Reading 6, Part 1: Specifications

Before we dive into the structure and meaning of specifications…

Why specifications?

Many of the nastiest bugs in programs arise because of misunderstandings about behavior at the interface between two pieces of code. 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.

Here's an example of one method from `BigInteger` — a class for representing integers up to arbitrary size without the size limit of primitive `int` — next to its code:

Specification from the API documentation :

add

`public BigInteger add(BigInteger val)`

Returns a BigInteger whose value is `(this + val)`.

Parameters:
`val` - value to be added to this BigInteger.

Returns:
`this + val`

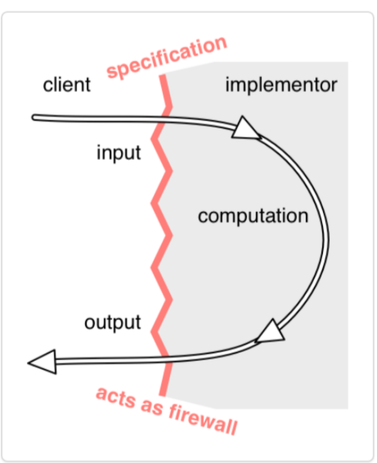
Method body from Java 8 source :

```
if (val.signum == 0)
    return this;
if (signum == 0)
    return val;
if (val.signum == signum)
    return new BigInteger(add(mag, val.mag), signum);

int cmp = compareMagnitude(val);
if (cmp == 0)
    return ZERO;
int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
                   : subtract(val.mag, mag));
resultMag = trustedStripLeadingZeroInts(resultMag);

return new BigInteger(resultMag, cmp == signum ? 1 : -1);
```

The spec for `BigInteger.add` is straightforward for clients to understand, and if we have questions about corner cases, the `BigInteger` class provides additional human-readable documentation. If all we had was the code, we'd have to read through the `BigInteger` constructor, `compareMagnitude`, `subtract`, and `trustedStripLeadingZeroInts` just as a starting point.

Specifications are good for the implementer of a method because they give the implementor freedom to change the implementation without telling clients. Specifications can make code faster, too. We'll see that using a weaker specification can rule out certain states in which a method might be called. This restriction on the inputs might allow the implementor to skip an expensive check that is no longer necessary and use a more efficient implementation.

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.



Behavioral equivalence

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

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

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

Of course the code is different, so in that sense they are different; and we've given them different names, just for the purpose of discussion. To determine *behavioral equivalence*, our question is whether we could substitute one implementation for the other.

Not only do these methods have different code, they actually have different behavior:

- when `val` is missing, `findFirst` returns the length of `arr` and `findLast` returns -1;
- when `val` appears twice, `findFirst` returns the lower index and `findLast` returns the higher.

But when `val` occurs at exactly one index of the array, the two methods behave the same: they both return that index. It may be that clients never rely on the behavior in the other cases. Whenever they call the method, they will be passing in an `arr` with exactly one element `val`. For such clients, these two methods are the same, and we could switch from one implementation to the other without issue.

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:

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

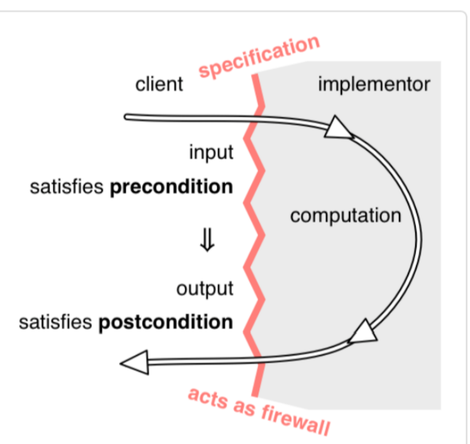
Specification structure

A specification of a method consists of several clauses:

- a *precondition*, indicated by the keyword *requires*
- a *postcondition*, indicated by the keyword *effects*

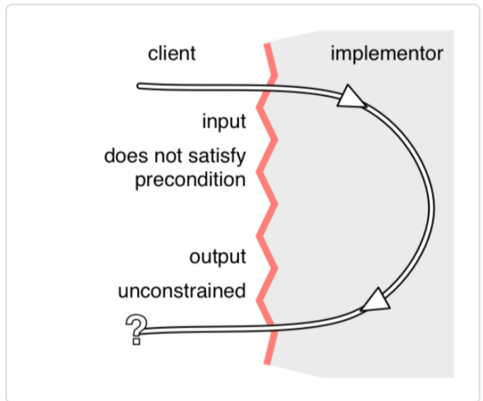
The precondition is an obligation on the client (i.e., the caller of the method). It's a condition over the state in which the method is invoked.

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 overall structure is a logical implication: *if* the precondition holds when the method is called, *then* the postcondition must hold when the method completes.

If the precondition does *not* hold when the method is called, the implementation is *not* bound by the postcondition. It is free to do anything, including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc.



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 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 put the preconditions into `@param` where possible, and postconditions into `@return` and `@throws` . So a specification like this:

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

… might be rendered in Java like this:

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

The Java API documentation is produced from Javadoc comments in the Java standard library source code . Documenting your specifications in Javadoc allows Eclipse to show you (and clients of your code) useful information, and allows you to produce HTML documentation in the same format as the Java API docs.

> Read: *Introduction* , *Commenting in Java* , and *Javadoc Comments* in **Javadoc Comments** .
>
> When writing your specifications, you can also refer to Oracle's **How to Write Doc Comments** .

Null references

In Java, references to objects and arrays can also take on the special value *null* , which means that the reference doesn't point to an object. Null values are an unfortunate hole in Java's type system.

Primitives cannot be null:

```
int size = null;     // illegal
double depth = null; // illegal
```

and the compiler will reject such attempts with static errors.

On the other hand, we can assign null to any non-primitive variable:

```
String name = null;
int[] points = null;
```

and the compiler happily accepts this code at compile time. But you'll get errors at runtime because you can't call any methods or use any fields with one of these references:

```
name.length()   // throws NullPointerException
points.length   // throws NullPointerException
```

Note, in particular, that `null` is not the same as an empty string `""` or an empty array. On an empty string or empty array, you *can* call methods and access fields. The length of an empty array or an empty string is 0. The length of a string variable that points to `null` isn't anything: calling `length()` throws a `NullPointerException` .

Also note that arrays of non-primitives and collections like `List` might be non-null but contain null as a value:

```
String[] names = new String[] { null };
List<Double> sizes = new ArrayList<>();
sizes.add(null);
```

These nulls are likely to cause errors as soon as someone tries to use the contents of the collection.

Null values are troublesome and unsafe, so much so that you're well advised to remove them from your design vocabulary. In 6.005 — and in fact in most good Java programming — **null values are implicitly disallowed in parameters and return values** . So every method implicitly has a precondition on its object and array parameters that they be non-null. Every method that returns an object or an array implicitly has a postcondition that its return value is non-null. If a method allows null values for a parameter, it should explicitly state it, or if it might return a null value as a result, it should explicitly state it. But these are in general not good ideas. **Avoid null** .

There are extensions to Java that allow you to forbid `null` directly in the type declaration, e.g.:

```
static boolean addAll(@NonNull List<T> list1, @NonNull List<T> list2)
```

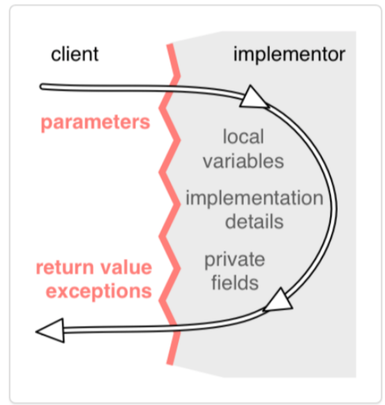
where it can be checked automatically at compile time or runtime.

Google has their own discussion of `null` in Guava, the company's core Java libraries . The project explains:

> Careless use of `null` can cause a staggering variety of bugs. Studying the Google code base, we found that something like 95% of collections weren't supposed to have any null values in them, and having those **fail fast** rather than silently accept `null` would have been helpful to developers.
>
> Additionally, `null` is unpleasantly ambiguous. It's rarely obvious what a `null` return value is supposed to mean — for example, `Map.get(key)` can return `null` either because the value in the map is `null` , or the value is not in the map. Null can mean failure, can mean success, can mean almost anything. Using something other than `null` **makes your meaning clear** .

(Emphasis added.)

What a specification may talk about



A specification of a method can talk about the parameters and return value of the method, but it should never talk about local variables of the method or private fields of the method's class. You should consider the implementation invisible to the reader of the spec.

In Java, the source code of the method is often unavailable to the reader of your spec, because the Javadoc tool extracts the spec comments from your code and renders them as HTML.

Testing and specifications

In testing, we talk about *black box tests* that are chosen with only the specification in mind, and *glass box tests* that are chosen with knowledge of the actual implementation (*Testing*). But it's important to note that **even glass box tests must follow the specification** . Your implementation may provide stronger guarantees than the specification calls for, or it may have specific behavior where the specification is undefined. But your test cases should not count on that behavior. Test cases must obey the contract, just like every other client.

For example, suppose you are testing this specification of `find` , slightly different from the one we've used so far:

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

This spec has a strong precondition in the sense that `val` is required to be found; and it has a fairly weak postcondition in the sense that if `val` appears more than once in the array, this specification says nothing about which particular index of `val` is returned. Even if you implemented `find` so that it always returns the lowest index, your test case can't assume that specific behavior:

```
int[] array = new int[] { 7, 7, 7 };
assertEquals(0, find(array, 7));  // bad test case: violates the spec
assertEquals(7, array[find(array, 7)]);  // correct
```

Similarly, even if you implemented `find` so that it (sensibly) throws an exception when `val` isn't found, instead of returning some arbitrary misleading index, your test case can't assume that behavior, because it can't call `find()` in a way that violates the precondition.

So what does glass box testing mean, if it can't go beyond the spec? It means you are trying to find new test cases that exercise different parts of the implementation, but still checking those test cases in an implementation-independent way.

Testing units

Recall the web search example from *Testing* with these methods:

```
/** @return the contents of the web page downloaded from url */
public static String getWebPage(URL url) { ... }

/** @return the words in string s, in the order they appear,
 *          where a word is a contiguous sequence of
 *          non-whitespace and non-punctuation characters */
public static List<String> extractWords(String s) { ... }

/** @return an index mapping a word to the set of URLs
 *          containing that word, for all webpages in the input set */
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    calls getWebPage and extractWords
    ...
}
```

We talked then about *unit testing* , the idea that we should write tests of each module of our program in isolation. A good unit test is focused on just a single specification. Our tests will nearly always rely on the specs of Java library methods, but a unit test for one method we've written shouldn't fail if a *different* method fails to satisfy its spec. As we saw in the example, a test for `extractWords` shouldn't fail if `getWebPage` doesn't satisfy its postcondition.

Good *integration tests* , tests that use a combination of modules, will make sure that our different methods have compatible specifications: callers and implementors of different methods are passing and returning values as the other expects. Integration tests cannot replace systematically-designed unit tests. From the example, if we only ever test `extractWords` by calling `makeIndex` , we will only test it on a potentially small part of its input space: inputs that are possible outputs of `getWebPage` . In doing so, we've left a place for bugs to hide, ready to jump out when we use `extractWords` for a different purpose elsewhere in our program, or when `getWebPage` starts returning web pages written in a new format, etc.

Specifications for mutating methods

We previously discussed mutable vs. immutable objects, but our specifications of `find` didn't give us the opportunity to illustrate how to describe side-effects — changes to mutable data — in the postcondition.

Here's a specification that describes a method that mutates an object:

```
static boolean addAll(List<T> list1, List<T> list2)
  requires: list1 != list2
  effects:  modifies list1 by adding the elements of list2 to the end of
              it, and returns true if list1 changed as a result of call
```

We've taken this, slightly simplified, from the Java `List` interface. First, look at the postcondition. It gives two constraints: the first telling us how `list1` is modified, and the second telling us how the return value is determined.

Second, look at the precondition. It tells us that the behavior of the method if you attempt to add the elements of a list to itself is undefined. You can easily imagine why the implementor of the method would want to impose this 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 `list2` and add it to `list1` , then go on to the next element of `list2` until you get to the end. If `list1` and `list2` are the same list, this algorithm will not terminate — an outcome permitted by the specification because of its precondition.

Remember also our implicit precondition that `list1` and `list2` must be valid objects, rather than `null` . We'll usually omit saying this because it's virtually always required of object references.

Here is another example of a mutating method:

```
static void sort(List<String> lst)
  requires: nothing
  effects:  puts lst in sorted order, i.e. lst[i] <= lst[j]
              for all 0 <= i < j < lst.size()
```

And an example of a method that does not mutate its argument:

```
static List<String> toLowerCase(List<String> lst)
  requires: nothing
  effects:  returns a new list t where t[i] = lst[i].toLowerCase()
```

Just as we've said that `null` is implicitly disallowed unless stated otherwise, we will also use the convention that **mutation is disallowed unless stated otherwise** . The spec of `toLowerCase` could explicitly state as an *effect* that "lst is not modified", but in the absence of a postcondition describing mutation, we demand no mutation of the inputs.

Next: Exceptions