

Procedural Abstraction and Specifications

6.170 Lecture 4

Spring 2004

“... when you are a Bear of Very Little Brain, and you Think of Things, you find sometimes that a Thing which seemed very Thingish inside you is quite different when it gets out into the open and has other people looking at it.”

A.A. Milne, *The House At Pooh Corner*

1 Introduction

In this lecture, we'll look at the role played by specifications of methods. Specifications are the linchpin of team work. It's impossible to delegate responsibility for implementing a method without a specification. The specification acts as a contract: the implementor is responsible for meeting the contract, and a client that uses the method can rely on the contract. In fact, we'll see that like real legal contracts, specifications place demands on both parties: when the specification has a precondition, the client has responsibilities too.

Many of the nastiest bugs in programs arise because of misunderstandings about behavior at interfaces. Although every programmer has specifications in mind, not all programmers write them down. As a result, different programmers on a team have different specifications in mind. When the program fails, it's hard to determine where the error is. Precise specifications in the code let you apportion blame (to code fragments, not people!), and can spare you the agony of puzzling over where a fix should go.

Specifications (or *specs*) are good for the client of a method because they spare her the task of reading code. If you're not convinced that reading a spec is easier than reading code, take a look at some of the standard Java specs and compare them to the source code that implements them. `ArrayList`, for example, in the package `java.util`, has a very simple spec but its code is not at all simple. Specifications are good for the implementor of a method because they give her freedom to change the implementation without telling clients. Specifications can make code faster too. Sometimes a weak specification makes it possible to do a much more efficient implementation. In particular, a precondition may rule out certain states in which a method might have been invoked that would have incurred an expensive check that is no longer necessary.

2 Contracts, firewalls, and decoupling

In traditional engineering disciplines, artifacts can often be constructed out of components taken off the shelf. These components are selected on the basis of specifications. This is one crucial role for specification, and indeed it plays this role in software. Unfortunately, however, software components have been largely an unfulfilled aspiration; we have small components (such as string manipulation packages) and huge components (such as relational databases), but very little in

between the two. The problem is that medium-sized components seem to be inflexible, and make conflicting assumptions about the structure of the program in which they are embedded. So if you use one of them, you typically can't use another.

And of course software is notoriously unreliable, so its specifications often can't be taken too seriously. Some companies are more honest than others about the guarantees they offer:

Cosmotronic Software Unlimited Inc. does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error-free. However, Cosmotronic Software Unlimited Inc. warrants the diskette(s) on which the program is furnished to be of black color and square shape under normal use for a period of ninety (90) days from the date of purchase.

We don't claim Interactive EasyFlow is good for anything . . . if you think it is, great, but it's up to you to decide. If Interactive EasyFlow doesn't work: tough. If you lose a million because Interactive EasyFlow messes up, it's you that's out of the million, not us. If you don't like this disclaimer: tough. We reserve the right to do the absolute minimum provided by law, up to and including nothing. This is basically the same disclaimer that comes with all software packages, but ours is in plain English and theirs is in legalese.

ACM Software Engineering Notes, Vol. 12, No. 3, 1987.

A specification imposes contractual obligations on both the client (user) of some unit (method, module, etc.), and on the implementor of that unit. The contract is understood as an implication:

`client-meets-obligation => implementor-meets-obligation`

where ' \Rightarrow ' denotes logical implication. In other words, if the client acts according to the terms of the specification, then the implementor must also do their part – but if the client does not meet her obligations, then the implementor is not obliged to do anything. For example, your contract with the electricity utility places an obligation on you not to exceed a certain load, and an obligation on the utility to provide power at a fixed voltage with only small fluctuations. If you don't exceed the load, and just run an air-conditioner and a handful of light bulbs, the utility maintains the voltage. But if you decide to run a server farm in your basement that takes a megawatt of power, the utility is not obliged to provide anything.

The contract acts as a *firewall* between client and implementor. It shields the client from the details of the *workings* of the unit – you don't need to read the source code of the method if you have its specification. And it shields the implementor from the details of the *usage* of the unit – the implementor doesn't have to ask every client how she plans to use the unit. This firewall results in *decoupling*, allowing the code of the unit and the code of a client to be changed independently, so long as the changes respect the specification – each side meeting their obligation to the other.

Specifications actually play two subtly different roles in software. One is to catalog reusable components: this is the purpose of the specifications in the Java collections framework, for example. The other is to regulate the connections between modules in a design. In this role, a single unit may have multiple specifications, one for each client. The specifications qualify the 'uses' relationship between modules, saying exactly how one module uses another. As the system evolves, these specifications are the part that is least affected. Consequently, perhaps more than anything else, these specifications characterize the design of the software – they *are* the design. When we study design patterns, we'll see how the motivation of most design patterns is to improve the decoupling of modules, and this is usually achieved by introducing new specifications, which are *weaker* than the specifications used in simpler designs.

3 Equivalence of implementations

Consider these two methods. Are they the same or different?

```
findA:      static int findA (int[] a, int val) {
                for (int i = 0; i < a.length; i++) {
                    if (a[i] == val) return i;
                }
                return a.length;
            }

findB:      static int findB (int[] a, int val) {
                for (int i = a.length-1 ; i > 0; i--) {
                    if (a[i] == val) return i;
                }
                return -1;
            }
```

Of course the details of the code are different, so in that sense the methods are different. A more important question though is whether one could substitute one implementation for the other. Not only do these methods have different code; they actually have different behavior:

- ▷ when `val` is not in the array `a`, `findA` returns the length of `a` while `findB` returns `-1`;
- ▷ when `val` appears multiple times in `a`, then `findA` returns the lowest index and `findB` returns the highest.

But when `val` occurs at exactly one index of the array, the two methods behave the same. It may be that clients never rely on the method's behavior in the other cases. So the notion of equivalence is in the eye of the beholder, that is, the client. In order to make it possible to substitute one implementation for another, and to know when this is acceptable, we need a specification that states exactly what the client depends on.

4 Structure of specifications

A specification of a method consists of several *clauses*:

- ▷ a **precondition**, indicated by the keyword *requires*;
- ▷ a **postcondition**, indicated by the keyword *effects*;
- ▷ a **frame condition**, indicated by the keyword *modifies*.

The **precondition** is an obligation on the client (i.e. the caller of the method). It is a condition over the state in which the method is invoked. If the precondition does not hold, the implementation of the method is free to do anything (including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc.).

The **postcondition** is an obligation on the implementor of the method. If the precondition holds for the invoking state, the method is obliged to obey the postcondition, by returning appropriate values, throwing specified exceptions, modifying or not modifying objects, and so on.

The **frame condition** is related to the postcondition. It allows more succinct specifications. Without a frame condition, it would be necessary to describe how all the objects accessible to a method may or may not change when the method is called. But usually only a few of these objects

are modified. The frame condition explicitly identifies which objects may be modified. If we say ‘*modifies x*’, this means that the object *x* may be modified by the method, but no other object may be. So in fact, the frame condition or *modifies clause* as it is sometimes called is really an assertion about the objects that are *not* mentioned. For the ones that are mentioned, a mutation is possible but not necessary; for the ones that are not mentioned, a mutation may not occur.

Omitted clauses have particular interpretations. If you omit the precondition, it is given the default value `true`. That means that every invoking state satisfies it, so there is no obligation on the caller. In this case, the method is said to be *total*. If the precondition is not `true`, but conditional instead, then the method is said to be *partial* since it is only applicable to some states. If you omit the frame condition, the default is ‘*modifies nothing*.’ In other words, the method makes no changes to any object. Omitting the postcondition makes no sense and is never done.

We have two conventions for writing specifications in this course. One is from the Liskov text, and uses just the *requires/modifies/effects* clauses described so far. The other convention is called *Javadoc6170*, and combines the Liskov clauses with a standard set of *Javadoc tags* used to document Java code. The combined set of tags are:

tag	argument(s)	Description
<i>param</i>	name description	Describes input parameters, with one line per parameter
<i>requires</i>	condition	Declares that condition must hold when the method is called (6.170 extension)
<i>modifies</i>	object₁, object₂, ...	Declares that nothing besides object₁ will be modified by the method, as long as the <i>requires</i> clause holds when it is invoked (6.170 extension)
<i>throws</i>	name description	Describes exceptions the method may throw, with one clause per exception; this tag is also called <i>exception</i>
<i>effects</i>	condition	Declares that condition will hold at exit from the method, as long as the <i>requires</i> clause holds when it is invoked (6.170 extension)
<i>return</i>	value	Declares that the method returns value

See handout S2 (“Tools”) for information on how to include these tags in your code so that they can be converted into external documentation using the `javadoc` utility.

When using the Liskov convention, your *effects* clause should describe the entire postcondition: what the method returns, what exceptions it throws (and when), and what side-effects it has on the objects in the *modifies* clause. When using the *Javadoc6170* convention, your *effects* clause should only describe the side-effects the method has on the objects in the *modifies* clause. Return values and exceptions should be documented in the *return* and *throws* clauses, respectively, and not in the *effects* clause. In 6.170, we’ll accept either convention in problem set solutions, as long as it’s consistent.

5 Specifying the find method

Here is a possible specification for the `find` method introduced in Section 3:

```
static int find (int[] a, int val)
    requires    val occurs exactly once in a
    effects    returns result such that a[result] = val
```

This specification is *deterministic*: when presented with a state satisfying the precondition, the outcome is fully determined. Both `findA` and `findB` given in Section 3 satisfy this specification. If this is the specification on which the client or clients relied, the two methods are equivalent and substitutable for one another. (Of course a method must have the name demanded by the specification; here we are using different names to allow us to talk about the two versions. To use either, you'd have to change its name to `find`).

Here is a slightly different specification:

```
static int find (int[] a, int val)
  requires    val occurs in a
  effects    returns result such that a[result] = val
```

This specification is not deterministic, since if `val` occurs multiple times in the array `a`, more than one result is consistent with the specification. Such a specification is often said to be non-deterministic, but this is a bit misleading. Non-deterministic code is code that you expect to sometimes behave one way and sometimes another. This can happen, for example, with concurrency: the scheduler chooses to run threads in different orders depending on conditions outside the program. But a 'non-deterministic' specification doesn't call for such non-determinism in the code. The behavior specified is not non-deterministic but *under-determined*. In this case, the specification doesn't say which index is returned if `val` occurs more than once; it simply says that if you look up the entry at the index given by the returned value, you'll find `val`.

This specification is again satisfied by both `findA` and `findB`, each of which resolves the underdeterminedness in its own way. A client of `find` cannot rely on or predict which index will be returned, but should not expect the behavior to be truly non-deterministic. Of course, the specification is satisfied by a non-deterministic method too – for example, one that rather improbably tosses a coin to decide whether to start searching from the top or the bottom of the array. But in almost all cases we'll encounter, non-determinism in specifications offers a choice that is made by the implementor at implementation time, and not at runtime.

So, as before, for this specification too, the two versions of `find` are equivalent. Finally, here's a specification that distinguishes the two:

```
static int find (int[] a, int val)
  effects    returns largest result such that a[result] = val or -1 if no such result
```

It is satisfied by `findB` but not `findA`.

Here is another example to illustrate under-determinedness. There is a class `BigInteger` in the package `java.math` whose objects are integers of unlimited size. The class has a method similar to this:

```
public boolean maybePrime()
  effects    if this BigInteger is composite, returns false
```

If this method returns `false`, the client knows the integer is not prime. But if it returns `true`, the integer may be prime or composite. So long as the method returns `false` a reasonable proportion of the time, it's useful. In fact, as the Java API states: the method takes an argument that is a measure of the uncertainty that the caller is willing to tolerate. The execution time of this method is proportional to the value of this parameter. We won't worry about probabilistic issues in this course; we mention this spec simply to note that it does not determine the outcome, and is still useful to clients.

6 Specification for a mutating method

Our specifications of `find` didn't give us the opportunity to illustrate frame conditions and the description of side effects. Here's a specification that describes a method that mutates an object:

```
class ArrayList {  
    ...  
    boolean addAll (ArrayList a)  
        requires      a != null and a != this  
        modifies    this  
        effects     adds the elements of a to the end of this, and returns true  
                    if this changed as a result of call  
}
```

We've taken this, slightly simplified, from the Java `ArrayList` class (recall that an `ArrayList` object stores a sequence of elements, like an array does, but can also grow and shrink by adding and removing elements). First, look at the frame condition: it tells us that only `this` (the object for which the method is called) is modified, so in particular the argument `a` is not mutated – likely to be a crucial property for most clients. Without a policy of specifying potential mutation, clients would need to read the implementation to be confident that `v` is safe to use after calling `addAll`. Second, look at the postcondition. It gives two constraints: the first telling us how `this` is modified, and the second telling us how the return value is determined. Finally, look at the precondition. It tells us that the behavior of the method is not constrained if you call it with a null argument, or if you attempt to add the elements of an `ArrayList` to itself. You can easily imagine why the implementor of the method would want to impose the second constraint: it's not likely to rule out any useful applications of the method, and it makes it easier to implement. The specification allows a simple implementation in which you take an element from `a` and add it to `this`, then go on to the next element of `a` until you get to the end. If `a` and `this` are the same `ArrayList`, this algorithm will not terminate – an outcome permitted by the specification.

7 Declarative versus operations specifications

Roughly speaking, there are two kinds of specifications for methods. *Operational* specifications detail a series of steps that the method performs; for example, pseudocode descriptions are operational. *Declarative* specifications do not give details of intermediate steps – instead, they just give properties of the final outcome, and how it is related to the initial state.

Almost always, declarative specifications are preferable. They are usually shorter, easier to understand, and most importantly, they don't expose implementation details inadvertently that a client may grow to rely on (and then find no longer hold when the implementation is changed). For example, if we want to allow either implementation of `find`, we would not want to say in the specification that the method “goes down the array until it finds `val`”, since aside from being rather vague, this statement suggests that the search proceeds from lower to higher indices and that the lowest will be returned, which perhaps the specifier did not intend.

Here are some examples of declarative specifications, starting with one from the Java `String` class. The `startsWith` method tests whether a string starts with a particular substring:

```

public boolean startsWith(String prefix)
  effects    if prefix = null throws NullPointerException,
             else returns true iff there exists a sequence s such that
             prefix.seq + s = this.seq

```

where `+` represents string concatenation. We have assumed that `String` objects have a *specification field* called `seq` that models the sequence of characters. We will discuss specification fields further in a lecture on Abstract Data Types – for now, simply think of them as an ordinary field/variable belonging to the class, which is defined for the purposes of writing the specification and may not exist in the actual implementation. The postcondition says that the method returns true if there is some suffix which, when concatenated to the argument, gives the character sequence of the string. The absence of a *modifies* clause indicates that no object is mutated (in fact, since `String` is an immutable type, none of its methods will have *modifies* clauses).

Another example from `String`:

```

public String substring(int i)
  effects    if i<0 or i>length(this.seq) throws IndexOutOfBoundsException
             else returns r such that there exists some sequence s with
             length(s) = i and s + r.seq = this.seq

```

This specification shows how a rather mathematical postcondition can sometimes be easier to understand than an informal description. Rather than talking about whether `i` is the starting index, whether it comes just before the `substring` returned, etc., we simply decompose the string into a prefix of length `i` and the returned string.

8 Specification ordering

Suppose you want to substitute one method for another. How do you compare their specifications? We define a notion of specification strength, such that specification **A** is stronger than specification **B** if **A** is harder to satisfy than **B** – that is, if the set of implementations that satisfy **A** is a subset of the implementations that satisfy **B**. This implies that an implementation that satisfies **A** will also satisfy the weaker specification **B**. But an implementation that satisfies **B** may not satisfy the stronger specification **A**.

We speak of preconditions and postconditions as being stronger or weaker based on how they contribute to the strength of the overall specification. If the postcondition becomes harder to satisfy, then the set of acceptable implementations dwindles in number, and the specification becomes stronger. The strongest postcondition is `false`, since it is never satisfied. If the precondition becomes harder to satisfy, then implementations that were previously unacceptable may become acceptable since their operation is specified for a smaller fraction of the input space – and so the specification becomes weaker. The strongest precondition is `true`; the weakest precondition is `false` (since any implementation at all is acceptable in this case).

More formally, a specification **A** is at least as strong as a specification **B** if:

- ▷ **A**'s precondition is no stronger than **B**'s
- ▷ **A**'s postcondition is no weaker than **B**'s, for the states that satisfy **B**'s precondition.

These two rules embody several ideas. They tell you that you can always weaken the precondition; placing fewer demands on a client will never upset her. You can always strengthen

the postcondition, which means making more promises – for example, the method `maybePrime` in Section 7 can be replaced in any context by a method `isPrime` that returns true if and only if the integer is prime. And where the precondition is false, you can do whatever you like. If the postcondition happens to specify the outcome for a state that violates the precondition, you can ignore it, since that outcome is not guaranteed anyway. These relationships between specifications will be important when we look at the conditions under which subclassing works correctly (in our lecture on subtyping and subclassing).

As an example, consider the following specifications for `find`:

```
static int find (int[] a, int val)
```

A	<i>requires</i>	<code>val</code> occurs exactly once in <code>a</code>
	<i>effects</i>	returns <code>result</code> such that <code>a[result] = val</code>
B	<i>requires</i>	<code>val</code> occurs in <code>a</code>
	<i>effects</i>	returns <code>result</code> such that <code>a[result] = val</code>
C	<i>effects</i>	returns largest <code>result</code> such that <code>a[result] = val</code> or <code>-1</code> if no such <code>result</code>
D	<i>requires</i>	<code>val</code> occurs in <code>a</code>
	<i>modifies</i>	<code>a</code>
	<i>effects</i>	returns <code>result</code> such that <code>a[result] = val</code>
E	<i>requires</i>	<code>a.length > 0</code>
	<i>effects</i>	returns largest <code>result</code> such that <code>a[result] = val</code> or <code>-1</code> if no such <code>result</code>
F	<i>effects</i>	returns <code>result</code> such that <code>a[result] = val</code>

Specification **A** and **B** have the same postcondition, but **B**'s precondition is weaker than **A**. Hence **B** is stronger than **A**. Specification **C** has no precondition (or, equivalently, its precondition is '*requires true*'). This is a weaker precondition than that of **B**. The postcondition for **C** is the same as for **B** when **B**'s preconditions hold. Hence **C** is stronger than **B**. This is easy to check by noting that any implementation of **C** would satisfy **B**, but not every implementation of **B** satisfies **C** (for example, a valid implementation of **B** might loop forever if `val` doesn't occur in `a`, which is not acceptable for **C**).

Specification **D** has a *modifies* clause. We can think of this as being part of the postcondition. This clause gives more freedom to the implementor, who is now allowed to modify `a` (although she need not). Adding a *modifies* clause weakens the postcondition by making it easier to satisfy, and hence weakens the overall specification. So **B** is stronger than **D**. Specification **C** is also stronger than **D**. This follows from transitivity, and can be checked by noting that any implementation that satisfies **C** will also satisfy **D**. Specification **C** is stronger than **E**, but **E** is incomparable with the other specifications since its precondition is incomparable (the set of conditions that satisfy one is not a subset or superset of the conditions that satisfy the other). Specification **A** and **D** are also incomparable, since **D**'s precondition and postcondition are *both* weaker than **A**'s (one of the pre- and postconditions needs to be stronger to establish an order). Specification **F** is simply a bad specification, since it is impossible to satisfy when `val` is not in `a`.

9 Judging specifications

What makes a good method? Designing a method means primarily writing a specification. There are no infallible rules, but there are some useful guidelines. About the form of the specification: it should obviously be succinct, clear and well-structured. The content is harder to prescribe.

The specification should be coherent: it shouldn't have lots of different cases. Long argument lists, deeply nested if-statements, and boolean flags are a sign of trouble. Consider this specification:

```
static int minFind (int[] a, int[] b, int val)
    effects      returns smallest index in arrays a and b at which val appears
```

Is this a well-designed method? Probably not: it's incoherent, since it does two things (finding and minimizing) that are not really related. It would be better to use two separate methods.

The results of a call should be informative. Consider the specification of the `put` method from Java's `HashMap` class:

```
Object put (Object key, Object val)
    effects      inserts (key, val) into the mapping, overriding any existing mapping for
                  key, and returns old value for key, unless none, in which case it returns null
```

Note that the precondition does not rule out `null` values, so the hash map can store `null`s. But the postcondition uses `null` as a special return value for a missing key. This means that if `null` is returned, you can't tell whether the key was not bound previously, or whether it was in fact bound to `null`. This is not a very good design, because the return value is useless unless you know you didn't insert `null`s.

The specification should be strong enough. For example, as we will see in a later lecture, "checked exceptions" can be used to signal error conditions from which the client might reasonably be expected to want to recover (as opposed to letting the entire program fail). But there's no point specifying that such an exception will be thrown for a bad argument if arbitrary mutations are also allowed, because a client won't be able to determine what mutations have actually been made – and hence won't easily be able to recover from the error. Here's a specification illustrating this flaw (and also written in an inappropriately operational style):

```
void addAll (ArrayList a)
    effects      adds the elements of a to this, unless it encounters a null element, at which
                  point it throws a NullPointerException
```

The specification should also be weak enough. Consider this specification for a method that opens a file:

```
static File open (String filename)
    effects      opens a file named filename
```

This is a bad specification. It lacks important details: is the file opened for reading or writing? Does it already exist or is it created? And it is also too strong, since there is no way it can truly guarantee to open a file. The process in which it runs may lack permission to open a file, or there might be some problem with the file system beyond the control of the program. Instead, the specification should say something much weaker: that it attempts to open a file, and if it succeeds, the file has certain properties.

10 Conclusion

A specification acts as a crucial firewall between the implementor of a method and its client. It makes separate development possible: the client is free to write code that uses the method without seeing its source code, and the implementor is free to write the code that implements the procedure without knowing how it will be used. Declarative specifications are the most useful in practice. Preconditions make life hard for the client, but, applied judiciously, are a vital tool in the software designer's repertoire.