# **10 Effective Ways to Become a Good Programmer**

### A good programmer is someone who always looks both ways before crossing a one-way street. -Doug Linder

**A good programmer is someone who always looks both ways before crossing a one-way street. ~Doug Linder**

Working as a software programmer in IT industry, one thing that drives us daily to the work place; is that fun and passion lies in programming. But to make that programming a fun and to get an eternal elation out of it, one needs to learn and adhere to some basics which make you a good programmer.

I am not writing mantras which you can follow to become a good programmer, but the intention is to collate a list of helping tips which I learned and implemented in the industry to get good results. There is no definition of a good programmer, but here we are referring to the category of programmer who have developed excellent IT solutions and helped in overall growth of this industry.

### ****1.Work on Basics****

As it is true for any industry and any job, the conceptual understanding is the key for success. Unless one has strong conceptual foundation, he/she can never be a good programmer. The core conceptual understanding helps you in designing and implementing the best solutions in the best possible way. If still you feel gap in core computer science and your programming language specific concepts, it’s never too late to go back and review the basics.

### ****2.Start putting question tags (how, what) with every set of code you write****

One thing that I realized creating a clear separating line between good programmer and rest is that zeal to know what and how it is happening. There is small group of people who can never leave a code without knowing exactly what is happening when it executes. I understand that in tight deadlines, we don’t get this liberty always and hence have to leave the code just knowing that it’s doing its job. Although this is a bit different topic of how to handle such situations, but as a programmer one can always try the level best to dig into as much as one can. And believe me, this becomes a habit with time and then you do it unknowingly every time.

### ****3.You learn more by helping others****

Most of us have a common tendency of turning our heads towards forums or groups only when we need help. And again a clear separation between the good programmer and rest that the formers visit these places more often to help others. This makes them learn more then they learn getting their problem solved by someone else. Within a team as well, help others to solve their problems. Believe me, understanding others’ problem in their context, investigating on that and providing solutions; will leave you much more learned than before.

### ****4.Write simple, understandable but logical code****

As in almost every aspect of life, the formula of KISS (Keep it simple and short) works in programming as well. Write more logical code and avoid complexity. Sometimes people do write complex code just to prove their capability to write such codes. My experience says that simple but logical codes always works well, resulted in fewer issues and are more extendable. I remember an excellent quote

**Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" ~Steve McConnell**

### ****5.Spend more time in analyzing the problem, you’ll need less time to fix it****

Spend more time in understanding and analyzing the problem and designing solutions for it. You will find the rest of the things quite easily doable. Designing not always mean using modeling languages and tools, it can be as simple as looking at sky and thinking solution in your mind. Those who have habits of pressing keyboard (for coding) the moment get the problem, usually ended us something different than the requirement.

**If you cannot grok the overall structure of a program while taking a shower, you are not ready to code it. ~Richard Pattis**

### ****6.Be the first to analyze and review your code****

Although a bit difficult, but try to break your own code before others can and with the time you will learn to write close-to-bug-free code. Always do a close and unbiased review of your code. Also never hesitate to take others view on your code. Working with good programmers and taking their feedbacks will surely help you become a good programmer.

### ****7.Don’t dismay yourself by looking at changing technology world****

Over these periods in IT industry, I met with many people who are either disappointed by their work or even left it to search new job saying they want to learn and work in latest technologies. I don’t see any problem with this aspiration but the very first incorrect word is the ‘latest technologies’. What we are hearing everyday and mean here is new tools, APIs, frameworks and others means coming up everyday to make the programming easier and quicker. This anyway will continue in technology world. But what needs to be understood is that the core and basic technologies changes with much lesser pace than frameworks, tools and APIs around it. This is like the sea where the surface water moves very rapidly but the deep water is relatively calm and concentrated and most of the aqua lives survive here. So, feel yourself in that deep water and close to core technologies. For e. g. in Java enterprise world, lots of web frameworks exist and new ones coming every other week. But the core concepts of request based client-server communication, MVS pattern, filters/servlets/JSP, resource bundling, XML parsing etc remains same. So spend more time in learning these core concepts rather than worrying about ever changing frameworks and tools around it. Believe me, with the foundation of core concepts, you will always find easier to learn new frameworks, tools and APIs.

### ****8.Work-arounds don’t work for longer time****

Many times software programmers implement work around solutions (may be because of lack of time, lack of problem understanding or lack of technology experience).But over the period these work around solutions always resulted in corrupting the code, making it less extendible and maintainable and lot of wastage of time later on. Always prefer to implement when you know the in-out of the solution. I understand that it becomes unavoidable in some circumstances, but it’s like, one should speak truth always but you tell lie in some circumstances.

### ****9.Read documentation****

One of the essential habits of good programmer is that they read lots of documentation. May it be specifications, JSR, API documents, tutorials etc. Reading documents helps you creating that essential foundation based on which you program in best of the way.

### ****10.You can learn from others code as well****

I interacted with some excellent programmers who actually have java source project inside their IDE all the time and read/refer that in daily work. They do it not only to fulfill their appetite of knowing the basics but also to learn ways of writing good programs. Reading and referring reliable and known open source code or your senior’s code, can also help you making your programming better.

### ****And the last, not listed above: Don’t compare yourself with others****

Your comparison of yourself with others will only result in evolution of negative feelings and un-healthy competition. Everyone has got his or her strengths and weaknesses. It is more important that we understand ours and work on it. I have seen many times that so called ‘fundoo-programmers’ (fundamentally strong programmer) also make silly mistakes. So, analyze yourself, list down your areas of improvement and work on it. Programming is a real fun, enjoy it.

**Any fool can write code that a computer can understand. Good programmers write code that humans can understand. ~Martin Fowler**

# 15 Rules for Writing Quality Code

Diomidis Spinellis, author of [Code Quality: The Open Source Perspective](http://www.informit.com/store/code-quality-the-open-source-perspective-9780321166074), lists the 15 most important rules for writing sparkling code. Follow them, and your code will look professional, live long, grow smoothly, and earn your colleagues’ love (rather than swearing).

There are myriad ways of writing poor code. Thankfully, rising to the level of writing quality programs involves just 15 rules. Following them won't make you a master programmer, but will allow you to fake one quite convincingly.

### Rule 1: Follow the Style Guide

Every programming language has a style guide that tells you in great detail how to indent your code, where to put spaces and braces, how to name stuff, how to comment—all the good and bad practices. For example, the style guide tells you the 12 mistakes lurking in this code snippet:

for(i=0 ;i<10 ;i++){

Read the guide carefully, learn the basics by heart, look up corner cases, apply the rules religiously, and your programs will be better than those written by the majority of university graduates.

Many organizations customize style guides to reflect the organization's specific practices. For instance, Google has developed and released style guides for more than a dozen languages. These guides are well thought out, so [check them out](https://code.google.com/p/google-styleguide/) if you're looking for help programming for Google. Guides even include editor settings to help you apply a programming style, and custom tools can verify that your code adheres to that style. Use these tools.

### Rule 2: Create Descriptive Names

Constrained by slow, clunky teletypes, programmers in the past used to contract the names of their variables and routines to save time, keystrokes, ink, and paper. This culture persists in some communities, in the name of backward compatibility; consider C's tongue-twisting wcscspn (wide character string complement span) function. But there's no excuse for this practice in modern code.

Use long descriptive names, like complementSpanLength, to help yourself, now and in the future, as well as your colleagues to understand what the code does. The only exception to this rule concerns the few key variables used within a method's body, such as a loop index, a parameter, an intermediate result, or a return value.

Even more importantly, think long and hard before you name something. Is the name accurate? Did you mean highestPrice, rather than bestPrice? Is the name specific enough to avoid taking more than its fair share of semantic space? Should you name your method getBestPrice, rather than getBest? Does its form match that of other similar names? If you have a method ReadEventLog, you shouldn't name another NetErrorLogRead. If you're naming a function, does the name describe what the function returns?

Finally, there are some easy naming rules. Class and type names should be nouns. Methods names should contain a verb. In particular, if a method returns a value indicating whether something holds true for an object, the method name should start with is. Other methods that return an object's property should start with get, and those that set a property should start with set.

### Rule 3: Comment and Document

Start every routine you write (function or method) with a comment outlining what the routine does, its parameters, and what it returns, as well as possible errors and exceptions. Summarize in a comment the role of each file and class, the contents of each class field, and the major steps of complex code. Write the comments as you develop the code; if you think you'll add them later, you're kidding yourself.

In addition, ensure that your code as a whole (for example, an application or library) comes with at least a guide explaining what it does; indicating its dependencies; and providing instructions on building, testing, installation, and use. This document should be short and sweet; a single README file is often enough.

### Rule 4: Don't Repeat Yourself

Never copy-and-paste code. Instead, abstract the common parts into a routine or class (or macro, if you must), and use it with appropriate parameters. More broadly, avoid duplicate instances of similar data or code. Keep a definitive version in one place, and let that version drive all other uses. Following are some good examples of this practice:

* Creation of API reference guides from comments, using Javadoc or Doxygen
* Automatic detection of unit tests through an annotation or a naming convention
* Generation of both PDF and HTML documentation from a single markup source
* Derivation of object classes from a database schema (or the opposite)

### Rule 5: Check for Errors and Respond to Them

Routines can return with an error indication, or they can raise an exception. Deal with it. Don't assume that a disk will never fill up, your configuration file will always be there, your application will run with the required permissions, memory-allocation requests will always succeed, or that a connection will never time out. Yes, good error-handling is hard to write, and it makes the code longer and less readable. But ignoring errors and exceptions simply sweeps the problem under the carpet, where an unsuspecting end user will inevitably find it one day.

### Rule 6: Split Your Code into Short, Focused Units

Every method, function, or logical code block should fit on a reasonably-sized screen window (25–50 lines long). If it's longer, split it into shorter pieces. An exception can be made for simple repetitive code sequences. However, in such cases, consider whether you could drive that code through a data table. Even within a routine, divide long code sequences into blocks whose function you can describe with a comment at the beginning of each block.

Furthermore, each class, module, file, or process should concern one single thing. If a code unit undertakes diverse responsibilities, split it accordingly.

### Rule 7: Use Framework APIs and Third-Party Libraries

Learn what functionality is available through an API in your programming framework, and also what's commonly available through mature, widely adopted third-party libraries. Libraries supported by your system's package manager are often a good bet. Use that code, resisting the temptation to reinvent the wheel (in a dysfunctional square shape).

### Rule 8: Don't Overdesign

Keep your design focused on today's needs. Your code can be general to accommodate future evolution, but only if that doesn't make it more complex. Don't create parameterized classes, factory methods, deep inheritance hierarchies, and arcane interfaces to solve problems that don't yet exist—you can't guess what tomorrow will bring. On the other hand, when the code's structure no longer fits the task at hand, don't shy away from refactoring it to a more appropriate design.

### Rule 9: Be Consistent

Do similar things in similar ways. If you're developing a routine whose functionality resembles that of an existing routine, use a similar name, the same parameter order, and a comparable structure for the code body. The same rule applies to classes: Give the new class similar fields and methods, make it adhere to the same interface, and match any new names with those already used in similar classes. Make the order and number of case statements or if else blocks follow the corresponding definition in the specifications you're using. Keep unrelated items in alphabetical or numerical order.

Your code should adopt the conventions of the framework in which you're programming. For instance, it's common practice to represent ranges half-open: closed (inclusive) on the left (the range's beginning), but open (exclusive) on the right (the end). If there's no a convention for a particular choice, establish one and follow it religiously.

### Rule 10: Avoid Security Pitfalls

Modern code rarely works in isolation. Therefore it will inevitably risk becoming the target of malicious attacks. They don't have to come from the Internet; the attack vector could be data fed into your application. Depending on your programming language and application domain, you might have to worry about buffer overflows, cross-site scripting, SQL injection, and similar problems. Learn what these problems are, and avoid them in your code. It's not difficult.

### Rule 11: Use Efficient Data Structures and Algorithms

Simple code is often more maintainable than equivalent code hand-tuned for efficiency. Fortunately, you can combine maintainability with efficiency by utilizing the data structures and algorithms provided by your programming framework. Use maps, sets, vectors, and the algorithms that work on them, and your code will be clearer, more scalable, faster, and memory-frugal. For example, if you keep a thousand values in an ordered set, a set intersection will find the values common with another set in a similar number of operations, rather than performing a million comparisons.

### Rule 12: Include Unit Tests

The complexity of modern software makes it expensive to deploy a system and difficult to test it as a black box. A more productive approach is to accompany every small part of your code with tests that verify its correct function. This approach simplifies debugging by allowing you to catch errors early, close to their source. Unit testing is indispensable when you program with dynamically typed languages such as Python and JavaScript, because they'll only catch at runtime any errors that that a statically typed language such as Java, C#, or C++ would catch at compile time. Unit testing also allows you to refactor the code with confidence. You can use an [xUnit framework](https://en.wikipedia.org/wiki/XUnit) to simplify writing these tests and automate running them.

### Rule 13: Keep Your Code Portable

Unless you have some compelling reason, avoid using functionality that's available only on a specific platform or framework. Don't assume that particular data types (such as integers, pointers, and time) are of a given width (for example, 32 bits), because this differs between various platforms. Store the program's messages separately from the code, and don't hardcode cultural conventions such as a decimal separator or date format. Conventions need to change when your code runs in other countries around the world, so keep this adaptation as painless as possible.

### Rule 14: Make Your Code Buildable

A single command should build your code into a form that's ready for distribution. The command should allow you to perform fast incremental builds and run the required tests. To achieve this goal, use a build automation tool like [Make](https://en.wikipedia.org/wiki/Make_(software)), [Apache Maven](http://maven.apache.org/), or [Ant](http://ant.apache.org/). Ideally, you should set up a continuous integration system that will check, build, and test your code every time you make a change.

### Rule 15: Put Everything Under Version Control

All elements of your system—code, documentation, tool sources, build scripts, test data—should be under version control. [Git](http://git-scm.com/) and [GitHub](http://github.com/) make this task cheap and hassle-free, but many other similarly powerful tools and services are available. You should be able to build and test your program on a properly configured system, simply by checking out the code from the repository.

## Summary

By making these 15 rules part of your everyday programming practices, you'll eventually create code that's easier to read, thoroughly tested, more likely to run correctly, and much simpler to revise when that time comes. You'll also save yourself and your program's users a lot of headaches. For detailed examples of both good and bad code, check out my book [Code Quality: The Open Source Perspective](http://www.informit.com/store/code-quality-the-open-source-perspective-9780321166074), which provides hundreds of examples taken from open source projects.