

# **Lecture Notes on Software Engineering**

Daniel Jackson  
Fall 2002



## Contents

<b>Lecture 1: Introduction</b>	<b>5</b>
1.1 What 6170's About	5
1.2 Admin & Policies	5
1.3 Why Software Engineering Matters	6
1.3.1 Development failures	7
1.3.2 Accidents	7
1.3.3 Software Quality	9
1.4 Why Design Matters	9
1.4.1 The Netscape Story	11
1.5 Advice	12
1.6 Parting Shots	13
<b>Lecture 2: Object Semantics</b>	<b>15</b>
2.1 Variables, References and Objects	15
2.2 Aliasing, Mutability and Reference Equality	17
2.3 Null References	20
2.4 User-defined Classes & Fields	21
2.5 User-defined Constructors	22
2.6 Method Call	23
2.7 Conclusion	26
<b>Lecture 3: Subclassing and Dynamic Dispatch</b>	<b>27</b>
3.1 Recap	27
3.2 Extending a Class by Inheritance	28
3.3 A Template Method	30
3.4 Downcasting	31
3.5 Downcasts are not Typecasts	33
3.6 Type Hierarchy and Safety	34
3.7 Conclusion	37
<b>Lecture 4: Object Models</b>	<b>39</b>
4.1 Object Diagrams	39
4.2 Object Model Basics	42
4.3 Multiplicity	45
4.4 Mutability	46
4.5 Subclassing in the Object Model	47
4.6 Conclusion	48

<b>Lecture 5: Procedure Specifications</b>	<b>49</b>
5.1 Contracts, Firewalls and Decoupling	49
5.2 Behavioural Equivalence	51
5.3 Specification Structure	52
5.4 Find Revisited	53
5.5 A Mutating Specification	54
5.6 Declarative Specification	54
5.7 Checking Preconditions	57
5.8 Shorthands	58
5.9 Specification Ordering	59
5.10 Judging Specifications	60
5.11 Summary	61

## Lecture 5: Procedure Specifications

In this lecture, we'll look at specifications of methods: what role they play, and how they are structured and interpreted.

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 behaviour 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.

### 5.1 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 contract imposes obligations on both the *client* (or user) of the unit specified, and on the implementor of the unit. The contract is understood as an implication:

5.1.1      client-meets-obligation => implementor-meets-obligation

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 lightbulbs, 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 procedure if you have its specification. And it shields the implementor from the details of the *usage* of the unit; he 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 obeying its obligation.

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.

## 5.2 Behavioural Equivalence

A fundamental benefit of specifications is that they allow us to determine what kinds of changes can be made to a unit without affecting its client – when one unit can be exchanged for another. To see why specifications help in this respect, consider two implementations of a procedure to find the index of an element in an array:

```
5.2.1 static int findA (int [] a, int val) {
5.2.2     for (int i = 0; i < a.length; i++) {
5.2.3         if (a[i] == val) return i;
5.2.4     }
5.2.5     return a.length;
5.2.6 }
5.2.7
5.2.8 static int findB (int [] a, int val) {
5.2.9     for (int i = a.length - 1 ; i >= 0; i-) {
5.2.10        if (a[i] == val) return i;
5.2.11    }
5.2.12    return -1;
5.2.13 }
```

Are these equivalent? 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 behaviour:

- when *val* is missing, *findA* returns the length and *findB* returns -1;
- when *val* appears more than once, *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 behaviour 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. Of course, we could

just look at the client as well, but if there is more than one client, this would be very tedious: the specification summarizes the assumptions of *all* clients.

### 5.3 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 modified. 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 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.

## 5.4 Find Revisited

Here is one possible specification of find:

```
5.4.1    static int find (int [] a, int val)
5.4.2        requires: val occurs exactly once in a
5.4.3        effects: returns result such that a[result] = 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 are equivalent and substitutable for one another. (Of course a procedure must have the *name* demanded by the specification; here I'm 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

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

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 behaviour 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 behaviour 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

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

5.4.8        effects: returns largest result such that  $a[\text{result}] = \text{val}$ , or -1 if no such result

It is satisfied by findB but not findA.

## 5.5 A Mutating Specification

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:

```
5.5.1        class Vector {
5.5.2        ...
5.5.3        boolean addAll (Vector v)
5.5.4            requires:  $v \neq \text{null}$  and  $v \neq \text{this}$ 
5.5.5            modifies: this
5.5.6            effects:
5.5.7                adds the elements of v to the end of this
5.5.8                returns true if this changed as a result of the call
```

I've taken this, slightly simplified, from the Java Vector class. First, look at the frame condition: it tells us that only this is modified, so in particular the argument vector v is not mutated – likely to be a crucial property for most clients. 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 behaviour of the method is not constrained if you call it with a null argument, or if you attempt to add the elements of a vector 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 v and add it to this, then go on to the next element of v until you get to the end. If v and this are the same vector, this algorithm will not terminate – an outcome permitted by the specification.

## 5.6 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. A good way to think about it is that while an operational specification asks the question "what happens

in a good execution?”, a declarative specification asks “how would you know if a good execution had happened?”.

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.

We’ve seen examples of declarative specifications above. Here are some more. 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 the methods of `String` are *producers*. The `reverse` method reverses a string. Here’s how it’s specified in the Java API:

```
5.6.1   StringBuffer reverse()
5.6.2   modifies: this
5.6.3   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 the postcondition like this:

```
5.6.4   effects:
5.6.5       length (this_pre.seq) = length (this.seq)
5.6.6       all k: 0..length(this_pre.seq)-1 |
5.6.7           this.seq[k] = this_pre.seq[length(this_pre.seq)-k-1]
```

Here I’ve used the notation `x_pre` to mean the value of `x` in the pre-state, before execution, and `x` itself to mean the value in the post-state. An alternative style is to write `x_post` or `x'` to mean the value after; this is what our course text does. The reason to prefer marking the pre-state value is that a postcondition that mentions only post-state values is then an *assertion* about the state after execution, and it may be possible to execute it directly in the Java code as a check.

The value of the buffer can be thought of as a sequence of characters. Since the ‘value’ of any object in Java is really given by the values of its fields, I’ve invented a field name

seqs that gives this sequence for a given buffer. Such a field is called an *abstract field* or *specification field*, and it's a convenient way to talk about an object when we don't want to talk about the fields that are actually used in its implementation. So `this.seq`, for example, means the sequence of characters of this string buffer in the post-state.

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:

```
5.6.8    public boolean startsWith(String prefix)
5.6.9      effects:
5.6.10     if (prefix == null) throws NullPointerException
5.6.11     else returns true iff
5.6.12     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`:

```
5.6.13    public String substring(int i)
5.6.14      effects:
5.6.15     if i < 0 or i >= length (this.seq) throws IndexOutOfBoundsException
5.6.16     else returns r such that
5.6.17     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 examples of this section illustrate 'under-determinedness' more. By not giving enough details to allow the client to infer the behaviour in all cases, the specification makes implementation easier. 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:

```
5.6.18    public boolean maybePrime ()
5.6.19      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 explains, 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.

In the Observer pattern, a set of objects 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:

```
5.6.20 public void addObserver(Observer o)
5.6.21     modifies: this
5.6.22     effects: this.observers = this.observers_pre + {o}
```

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

```
5.6.23 public void notifyObservers()
5.6.24     modifies: the objects in this.observers
5.6.25     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. Having chosen to model the observers as a set, there is no way to specify an order anyway. What order is chosen may indeed have an effect on overall program behaviour, but since the specification guarantees no ordering, a program that uses this mechanism should not rely on one, and should work whatever ordering is used.

This specification, although we've used the standard keywords, is actually not a declarative specification at all. The effects clause doesn't describe a relationship between pre- and post-states; instead, it says what happens when the method is executed. In this case, this is the best we can do, since we don't know what effect `notify` will have: it's not even a method of this class.

## 5.7 Checking Preconditions

An basic 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 next week.

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.

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 will discuss later in the course. 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.

## 5.8 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

```
5.8.1 public boolean startsWith(String prefix)
5.8.2     effects:
```

```

5.8.3      if (prefix = null) throws NullPointerException
5.8.4      else returns true iff
5.8.5      exists a sequence s such that (prefix.seq ^ s = this.seq)

```

we can write

```

5.8.6      public boolean startsWith(String prefix)
5.8.7      throws: NullPointerException if (prefix = null)
5.8.8      returns: true iff
5.8.9      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.

## 5.9 Specification Ordering

Suppose we have a client that uses a module on the assumption that it satisfies one specification, and we have a candidate module which satisfies another, different specification. Will the module work?

This question is about *ordering* of specifications. If we can define an ordering amongst specifications, then we can answer the question simply by comparing the two 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).

It is instructive to think about the extreme cases. What does a precondition of true mean? It's the weakest obligation on the client, and is therefore, according to our first rule, always an acceptable replacement for another precondition. A precondition of false, on the other hand, is the strongest: a specification with such a precondition is an acceptable replacement only for another specification with a false precondition, and a procedure satisfying such a specification is useless, since there is no state in which it can be legally invoked. A postcondition of true is the weakest obligation on the implementor; it says that the procedure can do anything. So it can only replace another true postcondition, and a procedure can satisfy it by doing anything (in particular, nothing!) except for not terminating. Such a procedure is not of much use to a client. A postcondition of false places the strongest obligation on the implementor. In fact, it's so strong, there's no real procedure that can satisfy it. For this reason, in the theory of specifications, it's called *miracle*. This pathological case is a bit odd, and we actually rule it out in our specifications by requiring that they are always *implementable*, by requiring that for every pre-state satisfying the precondition, there is at least one post-state satisfying the postcondition.

## 5.10 Judging Specifications

What makes a good procedure specification? There are no infallible rules, but we can give some 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:

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

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 the put method from Java's HashMap class:

```
5.10.3    Object put (Object key, Object val)
5.10.4    effects
5.10.5    inserts (key,val) into the mapping,
5.10.6    overriding any existing mapping for key
```

```
5.10.7     returns
5.10.8     old value for key,
5.10.9     unless none, in which case it returns null
```

Note that the precondition does not rule out null values, so the hash 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 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):

```
5.10.10    void addAll (Vector v)
5.10.11    effects adds the elements of v to this,
5.10.12    unless it encounters a null element,
5.10.13    at which point it throws a NullPointerException
```

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

```
5.10.14    static File open (String filename)
5.10.15    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.

## 5.11 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.