---

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

*$M$  satisfies  $S$*  if and only if

- Every behavior of  $M$  is permitted by  $S$
- “The behavior of  $M$  is a subset of  $S$ ”

The statement “ $M$  is correct” is meaningless!

- Though often made!

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

- “*One person’s feature is another person’s bug.*”
- Usually better to change the program than the spec

# Sneaky fringe benefit of specs #2

---

- Specification means that client doesn't need to look at implementation
  - So the code may 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)

# Comparing specifications

---

- Occasionally, we need to compare different versions of a specification (*Why?*)
  - For that, talk about *weaker* and *stronger* specifications
- A weaker specification gives greater freedom to the implementer
  - If specification  $S_1$  is weaker than  $S_2$ , then for any implementation  $M$ ,
    - $M$  satisfies  $S_2 \Rightarrow M$  satisfies  $S_1$
    - but the opposite implication does not hold in general
- Given two specifications, they may be *incomparable*
  - Neither is weaker/stronger than the other
  - *Some* implementations might still satisfy them both

# 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

# Example 1

---

```
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**

## Example 2

---

```
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

# Stronger and weaker specifications

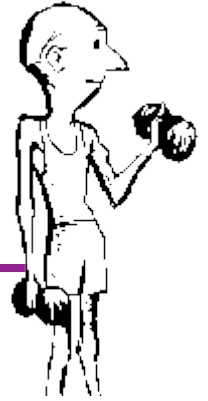
---

- A stronger specification is
  - Harder to satisfy (more constraints on the implementation)
  - Easier to use (more guarantees, more predictable, client can make more assumptions)
- A weaker specification is
  - Easier to satisfy (easier to implement, more implementations satisfy it)
  - Harder to use (makes fewer guarantees)



# Strengthening a specification

---



- Strengthen a specification by:
  - Promising more – any or all of:
    - Effects clause harder to satisfy
    - Returns clause harder to satisfy
    - Fewer objects in modifies clause
    - More specific exceptions (subclasses)
  - Asking less of client
    - Requires clause easier to satisfy
- Weaken a specification by:
  - (Opposite of everything above)

# “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”

# More formal stronger/weaker

---

- A specification is a logical formula
  - S1 stronger than S2 if S1 implies S2
  - From implication all things follows:
    - Example: S1 stronger if requires is weaker
    - Example: S1 stronger if returns is stronger
- As in all logic (cf. CSE311), two rigorous ways to check implication
  - Convert entire specifications to logical formulas and use logic rules to check implication (e.g.,  $P1 \wedge P2 \Rightarrow P2$ )
  - Check every *behavior* described by stronger also described by the other
    - CSE311: truth tables
    - CSE331: *transition relations*

# Transition relations

---

- There is a program state before a method call and after
  - All memory, values of all parameters/result, whether exception happened, etc.
- A specification “means” a set of pairs of program states
  - The legal pre/post-states
  - This is the transition relation defined by the spec
    - Could be infinite
    - Could be multiple legal outputs for same input
- Stronger specification means the transition relation is a subset
- Note: Transition relations often are infinite in size