# How To Write Code and Comment Well

“Programs must be written for people to read and only incidentally for machines to execute.”

* Hal Abelson, Structure and Interpretation of Computer Programs

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

* Brian Kernighan

“The ratio of time spent reading (code) versus writing is well over 10 to 1 … (therefore) making it easy to read makes it easier to write.”

* Robert C. Martin

Your code needs to be as simple as possible, but no simpler. Simple doesn’t mean (for example) not handling errors or unusual cases. Simplicity comes from organization, appropriate abstraction, lack of clutter, good naming, useful comments, and clear thinking.

## General Advice

Writing code is a lot like writing instructions to do something. Clarity is the **most** important thing. The easier it is to understand your steps, the better for humans, and, surprisingly, for the computer, too! Your first draft will rarely be the clearest it should be. This means that your first draft should not be your final draft. Go back and revise/edit. You might be tempted to think “it’s **just** an assignment for school. It doesn’t matter.” You learn what you practice. School is your practice time. You **can’t** practice being sloppy and unclear and then decide that, suddenly, clear and concise matters. Develop a sense of and a love for clear code.

Don’t try (yet) to imagine the steps the computer is taking to implement your instructions and micro-manage it. That will come, after a lot more experience. Also note that most of the time, performance doesn’t matter.

Sometimes, you will know exactly how to write something. When you don’t, talk or write (see Comments below) your way through the design. Explain it to a person in your life. Some people use stuffed animals or rubber ducks. Seriously, it helps. A lot.

## Naming

Naming is one of the hardest but most critical aspects of Computer Science because we are building a mental model that we are sharing with other people, expecting them to be able to read and understand what we wrote. Of course, we don’t all come from the same background and experience; this makes the explanation of the mental model more critical but also more difficult.

Be consistent when it makes the code clearer. Pick a name for a “thing” and use that name consistently. A bad example:

float getTaxPercentage {

return taxRate;

}

float taxValue = getTaxPercentage(); // value != percentage != rate

Characters are cheap; don’t use short names where longer names would reduce confusion. We have all been told “don’t use single letter variables”. But there is more nuance to naming. If I name something “users”, is that a list of users? The number of users? The names of the users? The allowed users? We have no idea. Data types can be helpful, of course, if we saw:

List<user> users;

We would have some more idea what was happening, but we wouldn’t know if it was all users, current users, allowed users, etc. Of course, context is helpful too:

users = getCurrentUsers();

But we would have to go and **find** this line to get the context. What if we changed the name?

currentUsers = getCurrentUsers(); // This is much easier to understand!

Don’t embed data types in your names (example: numberOfUsersInt or stringUserNames).

Don’t use abbreviations unless they are super common.

Good: getUSATaxRate, findURL

Bad: FName, LName, StdntNm

Don’t be cute, funny, or offensive. Naming your variables “Henry” or “HarryPotter” or “MyBossIsAJerk” does not help others understand your code (only your pain). Also try to avoid slang, concepts that don’t cross cultures and trendy names.

Don’t use filler words – words that don’t add understanding to your names.

Bad: MathHelper, Utils, ComplexStuff, UserManager, WorkHandler

Methods should typically be verbs if they modify state.

Example: AddUser, ShuffleCards, StopMusic

Methods should be Is, Has or Can (or something similar) if they are checking something.

Example: IsOdd, IsEven, CanAddUser, HasWeight, CanLift

Variables and members should be nouns.

Avoid using the same variable name in different scopes (for example, having a variable “score” in the class and as a parameter to a method).

Follow your languages common standards. Each language has common rules; following them makes your code more familiar to others who work in that language.

<https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>

One exception to longer, descriptive names is loop indices. Using “i” or “x” is very common and acceptable.

There is no “magic formula” for naming except that it should be clear to people who are fellow programmers and have some understanding of the domain that you are working in.

## Use the Right Level - Functions

There are many ways to accomplish many tasks in programming. You should try to use the simplest way to perform any given task that expresses what you are doing. For example:

int addTwoNumbers (int x, int y) {

for (int i = 1; i<y; i++)

x++;

return x;

}

This is a very obtuse way to replace:

int addTwoNumbers (int x, int y) {

return x+y;

}

And it has bugs (what happens with negative numbers?)! Of course, you shouldn’t even really have a function to add two numbers!

Your code should endeavor to be as easy to read as possible. Use the built-in functions as much as you can to make your code shorter and clearer.

For example:

File myObj = new File(fileName);

Scanner myReader = new Scanner(myObj);

LinkedList lines = new LinkedList();

while (myReader.hasNextLine()) {

lines.add(myReader.nextLine());

}

myReader.close();

Compared to:

List<String> lines = Files.readAllLines(Paths.get(fileName),

Charset.defaultCharset());

Notice here that the shorter version is **clearer** and more concise! It is certainly possible to make shorter code that is **less** clear and **less** easy to read:

Bad Example:

a^=b^=a^=b; // swaps two int values;

In short, our goal is **clear**, not **short**, but short may well be clearer.

Much like using the built-in functions to make your code clearer, you should create your own functions to make your code clearer. We do this in natural languages all the time – we create words that represent complex concepts. “Car” is nothing more than a shortened version of “a powered four wheeled passenger vehicle”. Your functions should represent an action (or a question) that makes sense in the mental model of your code. Dividing your code into named pieces allows you to express your algorithm without having an excessive number of details. When you don’t have enough functions/methods, you end up with a few very long methods that are hard to read and reason about. When you have too many functions/methods, the code becomes more difficult because you are introducing new “vocabulary” that is better expressed in-line.

Bad Example: int AddOne(int i) { return i+1;}

Other benefits of splitting code into functions – there are fewer variables in scope, debugging is easier, testing is easier, keeping all the details of what you want to accomplish in your head at once is easier and, of course, you get pieces of code that you can reuse.

Typically, a one-line function is a bad idea. The exception might be accessors/mutators or cases where you want to encode something tricky or complex that you reuse.

Good Example:

bool isLeapYear(int year) {

return ((year%4 == 0) && (year % 100 != 0)) || (year % 400 == 0);

}

So, what about 2-line functions? Three lines? Where is the dividing line? There is a mental “cost” of going to look up a function. Much like looking up a new word in the dictionary, it can take away from the reading of the code. On the other hand, repeated code and tracking details in the middle of code makes it harder to read. This is really a judgement call, and sometimes good, skilled people will disagree. Reading other people’s code can give you some insight into what good (and bad!) code looks like.

One trap to avoid is configurable functions. Consider you have a function to delete a user from a system.

boolean deleteUser(int userId)

Now you get a requirement to mark a user as not valid, but not delete them. You might be tempted to add a flag:

boolean deleteUser(int userId, bool makeThemInvalid)

The problem with this is that you might get a request to make the user invalid after some date:

boolean deleteUser(int userId, bool makeThemInvalid, Date invalidateDate)

You can see that this function is going to get more complex and more “flag driven”. Break it up instead of adding more configuration options.

Start to think about your code as having different levels of design/abstraction. You want to split high-level and low-level details into different layers. Typically, a function should not span multiple layers.

For example – imagine that you are writing Zoom. You wouldn’t have a function that updates the screen (low level) and does account management (high level) and listens to the microphone (low level).

## Use the Right Level - Functions

I mentioned “fewer variables in scope” – this is something that a lot of people struggle with. “Where do I put my variables?” When should something be a variable inside a function/method vs. an instance member variable vs. a static member variable? The answer is that a variable should be “as local as possible, but no more.” If a variable is not used outside of your function/member, then it should be only a local variable. Better yet is a variable scoped into a block, like this:

void someFunction(int input) {

if (input == 0) {

int calculation = 5;

// do something

}

}

The variable calculation is only valid/valuable inside the if block. This keeps your program “clutter-free” and makes finding bugs easier, since you have fewer places to look for “who changes this?” That same reasoning applies to other scope decisions. If your variables are too **broadly** scoped (members instead of locals), you will find yourself looking all over to try to figure out where all the places that change the variable are. On the other hand, if your variable is too **narrowly** scoped, you will find yourself passing that variable around between functions/methods a lot. Member variables should be logical – the object can’t “be” the object without that. For example – a car has an engine, so this makes sense:

public class Car {

private Engine engine;

}

But would you add this:

public class Car {

private Engine engine;

private Person[] passengers;

}

Probably not – passengers are not an inherent trait of cars. You might make them a parameter to a method, though:

public void goForADriver(Person driver, Person[] passengers) {}

Static members should be used only rarely – when a value should be shared across all instances of a class. The classic example for this is this:

public class someObject {

static int numberOfSomeObjects = 0;

public someObject() {

numberOfSomeObjects++;

}

}

In practice, you almost never need this. Static member variables are almost always the wrong choice.

## Comments

Text

Description automatically generatedIn introductory courses (ICSI201, for example), you might have been asked to comment every line with an explanation of what you are doing. This is a good teaching/learning tool. This is not the correct way to comment your code in the future. Excessive comments are **worse** than none! Why? Comments take time to write and maintain (that is, change when your code changes). Comments that restate what the code does take twice the space and convey no more information.

Example:

x++; // increment x

How, then, should we comment?

We should start with the things that we have already mentioned – name your functions/methods well and use **and** create appropriate levels of functions/methods. Good comments don’t fix bad code. Your comments should clarify the “why” of the code. Why are you incrementing x? Your comments should answer the reader’s question of why you are doing this piece of code if it isn’t stunningly clear to a competent reader. For example:

for (int i=1;i<max;i++)

print (a[i]);

A bad comment for this might read:

Loop from 1 to max, printing each element in a.

But, if the reader knows the language, that’s obvious. There are 2 questions here – why are we starting with 1, not 0 (which is how loops are typically done) and why are we printing a whole array? A better comment might be:

Print the array for debugging. Skip the first element (it will always be uninitialized).

Now I know why we are doing it **and** why we skip the first one. Another situation is when you implement some algorithm, if you have good source material, quote it:

// Implement the Fisher-Yates shuffle :

// https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates\_shuffle

void shuffle(int[] a)

You can assume that the reader knows super-basic things. The rule of thumb is that if you think that you and most of the people around you know this algorithm, you don’t have to include a reference. Incidentally, this “rule” can also apply to non-technical algorithms. For example:

// Implement Albany County Taxation:

// <https://www.salestaxhandbook.com/new-york/rates/albany>

Another good example of when to reference is when you have a specification:

// Per section 5.4 of version 1.2 of the XYZ specification

If I can’t complete code for some reason, I will write a “to do” item so that I remember what I was thinking when I can get to it. This lets me “forget” and focus on other things.

// TODO: Shuffle the data so that the user gets a different hand every game

Notice the “TODO:” – if you use this or something similar in all of your programs, you can use a text search easily.

Another good case for a comment is when you have to work around a bug in some other piece of code. Example:

// On Windows 7, failure is -1; on Windows 10, failure is 0

if (didSucceed == 0 || didSucceed == -1) {

Often, when I have a complicated function to write, I will start by writing the comments and explain what I will do (in broad terms), then I will go back and fill it in. I will leave the comments as summaries of the code.

Example:

// Open the file and read the contents into an array of strings, then close the file

// For each line in the array, split out the numbers (comma separated) and put them in an array

// Find the min, max, mean, median and mode of the numbers in the array

// Populate the XYZ data structure and return it.

I find this approach to be very helpful when I am thinking about something complex. It also allows me to skim the code because the comments summarize the code. But I only do this when the function is complex (but doesn’t make sense to break up).

Some things that I think that you should avoid:

Big header blocks that have method name, purpose, and the description of every variable in and out. Some people favor these. I think that they are a waste of time (to write/maintain) and space (more scrolling).

Example:

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \* CalculateHypotenuse \*

// \* Takes two sides of a right triangle and \*

// \* compute the hypotenuse by using the \*

// \* Pythagorean Theorem \*

// \* \*

// \* Parameters \*

// \* int x – one side of the triangle \*

// \* int y – the second side of the triangle \*

// \* \*

// \* Returns \*

// \* int – the hypotenuse of the triangle \*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

All of that for a one-line function! The big block, the trivial explanations, the time (and effort) to format it, the time to scroll by it, the time to read it later and effort to maintain it is all waste.

Likewise, copyright notices and code license information does not belong in comments, in my opinion. I am not a lawyer, though.

A picture containing text, sign, stop, grass

Description automatically generated

JavaDoc (you know - @param, @returns etc.) is, essentially, comments. The question that you have to ask is “will I do anything with it?” If you are making classes that will be used by some other team or (more importantly) someone outside of your organization, it makes a lot of sense to use JavaDoc to maintain your documentation for your interface. Often, though, I have seen JavaDoc used as a “replacement” for the big header block. It’s the same information! If it didn’t make sense in a big block, why does adding @ tags make it better? When **should** you use JavaDoc? When you plan to generate a document on how to use your class. For example – you are building a library to do some work. You expect others (especially outside of your organization) will use that library.

Avoid keeping a bug history in your programs. I know many people and companies advocate for doing this. There are a few problems with it:

1. You end up with a history of your failures and embarrassment on display for everyone to see.
2. Past bugs have been fixed when found. Knowing that there was an off-by-one error 10 years ago is not useful.
3. On a long-lived code base, this is a HUGE amount to scroll through.

Avoid funny, clever comments. Besides possible offence to your coworkers who don’t share your sense of humor, it is extra mental work to process for people who are trying to understand. Save it for your standup routine.

## White Space

White space, in programming is contentious – different people will have different rules and preferences. There are some, though, that everyone agrees on:

1. Always leave one (and only one!) blank line between methods.
2. Don’t leave blank lines for no reason (other than 1). Some good reasons:
   1. I want to divide between public/private/protected members
   2. I want to delineate between two sections of my code

void processData()

// some code to read a file

…

// some code to process the records

…

// some code to write the results to another file

..

1. Always indent your code to show scope:

class example {

public int counter = 0; // notice this is indented!

public void doWork() { // same indentation as counter

while (condition) // indented because inside method

counter++; // indented because loop body

There is a perpetual debate between using spaces and tabs for indentation. You can find the discussions easily on the internet. The one thing that everyone agrees on – use whatever your organization uses and be consistent.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Poor (0%) | OK (50%) | Good (80%) | Great (100%) |
| Consistent Names  (5) | More than 5 inconsistencies | 5 or fewer inconsistencies |  | No inconsistencies |
| Expressive Names (10) | More than 5 inexpressive names | 5 or fewer inexpressive names |  | No inexpressive names |
| Abbreviations  (5) | More than 5 abbreviations | 5 or fewer abbreviations |  | No abbreviations |
| Appropriateness  (-10 or 0) | Inappropriate |  |  | Appropriate |
| Methods are verbs (10) | More than 5 methods not verbs | 5 or fewer methods not verbs |  | All methods are verbs |
| Variables are nouns (5) | More than 5 variables not nouns | 5 or fewer variables not nouns |  | All variables are nouns |
| Methods at the right level of abstraction (30) | Very few methods or far too many methods | A few big methods or many tiny methods | Mostly right level | Absolutely correct. |
| Why, not what comments (5) | Very few comments or way too many | Many what comments | A few what or missing some why | Correctly commented |
| White Space between methods (5) | Many repeated blank lines or no blank lines | 5 or fewer cases of space between method issues |  | Correct white space between methods |
| Delineation  (5) | No blanks between “concepts” or excessive | 5 or fewer cases of inappropriate delineation |  | Correct delineation |
| Indentation (10) | Mostly incorrect indentation | 5 or fewer cases of incorrect indentation |  | Correct indentation |