6.005 Software Construction
Fall 2011
Prof. Rob Miller

L3: Specifications

Today

- Preconditions & postconditions
- Exceptions

Required reading (from the Java Tutorial)

- Exceptions
- Packages
- Controlling Access

Introduction

In this lecture, we'll look at the role played by specifications of methods. Specifications are the linchpin of teamwork. It's impossible to delegate responsibility for implementing a method without a specification. The specification acts as a contract: the implementer 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 pre-condition, the client has responsibilities too.

Why Specifications?

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 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 implementer of a method because they give the 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.

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

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors.

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.

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
```

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*.

The precondition is an obligation on the client (i.e., 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 implementer 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.

If you omit the frame condition, the default is *modifies nothing*. In other words, the method makes no changes to any object.

Specifications in Java

Some languages (notably Eiffel) incorporate preconditions and postconditions as a fundamental part of the language, as expressions that the runtime system (or possibly even the compiler) can automatically check to enforce the contracts between clients and implementers.

Java does not go quite so far, but its static type declarations *are* effectively part of the precondition and postcondition of a method, a part that is automatically checked and enforced by the compiler. The rest of the contract – the parts that we can't write as types – must be described in a comment preceding the method, and generally depends on human beings to check it and guarantee it.

Java has a convention for documentation comments, in which parameters are described by @param clauses and results are described by @return and @throws clauses. You should generally put the preconditions into @param where possible, and postconditions into @return and @throws. So a specification like this:

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

might be rendered in Java like this:

```
/**
 * Find value in an array.
 * @param a array to search; requires that val occurs exactly once in a.
 * @param val value to search for
 * @return index i such that a[i] = val
 */
static int find (int [] a, int val)
```

A frame condition (*modifies* clause) should be described in the @param clauses that it might modify. Side-effects

Find Revisited

Roughly speaking, there are two kinds of specifications. Here is one possible specification of find:

```
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 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 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. 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

```
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.

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:

```
static boolean addAll (List l1, List l2)
      requires: l1 != l2, l1!=null, l2!=null
      modifies: this
      effects: adds the elements of l2 to the end of l1,
                and returns true if l1 changed as a
             result of call
```

We've taken this, slightly simplified, from the Java List interface. First, look at the frame condition: it tells us that only this is modified, so in particular the argument *l2* 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 behavior of the method is not constrained if you call it with a null argument, or if you attempt to add the elements of a list to itself. The constraint that arguments not be null is often left implicit in practice, but 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 *l2* and add it to *l1*, then go on to the next element of *l1* until you get to the end. If *l1* and *l2* are the same list, this algorithm will not terminate -- an outcome permitted by the specification.

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.

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.

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 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:

```
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 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):

```
static void addAll (List<T> l1, List<T> l2)
      effects: adds the elements of l2 to l1,
                unless it encounters a null element,
                at which point it throws a NullPointerException
```

The specification should 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'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.

Exceptions for Special Results

You've probably already seen some exceptions in your Java programming so far, such as ArrayOutOfBoundsException (thrown when an array index a[i] is outside the valid range for the array) or NullPointerException (thrown when trying to call a method on a null object reference).

But exceptions are not just for handling failures like these. They can be used to improve the structure of code that involves procedures with special results.

A standard way to handle special results is to return special values. Lookup operations in the Java library are often designed like this: you get an index of -1 when expecting a positive integer, or a null reference when expecting an object. This approach is OK if used sparingly. It has two problems though. First, it's tedious to check the return value. Second, it's easy to forget to do it. (We'll see that with exceptions you can get help from the compiler in this.)

Also, its not easy to find a 'special value'. Suppose we have a `BirthdayBook` class with a lookup method. Here's one possible method signature:

```
Calendar lookup (String name)
```

What should the method do if the birthday book doesn't have an entry for the person whose name is given? Well, we could return some special date that is not going to be used as a real date. Bad programmers have been doing this for decades; they would return 9/9/99, for example, since it was *obvious* that no program written in 1960 would still be running at the end of the century.

Here's a better approach. The method throws an exception:

```
Calendar lookup (String name) throws NotFoundException {
  ...
  if // not found
    throw new NotFoundException ();
  ...
```

and the caller handles the exception with a catch clause. Now there's no need for any special value, nor the checking associated with it.

Abuse of Exceptions

Here's an example from *Effective Java* by Joshua Bloch (Item 39).

```
try {
   int i = 0;
   while (true)
       a[i++].f();
} catch(ArrayIndexOutofBoundsException e) { }
```

What does this code do? It is not at all obvious from inspection, and that's reason enough not to use it. The infinite loop terminates by throwing, catching and ignoring an `ArrayIndexOutofBoundsException` when it attempts to access the first array element outside the bounds of the array. It is supposed to be equivalent to:

```
for(int i = 0; i < a.length; i++)
   a[i].f();
```

The exception-based idiom is a misguided attempt to improve performance based on the faulty reasoning that, since the VM checks the bounds of array accesses, the normal loop termination test (`i < a.length`) is redundant and should be avoided. However, because exceptions in Java are designed for use only under exceptional circumstances, few, if any, JVM implementations attempt to optimize their performance. On a typical machine, the exception-based idiom runs 70 times slower than the standard one when looping from 0 to 99.

The exception-based idiom is also not guaranteed to work. Suppose the computation of `f()` in the body of the loop contains a bug that results in an out-of-bounds access to some unrelated array. If a reasonable loop idiom were used, the bug would generate an uncaught exception, resulting in immediate thread termination with an appropriate error message. If the exception-based idiom were used, the bug-related exception would be caught and misinterpreted as a normal loop termination.

Checked and Unchecked Exceptions

We've seen two different purposes for exceptions: failures and special results. Java provides two different kinds of exception for these two purposes. They behave the same at runtime; the only difference is what kind of checking the compiler provides.

If a method might throw a *checked* exception, the possibility must be declared in its signature. `NotFoundException` would be a checked exception, and that's why the signature ends `throws NotFoundException`. If a method calls another method that may throw a checked exception, it must either handle it, or declare the exception itself (since if it isn't caught locally it will be propagated).

So if you call the `lookup` method and forget to handle the exception, the compiler will reject your code. This is very useful, because it ensures that exceptions that are expected to occur should be handled. On the other hand, exceptions that correspond to failures are not expected to be handled except at the top level, and for reasons of modularity, we wouldn't want to declare the possibility of failure at every level.

For an *unchecked* exception, in contrast, the compiler will not check for try-catch or a throws declaration. Java still allows you write a throws clause as part of a signature for an unchecked exception, but this has no effect (and is thus a bit funny, and I don't recommend doing it).

All errors and exceptions may have a message associated with them. If not provided in the constructor, the reference to the message string is null.

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, 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.

Design Considerations

The rule we have given -- use checked exceptions for special results, and unchecked exceptions to signal failures -- makes sense, but it isn't the end of the story. The snag is that exceptions in Java aren't as lightweight as they might be.

Aside from the performance penalty, exceptions in Java incur another (more serious) cost: they're a pain to use. If you design a method to have its own exception, you have to create a new class for the exception. If you call a method that can throw a checked exception, you have to wrap it in a `try-catch` statement (even if you know the exception will never be thrown). This latter stipulation creates a dilemma. Suppose, for example, you're designing a queue abstraction. Should popping the queue throw a checked exception when the queue is empty? Suppose you want to support a style or programming in the client in which the queue is popped (in a loop say) until the exception is thrown. So you choose a checked exception. Now some client wants to use the method in a context in which, immediately prior to popping, the client tests whether the queue is empty and only pops if it isn't. Maddeningly, that client will still need to wrap the call in a `try-catch` statement.

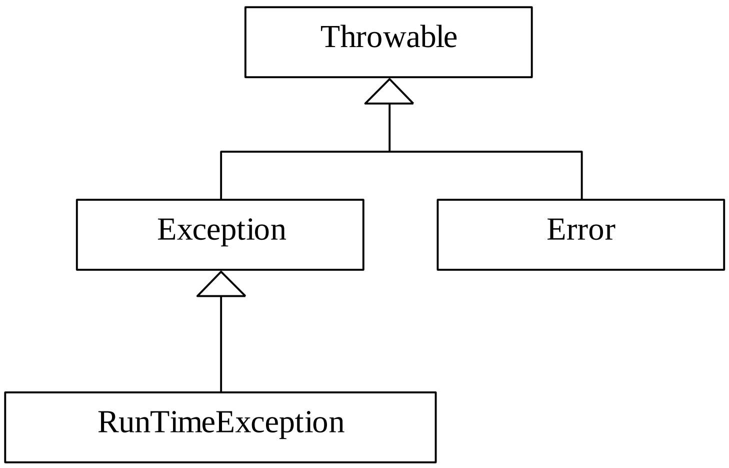
This suggests a more refined rule:

- You should use an unchecked exception only if you expect that clients will usually write code that ensures the exception will not happen, because there is a convenient and inexpensive way to avoid the exception, or because the exception reflects unexpected failures;
- Otherwise you should use a checked exception.

The cost of using exceptions in Java is one reason that many Java API's use the null reference as a special value. It's not a terrible thing to do, so long as it's done judiciously, and carefully specified.

Throwable Hierarchy

Let's look at the class hierarchy for Java exceptions.



`Throwable` is the class of objects that can be thrown or caught. Any object used in a `throw` or `catch` statement, or declared in the `throws` clause of a method, must be a subclass of `Throwable`. `Throwable`'s implementation records a stack trace at the point where the exception was thrown, along with an optional string describing the exception.

`Error` is a subclass of `Throwable` that is "reserved" for errors produced by the Java runtime system, such as `StackOverflowError` and `OutofMemoryError`. For some reason `AssertionError` also extends `Error`, even though it indicates a failure in user code, not in Java runtime. Errors should be considered recoverable, and are generally not caught. All the unchecked throwables you implement should subclass `RunTimeException` (directly or indirectly).

The classes Error and RuntimeException correspond to unchecked exceptions. All other Throwables and Exceptions are checked. Do not define a throwable that is not a subclass of Exception, RunTimeException, or Error.

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 strong 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 spec to implement:

```
/**
 * Find the most common word in a string of text.
 * @param s string of words
 * @return the word that occurs most often in s
 */
public static String mostCommonWord(String s) { ... }
```

Let's push back and criticize the spec we've been given.

- what exactly is meant by a "word?" the spec should be precise about this; that's not a kind of freedom that's particularly useful for the implementer, and it makes the method hard to use for the client.
- how are words compared, when counting their frequency? the spec should be precise about that too.
- the spec fails to even touch on two important cases. It only discusses the case where there is exactly one most common word, and doesn't put requirements on the client or the implementer to deal with the other cases. That means neither the client or the implementer will do it! And bugs will follow. So we need to think about the "0" case – what if there are no words in the string at all? – and about the "many" case – what if there are ties for most-common word? – and ensure that these cases are addressed either by preconditions or postconditions.

Here's how we might handle both these issues with a stronger postcondition:

```
 /**
 * Find the most common word in a string of text.
 * @param s string of words, where a word is
 *   a nonempty string of characters separated by spaces
 *   or punctuation.
 * @return a word that occurs most often in s (at least as much
 *      as any other word).  Alphabetic case is ignored when comparing
 *      words.
 * @throws NoWordException if s has no words
 */
public static String mostCommonWord(String s)
        throws NoWordsException { ... }
```

We've handled the "many answers" case with wording: the return value will be a word that occurs "at least as much as any other word," leaving the implementer free to decide how to break ties. That freedom will be convenient for us; if we wrote a more precise specification, such as the word that occurs first in s should win in the event of a tie, we might have to do a lot more bookkeeping inside mostCommonWord, remembering where each word was.

To handle the "no answers" case, we've introduced a new kind of exception, NoWordsException. We'll have to declare that exception class in order to use it. Here's the simplest way to do that:

```
/**
 * Exception thrown by mostCommonWord() when it can't find a word.
 */
public static class NoWordsException extends Exception {
}
```

Now we're ready to think about implementing this method. Let's break this problem down into several smaller steps:

1. Split the string into words.
2. Count the number of occurrences of each word.
3. Find the word with the maximum count.

We'll write each of these steps as a private helper method, so we can think about them as individual units (that we could test individually[1]).

Let's start with the method that splits the string into words. First we'll write the *signature* for the method, which has the types of its arguments and return values:

```
private static List<String> splitIntoWords(String s) {...}
```

We might have returned a String[] here instead of a List, but lists are easier to work with. Now we'll write its spec:

```
// Split s into words.
// @param s string to split
// @return list of words found in s, in order of their occurrence
//  (where word is defined by the spec for mostCommonWords).
private static List<String> splitIntoWords(String s) { ... }
```

The next step takes this list of strings and counts its elements. We'll write its spec this way:

```
// Count the number of occurrences of each element in a list.
// @param l list of strings
// @return map m such that m[s] == k if s occurs k times in l, while
//      m[s] == null if s never occurs in l.
private static Map<String, Integer> countOccurrences(List<String> l) {
    throw new RuntimeException("not implemented");
}
```

The return type we chose here is a Map, which is a data structure that maps values of one type (the *keys*, in this case Strings) into values of another type (the map's *values*, here Integers, or effectively

[1] Note that JUnit can't easily test private methods, so to use JUnit for testing these methods, we'd need to work around it.

ints). The spec says that every element of the input list becomes a key in the map, and the value corresponding to it is the number of times it appears in the list.

Note the decision here not even to talk about words, but more generally about "elements of the list." Taking the extra step to think and write more generally is often a good thing, because it makes the code (and its spec) easier to change in the future. If someday we wanted to change the caller of countOccurrences – perhaps to pass in lists of phrases, not just words – then the greater flexibility in the spec will be useful. This should be weighed carefully against other considerations, however, and it's not a good idea to make code more complex just to make it very general.

Now let's look at the combination of splitIntoWords and countOccurrences. We'll probably want to plug them together like this:

```
        List<String> words = splitIntoWords(s);
        Map<String, Integer> freq = countOccurrences(words);
```

But a requirement of mostCommonWord's contract has fallen through the cracks! Who's responsible for the ignore-alphabetic-case obligation? We could insert a new method in between that lower-cases the list of words. Or we could put the onus on either splitIntoWords or countOccurrences to handle that obligation. Let's strengthen the spec of splitIntoWords:

```
    // Split s into words.
    // @param s string to split
    // @return list of words found in s, in order of their occurrence
    //  (where word is defined by the spec for mostCommonWords).
    //  The words are all converted to lowercase.
    private static List<String> splitIntoWords(String s) { ... }
```

The final step of our process needs a method to find the word that has maximum count:

```
    // Find a key with maximum value.
    // @param m frequency counts for strings
    // @return s such that m[s] >= m[t] for all other keys t in the map,
    //     or null if no such s exists
    private static String findMax(Map<String,Integer> m) {...}
```

Note that we have written the postcondition of this method in a declarative way. We also chose to return null rather than throw an exception in the case where the map is empty. Is this a good idea or bad idea?

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.