



# The Importance of Being Declarative

Diomidis Spinellis

**A DECLARATIVE PROGRAMMING** style focuses on what you want your program to do rather than how to perform the task. Through diverse programming techniques, libraries, and specialized languages, you end up with code that sidesteps nitty-gritty implementation details, dealing instead with a task's big picture. For instance, instead of writing the following C code to calculate a number's factorial

```
int result = 1;
for (int i = 1; i <= n; ++i)
    result = result * i;
```

in Haskell, you might simply write `product [1..n]`. Let's see why you should strive to program declaratively, and how to go about it.

## Why?

By avoiding implementation details, well-written declarative code is easier to understand, modify, and maintain.

When you work with such code, you can concentrate on an operation's essential parts without getting distracted by details that are safely hidden away. Modifications are easier, first because what you need to change is easily discernible, and second, because there's less stuff to change. Shorter code is also more reliable. Whenever I get the opportunity to write declarative code, I'm always pleasantly surprised by how few errors the code has in comparison to code I would have written in plain C. More often than not, the code simply runs flawlessly on the first try. This is fortunate, because declarative code's behind-the-scenes execution is often complex and therefore difficult to debug.

The code you write in a declarative style is often so readable that you can share it with your project's domain experts, even if their IT knowledge is only rudimentary. You can thus discuss with them your implementation, have them go over your code to verify it, or even ask them to provide you with their own code, based on existing examples. When implementing a civil engineering CAD system, I coded the user interface in a declarative fashion using a custom-built language. Today, a civil engineer on our team with no C/C++ programming experience routinely changes the corresponding files to polish the user experience.

Interestingly, once you start working with declarative code, you can benefit handsomely in ways that aren't directly related to the code's execution. Given that good declarative code is essentially a system's specification written in a machine-readable fashion, you can automatically process that code to verify properties of its operation, generate test cases, or create parts of the system's documentation. In one case, I helped replace imperative Visual Basic code that modeled some financial instruments with Haskell code that declaratively specified their behavior. With the declarative code in hand, we could then implement the models, perform risk analyses, and even generate the formulas for the corresponding contracts.

Sadly, as the saying goes, there's no such thing as a free lunch. Declarative code is often slower and takes more space than a corresponding imperative implementation. This happens because the compiler or runtime system that gives you the benefits of a declarative style is general enough to handle all the possible cases you throw at it. Consequently, it often misses the opportunities for task-specific optimizations and shortcuts you could apply to the problem at hand. For instance, some implementations of my earlier declarative factorial example might waste space by first generating a list of all numbers and

Post your comments online  
by visiting the column's blog:

[www.spinellis.gr/tools](http://www.spinellis.gr/tools)

then multiplying the terms. However, once you have your code written in a declarative fashion, it's often easier to spot opportunities for higher-level optimizations, such as replacing a naive data structure with a sophisticated one or parallelizing a task. For large data sets, these optimizations are a lot more profitable than any bit twiddling you could perform in low-level code.

## How?

You can program more declaratively by making suitable choices about many parts of your system's implementation: code, data types, libraries, and languages. Advances in compilers and model-driven development, the availability of powerful libraries as open source software, and powerful hardware make such choices particularly attractive.

You don't have to program in an exotic language to write declarative code. Small choices, such as the naming of your methods and variables, matter. Name a method based on what it does, rather than how it performs its action. Thus, `getReplacement` is a better name than `getMaxElement`. The same goes for variable names: `passengerSet` is a better name than `passengerHashTable`. Suitable formatting can also make a difference. For instance, although cascading `if`, `else if` statements are technically nested within each other, we typically indent them at the same level to stress that these are equal alternative choices. You can also profitably line up parts of separate expressions to indicate that these are related.

If you're coding an algorithm, have your code match the algorithm's published description down to the choice of the variable names. Resist the temptation to optimize, until you have data that shows that the implementation hinders the program's performance. Algorithms are often specified using high-level constructs and operations, such as intersections and unions of sets. If

your language supports such types, use them; otherwise, provide them. In all cases, choose (or implement) high-level data types that match your problem; don't work directly with arrays, pointers, or bits, striving to obtain efficiency through direct manipulation of low-level data. If your language supports interfaces (or abstract classes), code in terms of them, thus removing another implementation detail from your code. Judicious use of operator overloading can also make your code more declarative, allowing you, for example, to manipulate matrices using algebraic notation, rather than nested method calls.

In other cases, you can specify your problem's properties as data (perhaps in a table) and have a simple algorithm go through it to respond to its input. As an example, when implementing control software for a rolling mill, instead of writing a separate routine for each product type, I created a table with each product's parameters. The logical extension of such a design choice is to devise or adopt a more expressive domain-specific language where you can code your program's operation using the nomenclature associated with your application area. For instance, instead of laboriously counting pixels and calling methods to instantiate fonts and draw lines, you can specify a page's appearance in terms of borders, markers, and positioning through the CSS language. Or, to parse complex textual input, simply provide its grammar to a parser generator tool like Yacc or ANTLR. If a suitable language isn't available, don't shy away from building one, perhaps using the macro or meta-programming facilities of the language you're using.

Third-party or platform-specific libraries can also help you be more declarative by providing high-level abstractions or a domain-specific language. For example, by adopting OpenGL, you can describe a 3D scene in terms of concrete

objects, lights, and a camera. When analyzing text strings, specifying the pattern you're interested in with a regular expression is a lot more readable and maintainable than calling your language's string-manipulation methods. And if you want to run complex queries on your program's data, consider gluing to it an embedded SQL engine, such as SQLite or HSQLDB. Again, declarative SQL queries win hands-down over hand-coded loops on arrays.

When you have the choice, pick the highest-level language you can afford and that's suitable for the task at hand. Higher-level languages offer more and better abstractions and make it easier to be declarative. Consider the task of escaping from a deeply nested error. In C, you'll play dice with God by calling `setjmp/longjmp` to explicitly manipulate the stack frame. In Java, you simply throw an exception. Consider the management of concurrency. In Java, you explicitly manage threads (from a shared pool to avoid thrashing) and communicate using carefully managed shared memory structures. In Erlang, you fire (and forget) thousands of processes and handle communication through messaging supported by the language's syntax. Finally, consider finding customers who signed up last year and were late on two bill payments. A SQL query is the only sane way to perform this task. It clearly pays to be declarative. 

**DIOMIDIS SPINELLIS** is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of the books *Code Reading* and *Code Quality: The Open Source Perspective* (Addison-Wesley, 2003, 2006). Contact him at [dds@aub.gr](mailto:dds@aub.gr).



See [www.computer.org/software-multimedia](http://www.computer.org/software-multimedia) for multimedia content related to this article.