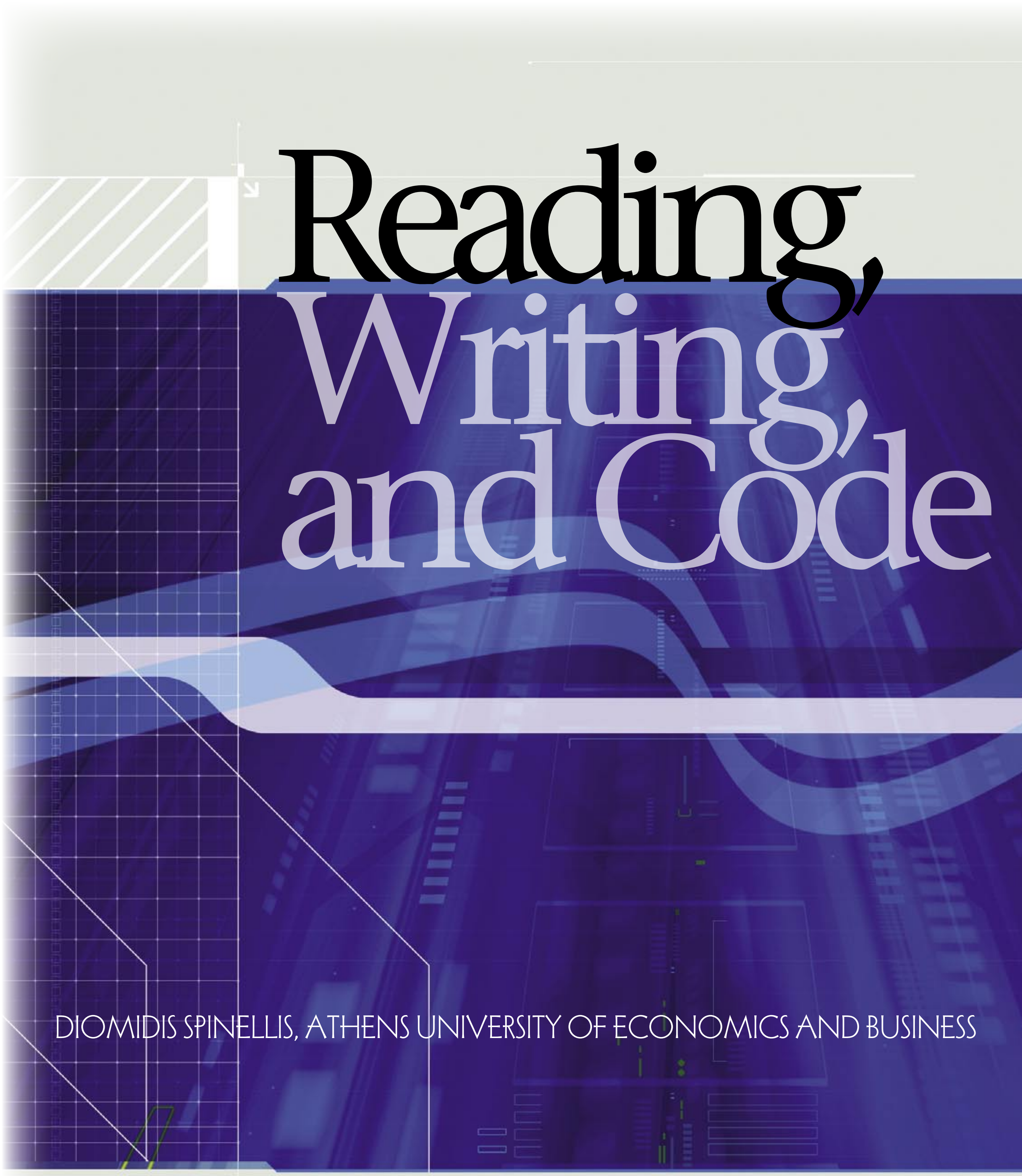
Reading, Writing, and Code



DIOMIDIS SPINELLIS, ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

The key to writing readable code is developing good coding style.

Forty years ago, when computer programming was an individual experience, the need for easily readable code wasn't on any priority list. Today, however, programming usually is a team-based activity, and writing code that others can easily decipher has become a necessity. Creating and developing readable code is not as easy as it sounds.

EASIER WRITTEN THAN READ

There's a theory explaining why computer code that is sometimes so easy to write is so hard to read, and it goes this way:

I am sure you have noticed that driving from your home to, say, a shopping mall where you've never been before can be a difficult navigating task. On the other hand, returning home from that mall tends to be a lot easier. The reason behind this asymmetry is a decision tree rooted at our home base. When venturing away from home into the unknown, every decision we make opens up more and unknown tree branches, but when we return to our base, our routing decisions eliminate unknown branches.

Similarly, when we write code, we can follow many different implementation paths to arrive at our program's specification, gradually narrowing our alternatives, thereby narrowing our choices—just as we did when we returned home from the mall where we had never been. The last character we type is often the only one that will result in a correct program.

On the other hand, when we read code, each different way that we interpret a statement or structure opens many new

Reading, Writing, and Code

interpretations for the rest of the code, yet only one path along this tree is the correct interpretation. We thus write code toward a decision tree root, but, unfortunately, read toward the tree's branches.

ONLY HUMAN NATURE

There's another reason why reading code is often so difficult, and here's where human nature comes into play. The benefits of code that works are immediate: usually a running system and a nice paycheck for the programmer. But the payoffs of readable code accrue in the future when our code will be maintained and extended. We humans typically discount future payoffs, especially when the payoffs will be collected by unknown code maintainers and not by ourselves.

Unfortunately, computer programming education often focuses on how to single-handedly develop programs from scratch in a single language and single execution environment, a development style prevalent in the 1950s and 60s. Nowadays, software development is typically a team-based activity and most often involves extending and maintaining existing systems written in a multitude of languages for diverse execution environments. It's now even more important to understand code concepts, forms, structures, and idioms to be able to write code that other programmers can read easily.

SCARCE READING MATERIAL

To complicate matters further, my research has revealed relatively little material on code reading, which may be because up until 10 years ago there was a scarcity of real-world or high-quality code to read. Companies often protect source code as a trade secret and rarely allow others to read, comment, experiment, and learn from it. In the few cases where important proprietary code was allowed out of a company's closet, it spurred enormous interest and creative advancements.

Consider, for example, John Lions' *Commentary on Unix 6th Edition, with Source Code* (Peer-to-Peer Communications, 1996), which used source code as a way of teaching operating system design. Although his book was originally written under a grant from AT&T for use in an internal operating system course and was not available to the general public, copies of it circulated for years as bootleg nth-generation photocopies.

In the last few years, however, the popularity of open source software has provided us with a large body of code that we can all read freely. Some of the most popular software systems today—such as the Apache Web server, the Perl language, the Linux operating system, the BIND domain-name server, and the sendmail mail-transfer agent—are, in fact, available in open source form.

UNREADABLE CODE

I'm often asked, "Of the 'production-level code' I've encountered, what sort is most unreadable?" I usually respond that system code, such as operating-system kernels, libraries, database systems, and graphics engines, can be the most difficult to disentangle for a number of reasons.

System code. First of all, performance constraints typically dictate the use of low-level constructs. Also, system code has to be extremely reliable, and you see this in the code as many special cases. System code often interacts directly with the hardware, adding another level of complexity.

My favorite example concerns the implementation of the quicksort algorithm. Robert Sedgewick, the author of *Algorithms in C++, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching* (Addison-Wesley, 1998), presents it as an elegant 19-line C program, yet the code I encountered in a real-world C library implementation spans more than 130 lines. The extra 111 lines are not code bloat, but the result of a carefully engineered and extremely efficient implementation. The extra lines represent the difference between theory and actual practice, and some developers encounter this difference only the first time they have to deal with a real-world system. An advantage of system code is that it is in many cases written by very brilliant people, and their clear writing style tends to compensate for some of the inherent complexity.

Over-engineered object-oriented systems code. One other type of unreadable code I often encounter is over-engineered object-oriented (OO) systems. Software of this type uses deep inheritance hierarchies and multiple levels of delegation for no immediately apparent reason (other than to show off the programmer's mastery of OO concepts). When such systems lack design documentation—and, unfortunately, this is often the case—they can be more difficult to understand than any old-style spaghetti code. The reason for this is that the relationships between the code elements cannot be determined by trivially reading the code, since the code's behavior changes at runtime depending on values of each object's class. Think of it as a sequence of goto's with the target being stored in a variable.

I get a lot more frustrated with code that's gratuitously over-engineered than with inherently complex code.

OUT OF SILVER BULLETS

You might then ask, if the object is to avoid that sort of unreadable code, is there a silver bullet—a rule you can follow to make code readable?

Unfortunately, it's not as easy as that. I've seen every single rule that is supposed to lead to clear and readable code abused to the point of absurdity. I've seen supposedly structured code of Byzantine complexity, heavily commented code where comments just rephrase the code on their left (multiply var by two), and simple algorithms obfuscated through the overzealous application of design patterns. The programmers who wrote those programs didn't set out to write unreadable code; they just took the rules and applied them blindly.

The single factor that will result in code that is easy to read is style. You see, readable code is similar to good prose. While a writer may have learned that it's important to keep your sentences short and to use good metaphors, doing so doesn't necessarily guarantee a first-class novel. Outstanding authors know the rules of good prose—and they also know when to break them.

Similarly, programmers should consciously think for every character they type: "How can I make this keystroke count? How will it lead to code that is easier to read?" You will never produce truly readable code by programming on autopilot. Coding style involves paying attention to the program's structure, naming conventions, comments, indentation, and so on. But, more importantly, it involves consciously deciding at every point and at every level what construct and which expression will result in readable code. Deliberately passing each code element you write through this litmus test of readability is the best favor you can do for yourself and your coworkers.

PROGRAM READABILITY

The relationship between code readability and programming language design is quite complex, leading to some counterintuitive results.

For example, the first large production-grade software I ever laid my eyes on was the source code for the original IBM PC BIOS, published as an appendix to IBM's technical reference manual in 1981. The 6,000 lines of 8086 assembly code could have easily been a real mess. And yet, they were exemplarily structured into subsystems—such as self-test code, video, disk, keyboard handlers, and so on—split into procedures with clearly documented interfaces. Almost every line was commented. I actually learned to program the 8086 assembler by reading that code. The coding practices at IBM ensured that a program written in a primitive language was actually readable.

A decade later, as part of my military service, I worked on a personnel management system written in Cobol and maintained for many years by civilians like me, each spending just a few months on it and then moving on. Again, I learned to use the language by reading the existing code. Despite what I was expecting, the code was easy to follow and maintain, if a little dull. Here, another force was in effect—Cobol, especially as practiced by young computer science graduates learning it on the job, left very little room for creative experimentation.

There are actually three orthogonal issues affecting the readability of programs written in a specific language. People arguing for or against the readability aspects of a given language often fail to differentiate among them.

- First, there are languages that give you the tools to write good code, and there are languages that get in your way. Modern languages such as Java, C++, Ada, and Perl give you all the features you need to write large and understandable programs, while older languages such as Fortran 77 force you to work a lot harder to achieve the same level of readability.
- Second, there are languages that discourage you from writing bad code through the lack of "dangerous" features, and there are languages that give you more than enough rope to hang yourself and all your code's future maintainers. Pascal and Java are representatives of the first category, while C and C++ are in the second.
- Third, there are language features that are often gratuitously used or just abused, resulting in unreadable code. The goto statement, operator overloading, pointers, dynamic memory allocation, the C/C++ preprocessor, exception handling, templates, inheritance, and dynamic dispatch are some of these features. And yet, each one of these features has legitimate uses that lead to code that is more readable than code written without them. As examples, the goto statement allows you to decently exit from a deeply nested structure or to efficiently code a state machine; vector or complex number arithmetic using overloaded operators is a lot more readable than the equivalent function calls; and pointers used for traversing arrays actually allow more robust type checking than integer-type array indices. Also,

some of the features that lend themselves to abuse are gradually being replaced by more robust alternatives. For example, Java's labeled break and continue instances are more readable than the equivalent goto structures they replace.

COMMENTS ON COMMENTS

Earlier, I mentioned how easy it was to read the source code for the original IBM PC BIOS because every line had been commented. And yet, there are developers who say their code is "inherently readable," that it needs no commenting, that all you need to do is "read the code."

Let me relate to you an experience I had in 1988 while working as a research student at the European Computer Industry Research Centre in Germany. One day I found the source code of eighth edition Unix lying unprotected on a disk partition. I immediately dwelled on it, looking for how my heroes who gave us C and Unix were actually programming. Reasoning that the ed line editor must have been a part of the Unix system from its inception, I located its source and started reading it. As I remember, the complete editor consisted of a single source file starting immediately with a global variable declaration or an include statement; there was no leading comment. That surprised me so I searched the code for another comment. There was none. Somewhat disillusioned, I started exploring the code. The variable names were short, many structured programming rules I had learned at the university were broken, yet the code was surprisingly readable and easy to follow.

Unfortunately, I know that if I end my tale there, programmers will rush with this article in hand to their bosses and say, "See, there's no need to write comments!"

Well, let me break the news to you: Most of us are not ACM Turing Award winners Ken Thompson and Dennis Ritchie. We don't routinely write a functioning Unix kernel and a C compiler in PDP assembler and then rewrite it and bootstrap it in C. And even if our coding capabilities are at that level, we almost certainly don't work at the 1970s Bell Labs where the person next door who will read our code tomorrow has probably either designed the language or written its compiler.

So, as a rule, we need comments to make up for the less-than-brilliant coding structures and naming constructs we employ. We also need comments detailing the functioning of an algorithm to help our colleagues who haven't yet memorized the complete Donald Knuth's *Art of Computer Programming* (Addison-Wesley, 1998).

In code that I write for my own use—code I don't expect anyone else to read—I recently counted approximately one comment for every 10 lines of code. Based on what I said before, you can interpret this as admitting that I am a lousy code writer and an even worse code reader. But the truth is that good comments are a valuable tool to aid code reading. Even if the code is self-documenting, a good comment can act as a shortcut, saving you the time you would need to understand what a particular method or function does.

Comments are also used as a basis for automatically generating the system's technical documentation. I am thinking of applications like Javadoc here. If a project is generating technical documentation, these comments are a far more productive method of generating it and keeping it up to date than drafting it in isolation.

But, you might ask, because it is so easy for them to become inconsistent with one another, can't comments become more of a hindrance than a help?

The "comments and code will drift away, so let's not write comments" argument is an often-repeated yet typically unfounded and lame excuse. I've read thousands of lines of code, and found that the two biggest hindrances in understanding a system are badly written code and a paucity of comments. In only a very few rare cases have misleading comments been a real obstacle to me.

It is true that a misleading comment can actually send you on a wild goose chase because we tend to trust the wording of the original programmer over our own judgment, but this is also often true of code. When we see an if statement, we tend to assume that there are conditions that will cause the execution of the code in its body. In sloppily maintained legacy systems, however, I tend to find more instances of dead code than inconsistent comments. Nobody is seriously suggesting to stop writing code because over time it becomes inconsistent with its own structure. And yet, many are attacking the writing of comments merely because they offer an easy target.

PREFER NOT TO COMMENT?

Programming tends to be an intense, immersive activity. In all such activities, anything that distracts from the immediate goal—sleeping, healthy eating, social interaction, and, unfortunately, in our case, commenting—tends to be assigned an infinitesimally low priority. Programmers are simply following traits that are the norm in other creative activities, like artistic painting.

I admit that I tend to be drawn into the same behavior. The solution I've adopted is to put off commenting until a bit later, but not too much later. When I finish writing a small unit of code—such as a class definition, a single method, or even a discrete part of a procedure's body—I know it's time to comment. Programmers who put off commenting until all the code is completed—and their managers who put up with them—will never end up with properly commented code. Even if you find time to comment afterward—and typically you don't—the comments will kowtow to the "comment god" and, in most cases, will be useless since they simply replicate your understanding of the code when you are reading it.

It is my observation that programmers don't comment mainly because they find the task so taxing. Form, such as a programming language's syntax and the design patterns we follow, liberates. Thus, code is a relatively easy medium of expression. Conversely, expressing ourselves accurately and concisely using natural language is quite difficult and, sadly, is not a skill being emphasized in the typical engineering or computer science course. Alternating between the two forms of expression is even more difficult, since it involves rapid context switching between two different brain processes competing to use the same underlying apparatus.

We programmers call this phenomenon *thrashing*, and anyone who has tried translating between two foreign languages will have an avid recollection of the difficulty. I believe it is a good idea to perform creative coding and commenting in batches. Another approach is to first write a series of empty method definitions together with their associated comments, and then fill in their bodies.

ONE FINAL IDEa

Novice programmers may find that they can format their code in a consistent way, making it more readable if they use an integrated development environment (IDE) with an intelligent editor. For more experienced programmers, code written using an IDE is typically less readable than code written using a program editor. That is because an IDE can increase programmer productivity, but it will impose on the software its own view of the world. Lacking the IDE view, a future maintainer will probably find many loose ends and unconnected pieces. As long as you use the same IDE to read and write code, you get a polished look of your system through the beautifying lens of the IDE. When that effect goes away because, perhaps, you have switched IDEs or moved to a platform lacking one, your code's structure crumbles.

An extreme example is Visual Basic code. I have, at times, browsed the textual source code of saved VB projects or used `egrep` on the code to locate lines matching a complex regular expression. But I would never dream of developing or even editing a VB program outside the IDE.

Also, keep in mind that with some IDEs it's impossible to develop readable code. I am talking about IDEs that save your code only in a binary file format. I avoid those like the plague. The native code format can't be read by humans or other tools, and it locks you into using the particular IDE. You can't put the code into a version control system of your choice, you can't see the differences between file versions unless the IDE supports this feature, and if a byte in the code gets corrupted, you're hosed.

My best advice to programmers is to learn in depth a powerful and widely available editor, like Emacs or Vi and its descendents. Write your code using that editor so that your colleagues who will read it 10 years from now using a tool of similar capabilities will be able to share your view of it. Modern IDEs do many things much better than individual tools: symbolic debugging, incremental compilation, and code browsing come to mind. All these activities are a boon to your productivity, but don't directly affect your code.

So, use the IDE for those tasks. But don't lock your code into it. Your code is likely to live longer and travel further than the IDE you use to develop it.

CONCLUSION

In the years to come, programmers will increasingly value coding that not only works, but that also can be easily read and understood. To withstand the test of time, coding must pass the litmus test of readability. Q

LOVE IT, HATE IT? LET US KNOW:
feedback@acmqueue.com or www.acmqueue.com/forums

DIOMIDIS SPINELLIS holds a master's degree in software engineering and a Ph.D. in computer science, both from Imperial College (University of London, UK). His recent book, *Code Reading: The Open Source Perspective* **(2003), inaugurated Addison-Wesley's Effective Programming Series. He is an assistant professor in the department of management science and technology at the Athens University of Economics and Business, Greece.**