Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Code Documentation

Diomidis Spinellis

Technical prose is almost immortal. — Frederick P. Brooks, Jr.

Although programming is a form of literary expression, the relationship between code and its documentation is uneasy at best. As Figure 1 shows, among the thousands of projects that FreeBSD maintainers have considered important enough to port to the platform, the number of comments per 100 lines varies substantially. Clearly, as developers, our views on how we should document our code are anything but consistent. Yet, there are universal principles, nifty tools, and useful practices that can benefit us all.

Principles

The golden rule of programming—DRY (don't repeat yourself)—is particularly important when we document our code. When commenting, we're always a couple of keystrokes away from disaster: restating the code's function in English. And that's when problems start. When the code changes, and change it will, the comment is likely to be left behind. At that point, we have an apparently helpful comment stating something other than what the code does. Enter confusion. Developers are more likely to respect and maintain comments that carry their weight, comments that help them navigate the code by outlining an algorithm, elaborating a data structure, or documenting unexpected edge cases. Useless comments are worse than missing ones, because they occupy screen real estate and sidetrack developers.

It's much easier to follow the DRY principle when the code and its documentation live together in the same file. This was the original idea behind the literate programming movement. Some might claim that movement failed to gain traction. The truth is that its main principles are now part of modern programming platforms. Languages and libraries supporting higher levels of abstraction bring code closer to our design by hiding unneeded implementation details. Look at legacy software and you'll find it riddled with code implementing (often substandard) data structures and algorithms. Modern code simply reuses efficient containers, databases, and services.

Self-documenting code is sometimes a tasteless joke, yet the principle is sound. Comments should be our last resort for documenting code. In particular, bad code should be rewritten, not documented. This reduces evil duplication and spreads the benefits throughout the code. For instance, although the comment above the `zgedi` Linpack subroutine explaining that it "computes the determinant and inverse of a matrix" is useful, a descriptive name would make all the instances where it's called more readable. The same goes for development processes: these should be automated rather than documented as manual steps. A Readme file outlining the 15 steps required for releasing a software's new version is useful, but build rules that automate those steps would be divine.

This brings me to one last important principle: comments aren't only for code. Any software artifact we produce deserves our love and comments. This includes shell scripts, makefiles, link specifications, batch files, debugger scripts, and configuration files. As I've seen with my students, following this principle is an indication of a maturing developer and an early sign of professionalism.

0740-7459/10/$26.00 © 2010 IEEE

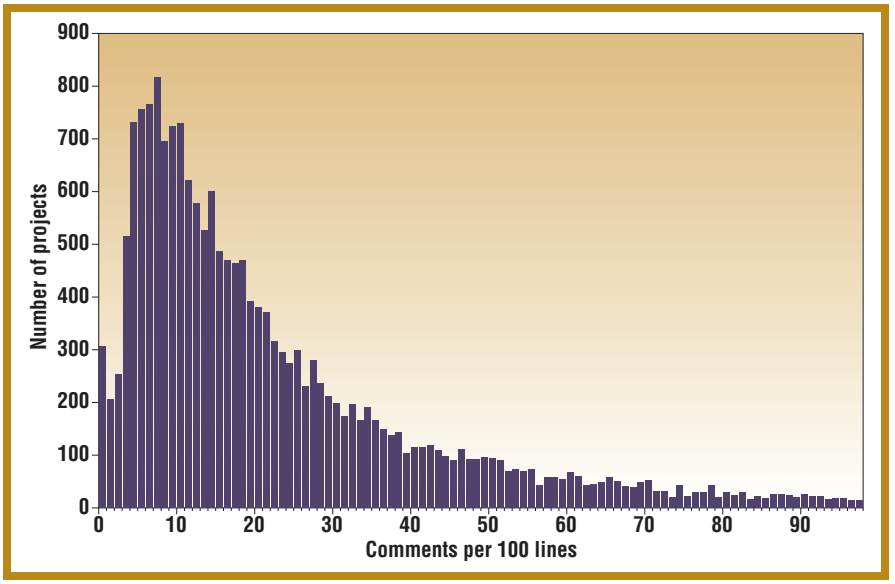

Figure 1. Comment density in third-party projects ported to the FreeBSD platform. The code of 307 projects has less than one comment per 100 lines. The wide distribution of the numbers shows the variance in code documentation practices.

Tools

We developers are, by definition, always trying to offload boring work to a computer. Therefore, there's no shortage of tools to handle the documentation's drudgery. Most useful are those that create online documents by extracting text from specially formatted comments. The use of *javadoc* is now standard practice in the Java world. For other languages, *doxygen* offers similar functionality, and many C and C++ programs rely on it. Interestingly, both tools (javadoc via the UMLGraph plug-in) can also reverse-engineer the code to create class diagrams. This is a case where we get valuable documentation, the proverbial picture worth a thousand words, for free.

Sometimes the problem is so complex that code must take the back seat. In such cases, instead of extracting readable prose from our source code, we must go the other way around and embed source code into a larger body of explanatory text. Here, we need tools to format source code in the most readable way. The choice of tools depends mostly on our text formatting system. On troff we'd use *vgrind*; on LaTeX, the *listings* package; and with DocBook, the *programlisting* tag. There are also many tools that will format code for HTML display; see the list in Wikipedia's *syntax highlighting* article.

Whenever your code contains the definitive version of facts that must be documented, consider using a custom tool to automate the generation of that documentation. I've used this approach for creating lists of error messages and their explanations, an outline for a product's manual, and documentation for a database's schema, while colleagues have thus created contracts for complex financial products. You can follow this path by processing specially formatted code or comments, by coming up with a domain-specific language that will generate both code and documentation, or even by having your code create the documentation at runtime. The concept is always the same: DRY.

Best Practices

We can easily determine what makes code documentation great by looking at successful software platforms. Completeness and consistency are by far the most important attributes. Consider the original Unix manual pages, which documented every system call and library function using the now classic structure: Name, Synopsis, Description, See Also, Diagnostics (later morphed into Return Values and Errors), and Bugs. The C++ standard template library (STL) upped the ante by documenting—apart from concrete entities—definitions, semantics, complexity guarantees, invariants, and models for abstract concepts, like an associative container. The documentation of Microsoft's universally adopted open database connectivity (ODBC) specification distinguishes itself by listing all possible error codes for every function; information that is sadly still not available for other parts of the Windows platform.

Effortless accessibility is another best practice we find in successful platforms. Again, the Unix manual pages are notable here, because users could view them online on any character terminal (even a Teletype), but also read them in high-quality typeset form. Perl's POD (plain old documentation) markup is also a chameleon of sorts, as it can be easily transformed into every imaginable output format. The documentation of Sun's Java platform is remarkable due to the extensive use of hyperlinking. Its most recent version contains more than 800,000 links.

Finally, we see that successful platforms produce their documentation automatically using a low-overhead process. Java and Perl include the documentation in specially formatted sections of code; a valuable practice in itself that most modern languages have adopted. Using such comments, javadoc processes the 7,000 Sun JDK (Java Development Kit) source files to create more than 12,000 HTML pages containing over 218,000 named elements. Automated builds ensure that the documentation is always current and consistent. IDEs such as Eclipse rely on this infrastructure to provide API help during program editing.

In his classic book *The Mythical Man-Month: Essays on Software Engineering*, Fred Brooks describes how, six months into the IBM OS/360 implementation, he realized that stacking the 100 five-foot-thick copies of the project's documentation would tower above Manhattan's Time-Life building. He also found that the maintenance of documentation changes would take a significant part of each workday. Luckily nowadays, the tools we have at hand make Brooks's problems sound quaint. We therefore owe it to our past, present, and future colleagues to create brilliant code documentation.

Diomidis Spinellis is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business. Currently, he is serving as the Secretary General responsible for information systems at the Greek Ministry of Finance. Contact him at dds@aueb.gr.