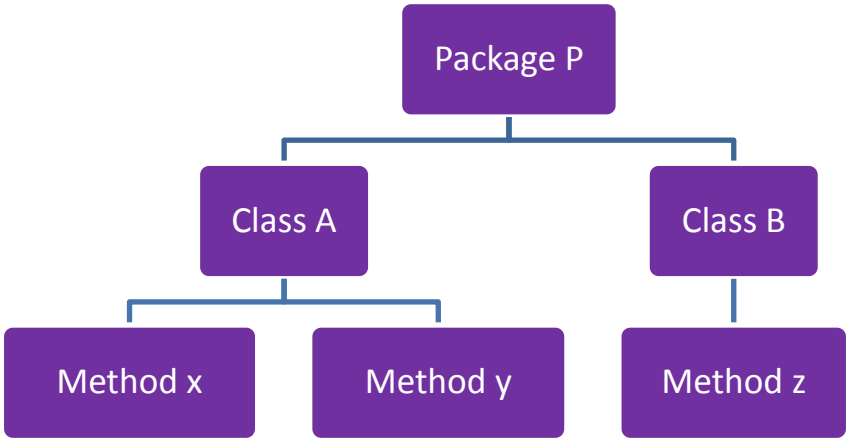
Specifications

CSE 331

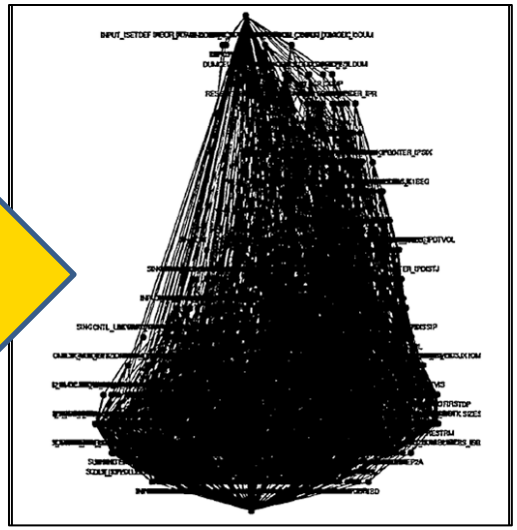
Autumn 2010

The challenge of scaling software

- Small programs tend to be simple and malleable
  - easy to write, easy to change
- Big programs tend to be complex and inflexible
  - hard to write, hard to change
- Why?
  - Because interactions become increasingly unmanageable



More classes, more methods, more calls, more generics, more imports, more inherits, more libraries, more static and private and public, …

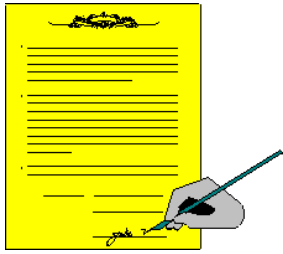


A discipline of modularity

- We aim at simpler and more malleable programs by viewing a program in two ways
  - How to build it: the implementer's view
  - How to use it: the client's view
- When you are wearing one “hat,” make as few assumptions about what someone wearing the other “hat” will decide
  - When you are constructing a client, make as few assumptions as possible about how others parts you depend upon are implemented
  - When you are constructing an implementation, make as few assumptions as possible about clients that may use the implementation

- **This separation is formalized through the idea of a specification**

A specification is a contract

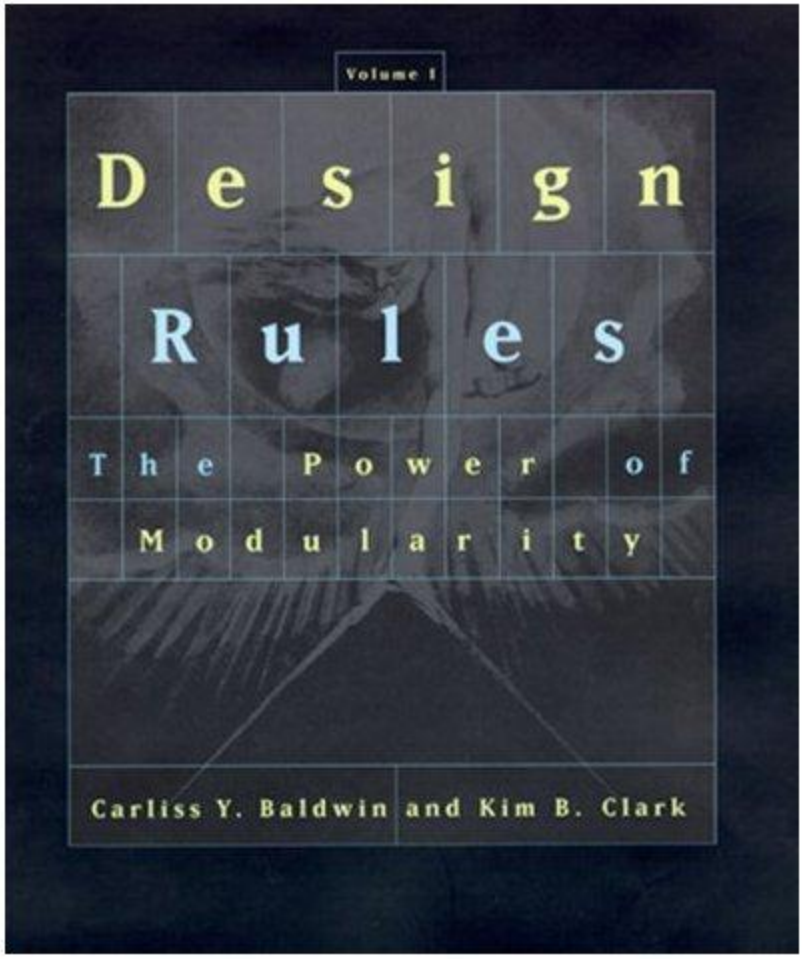


- A set of requirements agreed to by the user and the manufacturer of the product
  - It describes their expectations of each other
- Two-way isolation improves clarity of expectations and discourages implicit expectations
  - Isolate client from implementation details
    - At least for the moment, **you** are not responsible for the implementation
  - Isolate implementation from how the part is used
    - At least for the moment, **you** are not choosing what the clients do and how they do it
- Facilitates change
  - Making the specification more stable allows the client and the implementation to change more independently

An aside:
Design Rules • *The Power of Modularity*

"…. [Baldwin and Clark] develop a powerful theory of design and industrial evolution. They argue that the [computing] industry has experienced previously unimaginable levels of innovation and growth because it embraced the concept of *modularity*, building complex products from smaller subsystems that can be designed independently yet function together as a whole. Modularity freed designers to experiment with different approaches, as long as they obeyed the established *design rules*. …"

–amazon.com



Isn't a (Java) *interface* sufficient?

- The *interface* defines the boundary between the implementers and the clients

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

- It provides the syntax but nothing on the behavior and effects
- What do you think that ***add(E)*** does and ***sub(List<T>, List<T>)*** does? Why?

Why not just read the 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;
  }
```

- Code gives more detail than the client needs
- Understanding or even reading every line of code is a burden
  - Suppose you had to read source code of Java libraries to use them?
- Client cares only about what the code does, not how it does it

Code is vague

- A piece of code may be unambiguous **and** vague
  - Reading code lets you determine how it will execute, but it may not let you distinguish essential from incidental details
- This is key as the code is changed
  - Client needs to know what they can rely on over time
    - What properties might be changed by later optimization, improved algorithms, or bug fixes, etc.?
  - Implementer needs to know what features the client depends on, and which can be changed

Comments: essential but insufficient

- Most comments convey only an informal, general idea of what that the code does

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

- Ambiguity remains
  - Ex: what if ***src*** and ***part*** are both empty lists?

Towards 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, but otherwise is perfectly at liberty
- However, 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

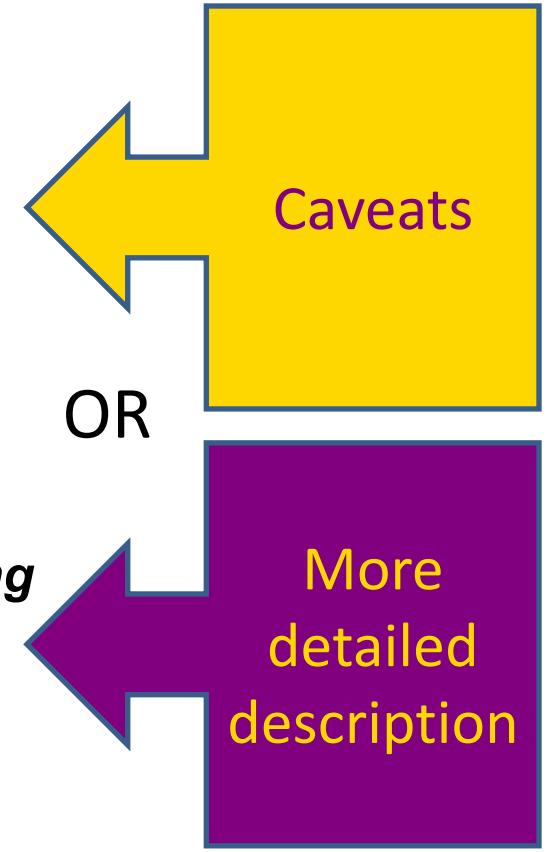
Recall the sublist example

```
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
// subsequence in "src".

// * 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).



// 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

- Complicated description suggests poor design
- Rewrite ***sub()*** to be more sensible and easier to describe. Then a good description would be:

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

- This is a decent specification
  - Mathematical flavor is not necessary, but can help avoid ambiguity

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 then the 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 – remember, False $\Rightarrow$ True)
  - 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 on lst that was oldelt and 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 not 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 not 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));
        }
}
```

example: java.util.Arrays.binarySearch

binarySearch

public static int binarySearch(int[] a,int key)

Searches the specified array of ints for the specified value using the binary search algorithm. The array must be sorted (as by the `sort` method, above) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a`- the array to be searched.

`key`- the value to be searched for.

Returns:

index of the search key, if it is contained in the list; otherwise, `(-(insertion point) - 1)`. (long description...)

Improved binarySearch specification

public static int **binarySearch**(int[] a,int key)

requires: a is sorted in ascending order

returns:

- some i such that a[i] = key if such an i exists,
- otherwise -1

(Returning `(-(`*`insertion point`*`) - 1)` *is very ugly, and an invitation to bugs and confusion; please read full specification and think about why the designers did this, and what the alternatives are. We'll return to the topic of exceptions and special values in a later lecture.)*

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 better, however, to *fail-fast: to provide an immediate error, rather than simply letting mysterious bad stuff happen*
- Preconditions are more reasonable to use in "helper" methods/classes than in public libraries – friendlier to just deal with all possible input
  - *Why does binarySearch impose a precondition rather than simply failing if list is not sorted?*
- 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
  - We talk about "weaker" and "stronger" specifications
- Intuitively, we weaken a specification when we change it to give 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 necessarily hold

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* (implement) because it promises more – that is, its effects clause is harder to satisfy and/or there are fewer objects in modifies clause – but
  - *easier to use* (more guarantees, more predictable) by the client – that is, the requires clause is easier to satisfy
- A weaker specification is
  - *easier to satisfy* (easier to implement and more implementations satisfy it) because it promises less – that is, the effects clause is easier to satisfy and/or there are more objects in modifies clause – but
  - *harder to use* (makes fewer guarantees) because it asks more of the client – that is, the requires clause is harder to satisfy

Choosing specifications

- There are different specifications for the same implementation (and vice versa)
  - Specification says more than method 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 and modularity, efficient

Sneaky fringe benefit of specs #2

- Specification means that client doesn't need to look at implementation
  - So 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

Whoa, that was fast!

- Reread these slides
- Read the assignments (see the calendar on the web)
- Do PS0 – and think about these issues in a focused context
- Come to office hours
- You'll get there, for sure