Lecture 4: Procedure Specifications

4.1. 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 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. Vector, 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.

This lecture is related to our discussion of decoupling and dependences in the last two lectures. There, we were concerned only with whether a dependence existed. Here, we are investigating the question of what form the dependence should take. By exposing only the specification of a procedure, its clients are less dependent on it, and therefore less likely to need changing when the procedure changes.

4.2. Behavioral Equivalence

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

```
static int findA (int [] a, int val) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == val) return i;
        }
    return a.length;
    }
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 code is different, so in that sense they are different. Our 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 missing, *findA* returns the length and *findB* returns -1;
- when *val* appears twice, *findA* returns the lower index and *findB* returns the higher.

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 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.

In this case, our specification might be

requires: *val occurs in a*
effects: *returns result such that a[result] = val*

4.3. Specification Structure

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*.

We'll explain each of these in turn. For each, we'll explain what the clause means, and what a missing clause implies. Later, we'll look at some convenient shorthands that allow particular common idioms to be specified as special kinds of clause.

The precondition is an obligation on the client (ie, the caller of the method). It's 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 reachable objects may or may not change. But usually only some small part of the state is modifed. The frame condition identifies which objects may be modified. If we say *modifies x*, this means that the object *x*, which is presumed to be mutable, may be modified, 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, the method is said to be *partial*, since it only works on 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.

4.4. Declarative Specification

Roughly speaking, there are two kinds of specifications. *Operational* specifications give a series of steps that the method performs; pseudocode descriptions are operational. *Declarative* specifications don't give details of intermediate steps. Instead, they just give properties of the final outcome, and how it's related to the initial state.

Almost always, declarative specifications are preferable. They're usually shorter, easier to understand, and most importantly, they don't expose implementation details inadvertently that a client may 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 spec that the method 'goes down the array until it finds val', since aside from being rather vague, this spec 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 example of declarative specification. The class *StringBuffer* provides objects that are like *String* objects but mutable. The methods of StringBuffer modify the object rather than creating new ones: they are *mutators*, whereas String's methods are *producers*. The *reverse* method reverses a string. Here's how it's specified in the Java API:

```
public StringBuffer reverse()
// modifies: this
// effects: Let n be the length of the old character sequence, the one contained in the
string buffer
//      just prior to execution of the reverse method. Then the character at index k in
the new
//      character sequence is equal to the character at index n-k-1 in the old character
sequence.
```

Note that the postcondition gives no hint of how the reversing is done; it simply gives a property that relates the character sequence before and after. (We've omitted part of the specification, by the way: the return value is simply the string buffer object itself.) A bit more formally, we might write

```
effects:
   length (this.seq) = length (this.seq')
   all k: 0..length(this.seq)-1  | this.seq'[k] = this.seq[length(this.seq)-k-1]
```

Here I've used the notation this.seq' to mean the value of the character sequence in this object after execution. The course text uses the keyword *post* as a subscript for the same purpose. There's no precondition, so the method must work when the string buffer is empty too; in this case, it will actually leave the buffer unchanged.

Another example, this time from *String.* The *startsWith* method tests whether a string starts with a particular substring.

```
public boolean startsWith(String prefix)
// Tests if this string starts with the specified prefix.
// effects:
//  if (prefix = null) throws NullPointerException
//  else returns true iff exists a sequence s such that (prefix.seq ^ s = this.seq)
```

I've assumed that String objects, like StringBuffer objects, have a specification field that models the sequence of characters. The caret is the concatenation operator, so 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. 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
//      some sequence s | length(s) = i && 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.

Our final example shows how a declarative specification can express what is often called non- determinism, but is better called 'under-determinedness'. By not giving enough details to allow the client to infer the behavior in all cases, the specification makes implementation easier. The term non-determinism suggests that the implementation should exhibit all possible behaviors that satisfy the specification, which is not the case.

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.

Here is an example of a truly underdetermined specification. In the *Observer* pattern, a set of obejects known as 'observers' are informed of changes to an object known as a 'subject'. The subject will belong to a class that subclasses *java.util.Observable*. In the specification of *Observable*, there is a specification field *observers* that holds the set of observer objects. This class provides methods to add an observer

```
public void addObserver(Observer o)
// modifies: this
// effects: this.observers' = this.observers + {o}
```

(using + to mean set union), and to notify the observers of a change in state:

```
public void notifyObservers()
// modifies the objects in this.observers
// effects: calls o.notify on each observer o in this.observers
```

The specification of notify does not indicate in what order the observers are notified. What order is chosen may have an effect on overall program behavior, but having chosen to model the observers as a set, there is no way to specify an order anyway.

4.5. Exceptions and Preconditions

An obvious design issue is whether to use a precondition, and if so, whether it should be checked. It is crucial to understand that a precondition does not require that checking be performed. On the contrary, the most common use of preconditions is to demand a property precisely because it would be hard or expensive to check.

As mentioned above, a non-trivial precondition renders the method partial. This inconveniences clients, because they have to ensure that they don't call the method in a bad state (that violates the precondition); if they do, there is no predictable way to recover from the error. So users of methods don't like preconditions, and for this reason the methods of a library will usually be total. That's why the Java API classes, for example, invariably throw exceptions when arguments are inappropriate. It makes the programs in which they are used more robust.

Sometimes though, a precondition allows you to write more efficient code and saves trouble. For example, in an implementation of a binary tree, you might have a private method that balances the tree. Should it handle the case in which the ordering invariant of the tree does not hold? Obviously not, since that would be expensive to check. Inside the class that implements the tree, it's reasonable to assume that the invariant holds. We'll generalize this notion when we talk about *representation invariants* in a forthcoming lecture.

The decision of whether to use a precondition is an engineering judgment. The key factors are the cost of the check (in writing and executing code), and the scope of the method. If it's only called locally in a class, the precondition can be discharged by carefully checking all the sites that call the method. But if the method is public, and used by other developers, it woul d be less wise to use a precondition.

Sometimes, it's not feasible to check a condition without making a method unacceptably slow, and a precondition is often necessary in this case. In the Java standard library, for example, the binary search methods of the *Arrays* class require that the array given be sorted. To check that the array is sorted would defeat the entire purpose of the binary search: to obtain a result in logarithmic and not linear time.

Even if you decide to use a precondition, it may be possible to insert useful checks that will detect, at least sometimes, that the precondition was violated. These are the runtime assertions that we discussed in our lecture on exceptions. Often you won't check the precondition explicitly at the start, but you'll discover the error during computation. For example, in balancing the binary tree, you might check when you visit a node that its children are appropriately ordered.

If a precondition is found to be violated, you should throw an *unchecked* exception, since the client will not be expected to handle it. The throwing of the exception will not be mentioned in the specification, although it can appear in implementation notes below it.

4.6. Shorthands

There are some convenient shorthands that make it easier to write specifications. When a method does not modify anything, we specify the return value in a *returns* clause. If an exception is thrown, the condition and the exception are given in a *throws* clause. For example, instead of

```
public boolean startsWith(String prefix)
// effects:
//  if (prefix = null) throws NullPointerException
//  else returns true iff exists a sequence s such that (prefix.seq ^ s = this.seq)
```

we can write

```
public boolean startsWith(String prefix)
// throws: NullPointerException if (prefix = null)
// returns: true iff exists a sequence s such that (prefix.seq ^ s = this.seq)
```

The use of these shorthands implies that no modifications occur. There is an implicit ordering in which conditions are evaluated: any throws clauses are considered in the order in which they appear, and then returns clauses. This allows us to omit the else part of the if-then-else statement.

Our 6170 JavaDoc html generator produces specifications formatted in the Java API style. It allows the clauses that we have discussed here, and which have been standard in the specification community for several decades, in addition to the shorthand clauses. We won't use the JavaDoc *parameters* clause: it is subsumed by the postcondition, and is often cumbersome to write.

4.7. Specification Ordering

Suppose you want to substitute one method for another. How do you compare the specifications?

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 him. You can always strengthen the postcondition, which means making more promises. For example, our method *maybePrime* 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).

4.8. 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:

- The specification should be *coherent*: it shouldn't have lots of different cases. Deeply nested if- statements are a sign of trouble, as are boolean flags presented as arguments.
- The results of a call should be *informative*. Java's HashMap class has a put method that takes a key and a value and returns a previous value if that key was already mapped, or null otherwise. HashMaps allow null references to be stored, so a null result is hard to interpret.
- The specification should be *strong enough*. There's no point throwing a checked exception for a bad argument but allowing arbitrary mutations, because a client won't be able to determine what mutations have actually been made.
- The specification should *be weak enough*. A method that takes a URL and returns a network connection clearly cannot promise always to succeed.

4.9. Summary

A specification acts as a crucial firewall between the implementor of a procedure and its client. It makes separate development possible: the client is free to write code that uses the procedure 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.