CSE 331
Software Design & Implementation

Hal Perkins

Autumn 2012

Specifications

(Slides by Mike Ernst)

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
  - It's 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 (and we <u>know</u> it because we can prove it!)
  - Testing: OK, we know it's right, but is it?
  - Debugging: If it's not, how do we systematically find the problems and fix them?
  - Maintain: 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? (Comments, JavaDoc, UML(?), …)

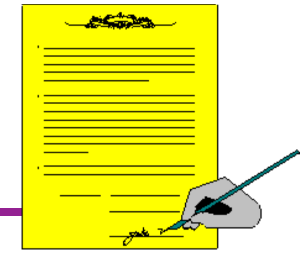
The challenge of scaling software

- 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
  - This 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 by *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 is to defines the boundary between the implementers and users:

```
public interface List<E> {
        public E get(int);
        public void set(int, E);
        public void add(E);
        public void add(int, E);
        …
        public static boolean sub(List<T>, List<T>);
}
```

Interface provides the syntax

But nothing about the behavior and effects

Why not just read code?

```
boolean sub(List<?> src, List<?> part) {
    int part_index = 0;
    for (Object o : src) {
        if (o.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 in order 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 just 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"
boolean sub(List<?> src, List<?> part) {
        ...
}
```

- *Problem: ambiguity remains*
  - *e.g. what if src and part are both empty lists?*

From vague comments to specifications

- ***Properties of a specification:***
  - The client agrees to rely *only* on information in the description in their use of the part.
  - The 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 simply running it, then depending on the results
  - This leads to bugs and to programs with unclear dependencies, reducing simplicity and flexibility

Recall the sublist example

```
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 sequences A, B exist such that
//   src = A : part : B
// where ":" is sequence concatenation
boolean sub(List<?> src, List<?> part)
```

Mathematical flavor is not (always) necessary, but can (often) help 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
  - 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

Examples of specifications

- Javadoc
  - Sometimes can be daunting; get used to using it
- Javadoc convention for writing specifications
  - method prototype
  - text description of method
  - param: description of what gets passed in
  - returns: description of what gets returned
  - throws: list of 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

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 (Javadoc uses this too)
  - effects: gives guarantees on the final state of modified objects
  - returns: describes return value (Javadoc uses this too)

Example 1

static int test(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 int test(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> listAdd(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> listAdd(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 listAdd2(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 listAdd2(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?

- 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 pre-condition rather than simply failing if list is not sorted. Why?*
- Rule of thumb: Check if cheap to do so
  - *Ex: list has to be non-null → check*
  - *Ex: list has to be sorted → skip*

Comparing specifications

- Occasionally, we need to compare different versions of a specification (*Why?*)
  - For that, we 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 I,
    - I satisfies S_2 => I satisfies S_1
    - but the opposite implication does not hold in general

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

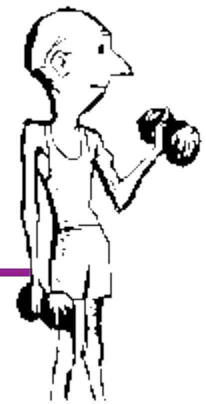
```
int find(int[] a, int value) {
    for (int i=0; i<a.length; i++) {
        if (a[i]==value) return i;
    }
    return -1;
}
```

- 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 (harder to implement)
  - Easier to use (more guarantees, more predictable)
- 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
    - effects clause harder to satisfy, and/or fewer objects in modifies clause
  - asking less of client
    - requires clause easier to satisfy
- weaken a specification by:
  - promising less
    - effects clause easier to satisfy, and/or extra objects in modifies clause
  - asking more of the client
    - requires clause harder to satisfy

Choosing specifications

- There can be different specifications for the same implementation!
  - Specification says more than implementation does
  - Declares which properties are essential – the method itself leaves that ambiguous
  - Clients know what they can rely on, implementers know what they are committed to
- Which is *better* : a strong or a weak specification?
  - It depends!
  - Criteria: simple, promotes reuse & modularity, efficient

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, as we'll see shortly