

**developer.\*****A Web Magazine for Software Developers**

## Principled Programming

By Daniel Read

Author's note: This was the first essay about software development that I ever wrote, which was sometime in early 2001. I enjoyed writing it so much, that I decided to start the **developer.\*** web site so that I could have a place to publish other writings. During the first two years of the site's existence, "Principled Programming" was by far the most popular article.

I had intentions at one time to expand to a list of about thirty principles, with the idea of collecting them into a book eventually. I still may do that someday. Looking back at these principles now, two years after writing them, I'm mostly happy with what I read. While I'm resisting the urge to do some re-writing on this one, I still agree 100% with the principles as stated, and hope that you'll get something out of them.

### The Principle of Personal Character

---

Whether we know it or not, and whether we like it or not, our character is reflected in every line of code we write, every report we design, every user interface we build, every diagram we produce. When another person looks at our code—or just as important, the output of our code—that person, consciously or not, makes a judgment about us. Think back on code you have written...what would that judgment be? Would you be proud to stand up and take ownership of your code? Or would you have to sheepishly admit that it's yours, and launch into excuses for why it's not as good as it could be? If code is bad, then another developer reading that code is likely to assume that this poor quality is not an isolated event. The good news is this: we have absolute control over the quality of our code.

The question is not about whether or not one has the *ability* to write the best code possible, but whether or not one cares to even try. If one lacks in certain abilities, but takes pains to write clear, readable, well commented code that at least shows that the developer has taken the time to learn some of the fundamentals, then that developer has done her due diligence—and that fact will be obvious to a keen observer. It would be unreasonable to fault a person for some level of inexperience, or ignorance of certain techniques. However, it is absolutely reasonable to find a connection between the overall quality of a developer's code and the quality of that developer's character.

**The Principle of Personal Character states:** Write your code so that it reflects, or rises above, the best parts of your personal character.

### The Principle of Aesthetics

---

One aspect of programming in general we too often neglect is that of aesthetics. Aesthetics is about beauty and elegance, and the appreciation of these qualities. Many people, however, believe that aesthetics is only important when talking about art and literature. Too few people realize the importance of beauty and elegance in everyday things, and too few developers realize the importance of these in the writing of code. Aesthetics is especially important in software development, a realm where we are constantly dealing in layers of abstractions. The aesthetic aspects of our abstractions are directly related to their understandability, and therefore their usefulness.

A developer should strive for beauty, no matter what tool or language he or she is using. Beauty can be achieved on many levels, from the high level of the overall elegance of a system design to the lowest level of the visual appearance of code on the screen. Neatness counts. The best code not only works efficiently and correctly, and is well formed from the compiler's point of view, but the best code is also visually pleasing to the human eye—and therefore easier for the human brain to absorb and understand.

Steve McConnell writes in *Code Complete*, "The visual and intellectual enjoyment of well-formatted code is a pleasure that few nonprogrammers can appreciate. But programmers who take pride in their work derive great artistic satisfaction from polishing the visual structure of their code." (Page 399)

**The Principle of Aesthetics states:** Strive for beauty and elegance in every aspect of your work.

## **The Principle of Clarity**

---

Clarity is a state that must be actively sought. One of the biggest transgressions that we as developers can commit is to forget that our code has a life far beyond the few moments it takes us to write it. Chances are excellent that someone else—maybe even the original author—is going to have to deal with our code sometime in the future. Even if we write code that works perfectly and never causes a problem for the user, we have done our fellow developers (not to mention our employers) a disservice if we have not attempted to be as clear as possible.

There is a difference between clarity and correctness, though the two are often confused. Correctness is a primary focus of most developers, as it should be. The quest for correctness focuses on making the syntax correct for the compiler, designing the interface to meet the user's needs, and writing the algorithms to match the requirements. But if attention to clarity is not given equal emphasis, then the understandability and maintainability of the code will suffer. For our code to be as clear as possible, we must consciously use techniques such as informative naming, modularity, indenting, white space, strong cohesion, weak coupling, testing and documenting assumptions, and proper commenting.

Lack of clarity in our code causes unnecessary pain, and professional embarrassment, for our fellow developers who will have to maintain our code in the years (or even decades) to

come. As for our employers or benefactors, we short-change them by delivering less value; if poor enough, our code may even become a liability to the company. Leaving our fellow developers in this situation is discourteous to say the least, but leaving our employers in this situation is much worse: we make an agreement to produce quality code for a price. Our employers have paid us for our work, but how did they fare in the bargain?

**The Principle of Clarity States:** Value clarity equally with correctness. Utilize the proven techniques that will produce clarity in your code. Correctness will likely follow suit.

## The Principle of Layout

---

The Principle of Layout is a corollary to both The Principle of Aesthetics and The Principle of Clarity. The Principle of Aesthetics tells us that, besides providing intellectual enjoyment, beauty and elegance play a crucial role in well-written code. The Principle of Clarity tells us to make our code as clear and understandable to human readers as possible, and that clarity goes hand-in-hand with correctness. The Principle of Layout puts these dual principles into practice.

It is difficult to discuss the importance of good visual layout without referring to Steve McConnell's "Fundamental Theorem of Formatting," which states, "Good visual layout shows the logical structure of a program." (page 403, *Code Complete*) This means that layout not only serves to make the code look more attractive, but also works on a subconscious level to make the code more readily understandable to the reader. The goal is to reduce the amount of work that a reader needs to perform in order to understand our code. Layout should be our first tool for communicating clearly with human readers of our code.

There are several techniques that will ensure that layout of code suggests or reveals its logic. These techniques should be part of the foundation of every programmer's style: proper use of white space (blank lines), indenting, grouping together of related lines, using extra parentheses to make Boolean logic and mathematical formulas clearer, horizontal alignment of related code on separate lines, proper placement of block delimiters, etc. As McConnell suggests in his Theorem, the idea is to use the visual layout of our code to *communicate on a subconscious level* with the reader. Remember that when we are working in the realm of software development, we are always working with abstractions, and an abstraction is only as useful as it is understandable.

**The Principle of Layout states:** Use the visual layout of your code to communicate the structure of your code to human readers.

## The Principle of Explicitness

---

Following the Principle of Explicitness will save us and our successors incalculable trouble. The Principle of Explicitness is a corollary of the Principle of Clarity. But the Principle of Clarity is about making one's code clear and understandable to human readers. The Principle of Explicitness applies to human understandability as well, but more importantly, following the Principle of Explicitness makes our code more tolerant to change.

Here is a simple example: many programming languages and platforms offer the concept of a "recordset," which is a collection of data, in the form of rows and columns, usually pulled from a relational database. The recordset, often implemented as an object, is usually not provided by the language itself, but rather by some kind of library or component. When you write code to open up a recordset, you would normally have properties and/or parameters to determine the "cursor type." When your code calls the function to open the recordset, that function probably allows for a default cursor type, meaning that you do not have to explicitly specify what cursor type to use—your code can just accept the default.

If a developer accepts the default cursor type, that developer has now created an *implicit assumption* in his code, which reduces clarity. Worse, he has also created a gigantic opportunity for future bugs. The cursor type chosen directly effects the behavior of the recordset. Your code depends on that behavior. What happens six months later when the team upgrades to a newer version of the component that provides the recordset? What happens if that new version changes the default recordset type from and a "dynamic/updateable" cursor type to a "forward only/read only" type? Do you know what's going to happen to all of that code that accepted the default cursor type? It's going to break. Or worse, it's going to silently change some behavior; some `IF` condition is going to return `False` instead of `True`, and the `Else` block is going to be executed instead; and then months later, someone realizes that data corruption is rampant throughout the system.

By not being explicit in this simple example, the fictional developer who implicitly accepted the default cursor type made his code less tolerant to change. This is unacceptable. We must acquire the knowledge of where the potholes in the road are likely to be (the cursor type pothole is but one example of many, and every language and platform is full of others), and then we must use that knowledge when we write our code so that the potholes don't hurt us. As professionals, it falls on us to acknowledge the fact that change is constant, and to do our best to ensure that our code can weather those changes without failing or producing incorrect results. When our code does fail, we must ensure that it will do so gracefully, and that upon doing so, it provides as much specific information about where and why it failed as possible.

The key to avoiding potholes is to be explicit. Trouble lurks in the implicit—in the undocumented and untested assumption, in the arcane technique, and in the undocumented intent of the developer, which gets lost forever like a ship sinking silently in a vast, opaque ocean.

**The Principle of Explicitness States:** Always favor the explicit over the implicit.

## **The Principle of Self-Documenting Code**

---

Self-documenting code does not get written by accident.

As developers, we must give attention to the development of a sound style, which is the key to self-documenting code. We must constantly try to improve and perfect our style, so that each program we write is better than the one before it. A highly developed programming style comes through the diligent incorporation of proven techniques: informative, consistent naming; modularization that pays close attention to cohesion and coupling; avoidance of hard to understand techniques; clear layout; use of named constants; testing and documenting assumptions; and much more.

To learn these techniques, we must read the writings of all of those who came before us, for these people have already blazed the trail. Seek out the classic literature. Learn from the masters. Subscribe to magazines. Join internet discussion lists. Read lots and lots of other people's code. When you see code that falls short, analyze it, and try to figure out why. When you see code that reaches that high level we seek, you should be able to identify the techniques the developer used to achieve that level of quality. There is a saying: any fool can learn from his own mistakes; a wise person learns from the mistakes of others.

What about the role of comments? Do we still need them if our code is to be self-documenting? Self-documenting code, like perfection, is an elusive goal, and one that is probably impossible to achieve fully. However, that should not stop us from always reaching for the goal of self-documenting code. Where we fall short of perfection, we must supplement our efforts with good comments. Well written code should not need a lot of comments; it should speak for itself. However, some amount of commenting is required—the comments just need to be of the most useful types (see The Principle of Comments).

When viewed up close, truly self-documenting code is a joy to behold, and it becomes clear to the first-time observer that such a marvel could only occur through the efforts of a conscientious and diligent software engineer.

**The Principle of Self-Documenting Code states:** The most reliable document of software is the code itself. In many cases, the code is the only documentation. Therefore, strive to make your code self-documenting, and where you can't, add comments.

## The Principle of Comments

---

Comments are a double-edged sword. Used properly, they can infinitely improve the understandability and maintainability of code. Used improperly, they can clutter your code and render it *less* readable. Poor commenting is at best of little value, and at worst creates a huge mess.

The Principle of Comments has three parts:

First, comment in full sentences. This simple technique greatly increases a reader's comprehension of both the comment and the code it denotes. Sentence fragments tend to be cryptic. Writing each comment as a full sentence also makes comments more understandable to a wider audience, which is especially important in today's multi-cultural environment. High quality comments written as full sentences also act as an instructional aid for less experienced developers.

Second, use comments to summarize. "Summary comments" summarize a block of code in order to save a person from having to read all of the code that the comment describes. A summary comment would usually appear as a few lines at the top of a block of code. Note that a good summary comment does not repeat the code, but rather "distills" several lines of code down to two or three sentences.

Third, try always to comment "at the level of intent." What this means is to comment at the level of the problem, rather than the level of the solution. The code is the solution to the problem being solved. Ideally, the code should speak for itself (see The Principle of Self Documenting Code). A person can read the code, and if it's good code, should be able to readily see what the code is doing and how it's doing it. However, what is lost over time is *what was in the mind of the developer who wrote the code*. In general, *this* is what needs to be commented on. What was the *intent* of the developer? How is this code *intended* to be used? How does this code *intend* to solve the problem at hand? What was the idea behind the code? How does this code explicitly interrelate to other parts of the code? One of the greatest sins a developer can commit is to leave his code without his intent being clear to subsequent readers of the code.

**The Principle of Comments states:** Comment in full sentences in order to summarize and communicate intent.

## The Principle of Assumptions

---

The Principle of Assumptions is a corollary to the Principle of Explicitness. The practice of testing and documenting assumptions in your code has several benefits: one, it increases readability; two, it makes your code more predictable; three, it makes your code more maintainable; four, it reduces the need for comments; five, it makes your code more reliable; six, it makes your code more communicative when something goes wrong, thereby making your programs easier to troubleshoot; seven, early detection of failed assumption tests protects data from corruption; eight, it forces you to be aware of the

assumptions any given routine makes, and its relationship with other routines and shared data, which decreases the incidence of bugs.

The testing of assumptions is one of the cornerstones of defensive programming. Every piece of code contains assumptions. Some assumptions are fairly safe and don't need testing or documenting. (That said, our error handling scheme should always be there to gracefully catch when our untested assumptions do indeed fail.) However, we can and should test for (and document) the many other assumptions that are less obvious.

The most common kind of assumptions that should be tested (and thereby documented), are the prerequisites that a given routine relies upon. These tests usually take the form of a series of `If...Then` statements at the top of the routine. If any of the tests fail, then your code might either take corrective action, or perhaps raise a specific error message explaining that an assumption failed. (Another often overlooked method for testing assumptions is the use of assertions, which are `True/False` expressions that are usually only compiled into "debug" versions of the program for testing.)

**The Principle of Assumptions states:** Take reasonable steps to test, document, and otherwise draw attention to the assumptions made in every module and routine.

## The Principle of User Interaction

---

The phrasing for this rule or is borrowed from *About Face: The Essentials of User Interface Design*, by interaction design guru Alan Cooper. Cooper expands powerfully on this idea in his subsequent book, *The Inmates are Running the Asylum*: "Most software is used in a business context, so most victims of bad interaction are paid for their suffering. Their job forces them to use software, so they cannot choose *not* to use it—they can only tolerate it as well as they can. They are forced to submerge their frustration and ignore the embarrassment they feel because the software makes them feel stupid." (Page 34) This is a powerful statement, and one that should make us all pause to consider the real impact, good or bad, that our software has on real people.

How are users made to feel stupid? Alan Cooper has devoted two entire books to answering this question, and other author's have tackled the subject as well (including Donald Norman in his excellent book, *The Design of Everyday Things*.) Obviously, then, we cannot do the subject justice here. However, here is one simple example: a user clicks a button on a form and immediately an error message pops up that says "You cannot use that function right now." *Then why was the button even available for clicking?* The developer, by not taking the simple step of disabling or hiding the button, unwittingly creates a situation in which he or she is just a joker, pointing to a user's shirt and then flicking the poor user in the nose when he looks down. Very funny.

Poorly designed user interaction is a huge problem that applies to the entire hardware and software industry. It is a sad truth that developers throughout the hardware and software industry regularly design solutions that end up making their users feel stupid. It is even more sad that so many developers are blissfully unaware of the stress created in people's

lives. However, as developers, we are in a position to help solve this problem—one application at a time.

As software developers, our primary responsibility for user interaction lies in user interface design. Let's face it: in most cases there is not a detailed user interface design prepared in advance. Most user interface design decisions (and this includes report design) are made *by the developer at the time of construction*. Therefore, in most cases, it is *solely the job of the developer* to take steps to make sure the user is never made to feel stupid. The developer writes the error messages and prompts. The developer places the buttons and fields on the form. The developer, therefore, has almost total control over the user's experience.

**The Principle of User Interaction states:** Never make the user feel stupid.

## The Principle of Going Back

---

We've all been guilty of this one at one time or another: "I don't have time to do that now . I'll come back and do it later." This kind of procrastination usually applies to tasks such as commenting, code formatting, error handling, implementation of proper modularization, etc. Maybe you are that rare person who always goes back later and does all this "tedious" work, but most of us mortals never do.

The time to do all of the tedious tasks associated with coding is *at the time you are writing the code*. The main reason that no one goes back later to "clean up" their code is that any task that is tedious while you are in the middle of writing your code is monumentally more tedious when you have to go back and do it after the fact. Does anyone really think that going back and putting good error handling into hundreds of routines is *less* tedious than creating those routines with proper error-handling in the first place? Not only will you hate every minute of the task, you very well may introduce bugs that were not there before.

In the case of comments, the comments you add later will *never* be as good as the comments you could have written right at the moment you were writing the routine. And what happens if you have to leave an employer or project before you have a chance to "go back later"? Now you've left that monumental and tedious task for someone else to do, which is hugely unprofessional.

**The Principle of Going Back states:** The time to write good code is at the time you are writing it.

## The Principle of Other People's Time and Money

---

The Principle of Other People's Time and Money applies to all of the work product of a developer: code, reports, user interfaces, models, diagrams, test results, and documentation. The issue at stake for this rule is not just code, but rather professionalism and craftsmanship. Remember the rule that we started with? The Principle of Personal Character states: *Strive to make your code reflect only the best parts of your personal*

*character.* The Principle of Other People's Time and Money is a less friendly way of saying the same thing.

Take pride in your work, because your work is you, and other people will judge you based on your work—sometimes *solely* on your work. Even if you don't care about being judged, do you care about doing the right thing? Do you care about feeling good about taking money for writing that code? Do you care about how you and your work reflects on the profession of software engineering as a whole? Writing code for hire is not a game for our own amusement, and the quality of each developer's work reflects on all other developers.

**The Principle of Other People's Time and Money states:** A true professional does not waste the time and money of other people by delivering poor quality work.

###

Daniel Read is editor and publisher of the `developer.*` web magazine. He lives in Atlanta, GA, where he works as a software architect. He is currently at work on a book about software development crafted for a business audience.