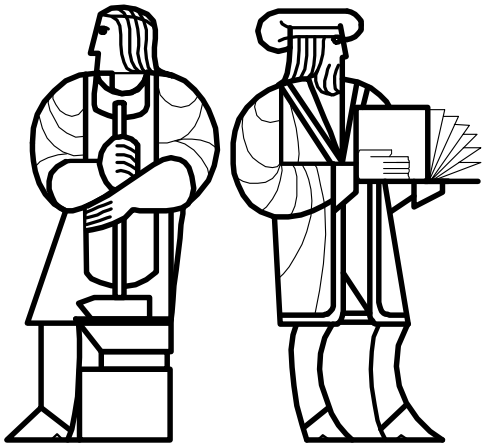
Specifications



6.170

MIT EECS

a useful discipline

We often view a program in two distinct ways:

- The implementor'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 implementor's eyes
- This helps dampen interactions between parts

Formalized through the idea of a specification

Situation

Designing a component for a client to use

Questions:

- How do you tell the client what the component does?
- What do you tell the implementer to deliver?

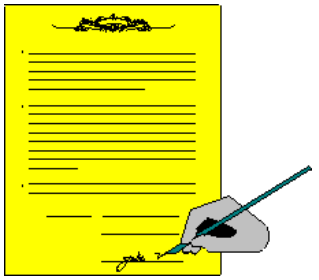
You need a specification

what is a specification?

"Specification is the specific set of (high level) requirements agreed to by the sponsor/user and the manufacturer/producer of a system." - Wikipedia

A specification is a contract

Between client and implementer, describing each other's expectations



Facilitates simplicity

Isolate client from implementation details

Isolate implementor from what use is made of the part

Discourages implicit, unwritten expectations

Facilitates change

Reduces the "Medusa" effect: the specification, rather than the code, gets "turned to stone" by client dependencies

Specification for sqrt

double y = sqrt(double x);

Specification 1

requires: x >= 0

modifies: nothing

returns:

y such that |y*y-x| < eps
and y >= 0

Specification 2

requires: nothing

modifies: nothing

throws:

if (x < 0) throws
NegativeArgument;

returns: if (x < 0) y such that
|y*y-x| < eps and y >= 0

Things to Remember

Specification must give implementer enough room

If client violates precondition (requires clause), implementation can do ANYTHING

Type system usually provides some parts of specification

6.170 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
- effects – gives guarantees on the final state of modified objects
- returns – describes return value

Set Add Specification

static boolean add(Set<T> s, T o);

Requires: s != null and o != null

Modifies: s

Throws: IllegalArgument, OutOfResources

Effects: new s = s union { o }

Returns: true iff o not in s

Remarks:

- Need to refer to original values and new values
- Very precise, digital feel
- Abstract specification of set

Google Specification

static List<URL> Google(String s);

Requires: nothing

Modifies: nothing

Effects: nothing

Returns: no constraint

Throws: InternalError

Remarks

What is the simplest implementation of this specification?

Illustrates limitations of specification-based approach

Isn't the interface sufficient?

The main reason for the interface is to define the boundary between the implementers and users:

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

Interface provides the syntax

But doesn't say what it does

why not just read code?

```
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
- For large program, 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 code does, not how it does it

Problems with Code

Code may be wrong, code may be overly specific

Code invariably gets rewritten, so 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?

Implementor needs to know what features the client depends on, and which can be changed

Nasty problem in practice: bugs become part of specification…

what about comments?

Much of the code on the planet is decorated with short descriptions designed to convey the general idea of what that the code does:

```
// Check whether "part" appears as a
// sub-sequence in "src".
T boolean sub(List<T> src, List<T> part) {
   ...
}
```

Now, the client can often get along without reading the code, and if there's ambiguity they can just run a test to see what happens, right?

e.g. what if src and part are both empty list?

beyond the “general idea”

A description of a part becomes a specification if:

The client can and agrees to rely only on information in the description in their use of the part.

The implementor of the part promises to support everything in the description, but otherwise is perfectly at liberty

But this is not common behavior!

Clients often work out what a method/class does in ambiguous cases by simply running it, then depending on the results

This leads to programs with unclear dependencies, reducing simplicity and flexibility

Let's look at this code again…

```
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:

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

or reorganize to be simple

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
T boolean sub(List<T> src, List<T> part)
```

This is a decent specification

Mathematical flavor is not necessary, but can help avoid ambiguity

One Benefit of Specifications

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 hack that assumes no partial matches before true matches – but cost of forcing client to understand the details is too high

Another Example

public static List<Integer> listAdd(List<Integer> lst1, List<Integer> lst2)

requires

modifies

effects

returns ;um

```
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;
}
```

Yet Another Example

static void listAdd2(List<Integer> lst1, List<Integer> lst2)

requires

modifies

effects

returns

```
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));
   }
}
```

6.170 binarySearch

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

requires: a is sorted in ascending order
modifies: none
effects: none
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 special values in a later lecture.*

comparing specifications

When is specification S1 weaker than S2?

When $\forall$ P, (P satisfies S2) $\Rightarrow$ (P satisfies S1)

Intuitively, we weaken a specification when we change it to give greater freedom to the implementor

We can weaken a specification by

Making requires harder to satisfy

Adding things to modifies clause

Making effects easier to satisfy

What is the strongest (most constraining) requires clause?

true

What is the strongest (most constraining) effects clause?

false

What is the strongest (most constraining) modifies clause?

modifies nothing

Another Benefit of Specifications

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 implementors to different modules

- Allows teamwork and parallel development (this is crucial, as you'll see towards the end of term)
- Also helps with testing, as we'll see shortly