High Quality Java Code: Contracts & Assertions

by Pedro Agulló Soliveres

Writing correct code is really hard. Maybe the main reason is that, as Bertrand Meyer said, *software correctness is a relative notion*. As he said, *"a software element is neither correct nor incorrect on it own; it is correct or incorrect with respect to a certain specification"*.

In the real world, I find all too often that the main reason for code to be incorrect is that the specificacion is not clear enough, that there is no clear *contract*. How can you write correct code if you don't know with certainty what *correct* looks like? In this article we will take a look at *Contract Based Programming*, an approach that focuses in arriving to a clear cut contract at the lowest level as a means to produce correct software.

Two styles of programming: tolerant and demanding

Let's study some sample code in order to get things rolling. **Listing 1** shows the code of `stdDev`, a function that calculates the standard deviation of a series of numbers.

```
public static double stdDev( double...values ) {
  int count = values.length;
  double average = mean(values);
  double sum = 0.0;
  for( double current : values ) {
    double value = current - average;
    value *= value;
    sum += value;
  }

  return result = Math.sqrt( sum / (count - 1));
}


public static double mean( double...values ) {
  double total = 0.0;
  for( double value : values) {
    total += value;
  }
  return result = total / values.length;
}
```

***Listing 1**. Calculating standard deviation.*

In the last line of `stdDev` we are dividing a value by the number of elements in `values` minus one. If the number of values is one, we will be dividing by zero. If the number of values is zero, then we will be calculating the square root of a negative number. And, what will happen is `values` itself is `null`? How should we handle these issues?

On the one hand, you might argue that `stdDev` should be as robust as possible, and therefore it should raise an exception if the number of elements in `values` is less than two, or `values` is `null`. This we will call the *tolerant* style of programming, which can be summarized with the phrase "throw whatever you want to my method, I'll handle it or either I'll raise an exception if I can't do what I have to". On the

other hand, you might argue that the user of `stdDev` must do whatever it takes to call it with an array of two or more elements: if he does not guarantee that, then that's a *defect* (a bug) in his code. This we will call the demanding style of programming, which is well summarized by the phrase "it is your responsibility to call functions with valid data, don't bother me with data in bad shape".

To be fair, this is not a black or white distinction, most programmers will prefer one or the other approach, but they'll probably use both styles to different degrees. In fact, I think the key issue is not which style is best. Whatever style you prefer, the most important thing to do is to establish a clear and explicit *contract* between a function and their users. The contract includes the obligations of the caller, what the function guarantees if the caller keeps his word, and the special cases that might lead to the function not being able to do its work and raise some exception.

Whether the contract is "`stdDev` will accept an array of any size, but it will raise a `WhateverException` exception if the number of values is less than two" (tolerant), or it is "the user of `stdDev` has the obligation of calling it with an array of two or more elements and, in return, I guarantee that a correct value will be returned", what's important is that it is clearly established. The method users should never be left guessing about the contract. In fact, it is much better to have a clear contract, even if it is not the contract of your dreams, than having to guess –or just missing the fact that there might be special cases that can lead to failure. Let's make things usable, not perfect.

Preconditions

The conditions that the caller of a method must guarantee before it can call the method are called preconditions. Of course, the preconditions must be reasonable for the contract to be honoured, but this is not all that difficult: most disagreements about preconditions come from the fact that different programmers tend to have different preferences when it comes to deciding between tolerant or demanding contracts. Don't fall in this trap. Get a contract.

Of course, if the writer of a function is so kind as to check the contract for us, just to help in getting things right, then we will be more than happy. How kind of `stdDev` to whisper us "uh, oh, you have broken the contract because the array has less than two elements", whenever we forget the contract or just can't manage to fulfil it. Note that this is not returning to the "`stdDev` will handle it all" policy, because all we want is just some help while developing, in debug mode. In fact, we want those checks removed at runtime, because their only purpose is to detect defects in the calling code. And yes, any violation of the preconditions implies that there is a defect in the calling code. You can perform the required checks using the `assert` statement, as in **Listing 2**.

```
public static double stdDev( double...values ) {
  assert values != null;
  assert values.length >= 2;
  // ...
}
```

***Listing 2**. Using Assert in stdDev.*

Java includes the `assert` statement for checking assumptions about your program since Java 1.4. This statement is ignored when you run your application without passing the `-enableassertions` switch to the JVM. This means that you should always pass this switch to your JVM during development. If this switch is on, and the condition checked is false, then an `AssertionError` exception will be raised. This exception is not meant to be handled, and you should only ensure you perform adequate cleanup. Once you have violated a contract, nothing is guaranteed. Remember that a function makes the promise that it will work only if all the conditions of the contract are met when called.

As a matter of fact, you should always check all preconditions on entry to your functions, before anything else is done. This will serve as a superb debugging aid. Furthermore, its value as documentation shouldn't be underestimated. While documentation tends not to be very reliable, the code must always compile and work: therefore, assertions will always tell the truth about how to use a function.

Given that preconditions are so important as a documentation aid, I tend to isolate minimally complex checks in functions that provide a descriptive name. For example, **Listing 3** shows a hypothetical method that sets a range of dates. Do you *immediately* understand what the precondition is? I don't, even though I can guess. But guessing might not be a good idea, right? And, yes, you can check the code and understand what it does easily, but what I want is to understand it *now*.

```
public static void setIntRange( Integer min, Integer max ) {
  assert min != null && max != null &&
         min.intValue() <= max.intValue() &&
         min.intValue() >= MIN_VALUE && max.intValue() <= MAX_VALUE;
  // ...
}
```

***Listing 3**. A precondition you have to chew a bit to understand.*

(SEE LISTING3.TXT)

Listing 3. A precondition you have to chew a bit to understand.

Listing 4 is equivalent, but the precondition calls a method to perform the check, which is self-explaining. As always the devil is in the details: in **Listing 3** you need to understand the *how* to get the *what*, whereas in **Listing 4** you get the *what* immediately. If you want the *how*, you take a look at the code for `isValidRange`, but only then.

```
public static void setIntRange( Integer min, Integer max ) {
  assert isValidRange(min, max );
  // ...
}

public static boolean isValidRange( Integer min, Integer max ) {
  boolean ok = min != null && max != null &&
               min.intValue() <= max.intValue() &&
               min.intValue() >= MIN_VALUE &&
               max.intValue() <= MAX_VALUE;
  return ok;
}
```

***Listing 4**. A self-documenting precondition.*

If you think about it, it is not reasonable for a method to demand that a very complex precondition is met, if there is no accompanying query function that allows the caller to check that. Even a function as easy to write as `isValidRange` should be provided as a courtesy, instead of forcing the method user to write it. Make your user's life easier! While this is plain common sense, I have used more than one library that makes you jump through hoops to check for some preconditions. In fact, I have found cases in which I had no way to check for the error condition!

Other benefits of contracts and assertions

Contracts have many benefits, especially when assertions are used to help everybody honour them. The first benefit is that they force us to think harder about the responsibilities of our code, contributing to a better understanding of the problem we have to solve. This injects quality in the code from the very beginning, something whose impact should not be underestimated.

As we have already seen, assertions provide an excellent support for debugging. They are superb as documentation, as well, because assertions are never desynchronized with reality, while documentation tends to get out of sync sooner or later. Even worse, most documentation doesn't even consider preconditions in most cases, and the source code will end up being the real documentation: assertions are invaluable in this scenario.

Exceptions vs. assertions

The functions we have seen make very reasonable demands when it comes to preconditions. However, there are conditions that we can't reasonably demand to be met. First of all, there are events that can't be anticipated, such as hardware failures, a file being deleted just before we try to open it, etc. In cases where we can't perform checks in advance, we have to react to the problem as it happens.

Exceptions are a good mechanism to handle unexpected problems, allowing a function to make it clear that it can't do what we want, and that somebody else will have to do something about the problem. This is so because exceptions can't be ignored the way you can ignore a return value that indicates that there is a problem. Using exceptions tends to separate code handling the exceptional cases from "normal" code, making things cleaner. Furthermore, exceptions can contain lots of useful information, because you can create new exception classes with additional data.

Note that there is a subtle division of labour between exceptions and assertions: assertions help us in making the code correct, while exceptions help us in making the code robust, allowing us to react to problems and exceptional conditions. To me, the fact that `assert` just raises an exception is almost an accident, and you shouldn't think about `AssertionError` as a "real" exception: the purpose of `assert` is to signal a defect in the caller's code, not to allow us to recover from defects. We want to avoid defects, not to recover from them.

The grey zone

As always, there is a grey zone. There are cases where a check can be provided in advance, but it will have a cost greater than just

performing the operation and raising an exception as needed. For example, there are matrix operations for which performing a check is almost the same as performing the "real" operation: in this case, performing the check first will take twice the time, and is therefore not much of a good idea. Raising an exception as needed is the way to go in these scenarios.

Note that run-time cost is not the only cost to be considered. For example, checking that a file name is correct is not as easy as it seems (think about UNCs and all those things), and it might be better to call `open` and raise an exception if it fails due to the name being incorrect, instead of wasting lots of time trying to write a function that checks whether a file name is correct. However, I recommend that you try to be as strict as possible with preconditions, preferring reasonable preconditions to relying on exceptions.

More about contracts

Up until now, we have talked just about the obligations the users of a method have, that is, preconditions. However, the method has its obligations, too. For example, consider an `insert` operation for a binary tree. If you faithfully follow the rule forbidding insertion of `null` objects, then it must guarantee that the binary tree will continue to be sorted, and will contain the specified object. That is, the other party is obliged to meet one or more postconditions. In fact, if a function doesn't meet a postcondition, then there is a bug in its code, the same way a precondition violation means that there is a defect in the calling code.

In many cases, postconditions seem a bit dumb, but they have their uses too. For example, I wouldn't mind adding an assertion to `insert` that checks that the binary tree continues to be sorted once the insertion has been performed. Postconditions are interesting because they force the implementer of the code to do his homework and test things adequately. In our case, for example, all it would take for the implementer to check the postcondition is to call an `isSorted` function. If he has already written it, that's easy. If he hasn't…well, I wouldn't use a binary tree for which nobody has ever checked if it is really sorted, and that's what I will think if he hasn't implemented an auxiliary `isSorted` function -even if it is for debugging purposes only.

If you think about it, it should be true for all public `BinaryTree` methods that the tree is always sorted on entry, and on exit. This is a condition that must be always true, and this is why it is called *class invariant*. Class invariants look like a theoretical thing, but they can be very interesting as a tool for deeply thinking about our classes, before we implement them. In fact, I think that it is a good deal to write the class invariant for all complex data structures as the first step for implementing them.

Furthermore, take into account that it is very important that virtual and protected methods are documented well enough that implementers of derived classes do not implement or use them in a way that leaves the class unstable.

Last, but not least, it can be very helpful to place assertions in selected places, just to check that things are all right: for example, you might want to check that an intermediate database is closed

before you reach a certain point in your code, etc.. Better to use `assert` in excess than to use it very little.

Being reasonable with debug mode

Right now, it is clear that we should use assertions extensively, and that they should be always active in debug mode -the mode in which programmers really live!

However, we will have to limit the reach of that debug mode. Think about a `binarySearch` function. Clearly, for `binarySearch` to work, it must receive a sorted collection –a precondition. But the binary search is a very fast operation, and the `isSorted` operation is much slower: `binarySearch` is an O(log(n)) operation, but `isSorted` is an O(n) operation: in other words, to check whether one million of elements is sorted will require one million comparison, while the binary search itself will cost you something like twenty comparisons. This is a level of overhead unacceptable for everyday work, even conceding that debug mode has to have some overhead.

The problem here arises due to the mismatch between the cost of "real" work and the cost of checks. If something costs you 100 seconds, then you might accept adding another 50 seconds in exchange for debugging support, but if something costs you 1 second, you will not want to pay 50 extra seconds each and every time.

I think that the overhead added in standard debug mode should never be of a greater order of magnitude than the "real" code itself. Therefore, I advocate that you should develop with checks that do not change the time or space complexity of an operation. You should handle operations that add much more complexity especially, probably only when actively debugging the code containing the expensive checks -in what we might call active debug mode.

Inheritance and contracts

Inheritance has subtle implications when it comes to contracts. When you define the contract of a virtual function, things get pretty interesting. Basically, the overridden method in derived classes must be always equally or less demanding when it comes to preconditions. The opposite happens with postconditions: an overridden method must guarantee at least the same postconditions -or more.

Why? Imagine that a virtual method in the base class says that it can accept `null` objects (a precondition), and that it guarantees that the result is sorted (a postcondition). If a derived class overrides the method and adds the precondition that it can't accept `null` objects, then any code that can work with the base class might crash: because it *knows* that `null` can be handled correctly, it will pass a `null` value sooner or later, and then the derived class method will fail.

When it comes to postconditions, if the base class method is promising that it will return data in sorted order, and a derived class overrides the method so that it does not respect that postcondition, then code that works correctly for the base class will not work for the derived class, leading to the violation of the promises made by the base class.

With respect to class invariants, a derived class must have either the same or a more restrictive invariant.

Leaving assertions in release mode

The purpose of assertions is to support contracts in debug mode, to help us in our quest for correctness. However, given that `assert` will raise an exception when the contract is broken, it is tempting to think that leaving them enabled in the release version might enhance the robustness of the code.

To be fair, leaving assertions activated in the release version can help you diagnose problems, and might even avoid greater damage. However, this is not guaranteed, and it might do more harm as well. A different thing is that you decide that you should be able to handle some assertion violations. Well, from the very moment you want to react to assertions, it makes no sense for them to continue being assertions. Change the contract and raise a "real" exception! Remember, assertions are there to support correctness, while robustness is provided by exceptions. Don't mix things.

Should I be tolerant, or demanding?

At the beginning of the article we have mentioned that there were two different styles for handling contracts, what we called the tolerant style ("I'll handle whatever you throw at me"), and what we called the demanding style ("No, no, I can't accept an empty array, it is your responsibility to call the function with valid data"). Before discussing this style issue, let me repeat this: deciding the style is secondary compared with the need of establishing a clear contract. You have been warned!

Undoubtedly, the tolerant style looks safer. The promise is that, no matter what happens, a method will never fail with a disaster, because it will raise an exception if there are problems. This has a subtle effect, which I think that we programmers tend to underestimate. It can undermine our willingness to assume the responsibility for ensuring that all is at it should be, because "nothing really bad will happen" if that's not the case.

Besides, the purpose of raising an exception is to allow somebody else to react to problems and exceptional cases, but it is highly unlikely that you want to raise an `AssertionError` exception so that other code can react to it. If that were the case, you could have used a "real" exception, instead of an assertion.

I think that highly reusable code must take a minimum of decisions, and therefore the demanding style is better suited to write reusable code. On the other hand, I admit that tolerant code can be really easy to use, and when reusability is not a concern, it can suit your needs very well.

Additionally, in tolerant mode the same thing will be checked over and over, adding overhead in many places. Just checking things where it is really needed will be faster: decide who has the responsibility of checking those things, and just make sure that the check is performed when needed. You'll write faster and cleaner code if responsibility is clearly delimited in each and every case.

As you can see, I prefer a demanding style. I'm sure that you can serve better your clients by recognizing your limits, instead of trying to be all to everybody.

About the author

Pedro Agulló is a consultant, programmer and freelance writer in Barcelona, Spain.

He specialises in mentoring on agile methodologies and techniques, building & leading agile teams, design and development of complex business models using TDD (Test Driven Development) and all kinds of programming with a degree of complexity.

He has published more than one hundred technical articles in printed magazines such as Delphi Magazine, Revista Profesional para Programadores, Programación Actual, Delphi Informant, etc.

He is the author of DirectJNgine, an open source Ajax library that allows direct access to Java objects from Javascript. You can find it in code.google.com/p/directjngine/