L7: Specifications Part 2

Today
- Deterministic vs. underdetermined
- Declarative vs. operational
- Strong vs. weak

Introduction

In this lecture, Part 2 of the lecture about specifications, we’ll look at different specs for similar behaviors, and talk about the tradeoffs between them. We’ll look at three dimensions for comparing specs:

- How deterministic it is. Does the spec define only a single possible output for a given input, or allow the implementor to choose from a set of legal outputs?
- How declarative it is. Does the spec just characterize what the output should be, or does it explicitly say how to compute the output?
- How strong it is. Does the spec have a small set of legal implementations, or a large set?

Deterministic vs. Underdetermined Specs

Recall our two example implementations of find from the previous lecture:

```java
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;
}
```

Here is one possible specification of find:

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

This specification is deterministic: when presented with a state satisfying the precondition, the outcome is determined. Both findA and findB satisfy the specification, so if this is the specification on which the clients relied, the two implementations are equivalent and substitutable for one another. (Of course a procedure 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:

\[
\begin{align*}
\text{static int find (int} & \text{[]} \text{a, int val)} \\
\text{requires: val occurs in a} \\
\text{effects: returns index } & \text{i such that } a[i] = \text{val}
\end{align*}
\]

This specification is not deterministic. 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 'resolving' the underdeterminedness in its own way. A client of find can't 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 procedure 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:

\[
\begin{align*}
\text{static int find (int} & \text{[]} \text{a, int val)} \\
\text{effects: returns largest index } & \text{i such that } a[i] = \text{val, or -1 if no such i}
\end{align*}
\]

It is satisfied by findB but not findA.

**Declarative vs. Operation Specs**

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.

One reason programmers sometimes lapse into operational specifications is because they’re using the spec comment to explain the implementation for a maintainer. Don’t. Do that using comments within the body of the method, not in the spec comment.
Stronger vs. Weaker Specs

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

A specification \( A \) is stronger than or equal to a specification \( B \) if

\[ \cdot \text{A's precondition is weaker than or equal to B's} \]
\[ \cdot \text{A's postcondition is stronger than or equal to B's, for the states that satisfy B's precondition.} \]

If this is the case, then an implementation that satisfies \( A \) can be used to satisfy \( B \) as well.

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, this spec for find:

\[
\begin{align*}
\text{static int find1 (int } &\text{[] } a, \text{ int val)} \\
\text{requires: val occurs exactly once in } &a \\
\text{effects: returns index } i \text{ such that } a[i] = \text{val}
\end{align*}
\]

can be replaced in any context by

\[
\begin{align*}
\text{static int findStronger2 (int } &\text{[] } a, \text{ int val)} \\
\text{requires: val occurs at least once in } &a \\
\text{effects: returns index } i \text{ such that } a[i] = \text{val}
\end{align*}
\]

which has a weaker precondition. This in turn can be replaced by

\[
\begin{align*}
\text{static int findStronger3 (int } &\text{[] } a, \text{ int val)} \\
\text{requires: val occurs at least once in } &a \\
\text{effects: returns lowest index } i \text{ such that } a[i] = \text{val}
\end{align*}
\]

which has a stronger precondition.

What about this method:

\[
\begin{align*}
\text{static int find4 (int } &\text{[] } a, \text{ int val)} \\
\text{requires: nothing} \\
\text{effects: returns index } i \text{ such that } a[i] = \text{val, or -1 if no such } i
\end{align*}
\]

Is it stronger or weaker than the other finds?

Designing Good Specifications

What makes a good method? Designing a method means primarily writing a specification.

About the form of the specification: it should obviously be succinct, clear and well-structured, so that it’s easy to read.

The content of the specification, however, is harder to prescribe. There are no infallible rules, but there are some useful guidelines.

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:

\[
\begin{align*}
\text{static int minFind (int } &\text{[]} a, \text{ int } b, \text{ int val)} \\
\text{effects: returns smallest index in arrays } &a \text{ and } b \text{ at which val appears}
\end{align*}
\]
Is this a well-designed procedure? 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 procedures.

The results of a call should be informative. Consider the specification of a method that puts a value in a map:

```java
static V put (Map<K,V> map, K key, V 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 map can store nulls. 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 for sure that you didn't insert nulls.

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. Here's a specification illustrating this flaw (and also written in an inappropriately operational style):

```java
static void addAll (List<T> list1, List<T> list2)
    effects: adds the elements of list2 to list1,
              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:

```java
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's too strong, since there's no way it can 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.

The specification should use abstract types where possible, giving more freedom to both the client and the implementor. Consider this specification:

```java
static ArrayList<T> reverse(ArrayList<T> list)
    effects: returns a new list which is the reversal of list, i.e. newList[i] == list[n-i-1]
             for all 0 <= i < n, where n = list.size()
```

This forces the client to pass in an ArrayList, and forces the implementor to return an ArrayList, even if there might be alternative List implementations that they would rather use. Since the behavior of the specification doesn't depend on anything specific about ArrayList, it would be better to write this spec in terms of the more abstract List<T>.

**Precondition or Postcondition?**

An obvious design issue is whether to use a precondition, and if so, whether the method code should attempt to make sure the precondition has been met before proceeding. In fact, the most common
use of preconditions is to demand a property precisely because it would be hard or expensive for the method to check it.

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 specify (as a postcondition) that they throw unchecked exceptions when arguments are inappropriate. This approach makes it easier to find the bug or incorrect assumption in the caller code that led to passing bad arguments. In general, it’s better to fail fast, as close as possible to the site of the bug, rather than let bad values propagate through a program far from their original cause.

Sometimes, it’s not feasible to check a condition without making a method unacceptably slow, and a precondition is often necessary in this case. If we wanted to implement the find() method using binary search, we would have to require that the array be sorted. Forcing the method to actually 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.

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 would be less wise to use a precondition. Instead, like the Java API classes, you should throw an exception.

A word about access control

We have been using public for almost all of our methods, without really thinking about it. The decision to make a method public or private is actually a decision about the contract of the class. Public methods are freely accessible to other parts of the program. Making a method public advertises it as a service that your class is willing to provide. If you make all your methods public – including helper methods that are really meant only for local use within the class – then other parts of the program may come to depend on them, which will make it harder for you to change the internal implementation of the class in the future. Your code won’t be as ready for change.

Making internal helper methods public will also add clutter to the visible interface your class offers. Keeping internal things private makes your class’s public interface smaller and more coherent (meaning that it does one thing and does it well). Your code will be easier to understand.

We will see even stronger reasons to use private in the next few lectures, when we start to write classes with persistent internal state. Protecting this state will help keep the program safe from bugs.

An example of specification design

Let’s finish by working an example in which we design some specifications. Suppose we’re given this problem:

Given enough time, a hypothetical monkey typing at random would, as part of its output, almost surely produce all of Shakespeare's plays. Write a program that simulates a monkey typing words randomly, and each time the monkey types a word that appears somewhere in Shakespeare, print it.
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.

Let’s review how specifications help with the main goals of this course:

- **Safe from bugs.** A good specification clearly documents the mutual assumptions that a client and implementor are relying on. Bugs often come from disagreements at the interfaces, and the presence of a specification reduces that. Using machine-checked language features in your spec, like static typing and exceptions rather than just a human-readable comment, can reduce bugs still more.

- **Easy to understand.** A short, simple spec is easier to understand than the implementation itself, and saves other people from having to read the code.

- **Ready for change.** Specs establish contracts between different parts of your code, allowing those parts to change independently as long as they continue to satisfy the requirements of the contract.