Project Coding Guidelines

Peter Schachte

March 25, 2019

> *Commenting your code is like cleaning your bathroom — you never want to do it, but it really does create a more pleasant experience for you and your guests.*
> — Ryan Campbell

Of course, your project submissions should closely follow the project specification. But that is not enough to get full marks for a programming project — programs are also assessed on how well-written they are. In the real world, programs must be *maintained*, which means they must be read and understood by people other than the original author.

Try taking the cold reader test: pretend you have not seen your program before and consider how easy it is to understand. One good way to improve your coding is to make a habit of reading other people's code. Learn what makes code a pleasure to read and what makes it a chore.

This document lists a few factors that contribute to code quality. It is intended to be programming language independent, so we use the term *operation* to refer to whatever operational abstractions are provided by the language, whether they be procedures, functions, methods, predicates or even macros.

Each of these guidelines has exceptions, but you should know when you are violating them, and have a very good reason.

1 Documentation

Probably the most important factor in code quality is how well documented your code is. By "documentation" here we refer not to supporting documentation for your project, such as a System Requirements Specification. Nor are we referring to end-user documentation, explaining how to use the code. Such documentation is important, but it lies outside the scope of this discussion. Here we refer to the documentation embedded in the code as comments.

1.1 Sign your work

At or very near the top of every source code file should be a few lines that identify the programmer(s) and the file. It should contain your name, your login id, and a one or two line summary of the purpose of the file. This isn't bragging, it's admitting culpability for your mistakes.

1.2 Put first things first

After this, the next thing to appear in a source file should explain the purpose of the file in greater depth. It does not need to cover every detail, but should well prepare the reader to read the code. It should explain the role of that file in the overall program and any important assumptions the implementation makes. For the main program file, it should also explain the purpose of the program and the problem it solves. Do not assume the reader knows what the code is for.

1.3 Explain your data

It is also important to document the major data structures declared in each file, outlining how each is used, preferably next to its declaration. For an untyped language, where data types are not declared, this documentation should be placed where you would put the declaration if there were one. Do not duplicate this documentation everywhere the type is used.

1.4 Explain your operations

Each major operation should also be documented. This must include any assumptions made by the implementation (other than what is made clear by the types), as well as any changes it makes to its arguments or other data. It is common to put this documentation in front of the code it documents, but some programming languages may suggest an alternative convention for this.

1.5 Anticipate questions

Recalling the cold reader test, file documentation should try to answer questions a reader might have before she has them. For example, if a particular algorithm has some subtleties, they should be documented before the subtle code. Where the correctness of a piece of code relies on some non-obvious analysis, that should be made clear. Similarly, important properties of variables that hold at a particular point in the code through all iterations ("loop invariants"), and indeed anything non-obvious that is important to understand the code, should be documented.

1.6 Make it pretty

A program need not be a work of art, but spending a little time tidying it up can make it much more readable. Lines should not be more than 79 columns, or they will probably wrap around when it is printed, making the code very difficult to read.

You should assume tab stops will be placed every 8 columns, this is the usual default. That usually means that indenting nested code by a full tab stop for each level of nesting will make it difficult to avoid violating the 79 column limit.

1.7 Be consistent

Establish conventions for code layout, commenting style, identifier naming, *etc.*, and stick to it. Being consistent with your style is more important than the actual style you choose. This is more difficult, and more important, when the code is developed by different programmers. Explicitly establishing coding conventions at the start of a team project can save a lot of trouble later.

1.8 Do not be repetitive and redundant

Do not document anything that should be obvious to anyone who knows the programming language. Usually code is more succinct and precise than any documentation you could write. Documenting the obvious makes the program *harder* to read, because it makes it longer without making it clearer. High level documentation is much more important than documentation of individual lines of code.

2 Code

Coding style varies from language to language, but some principles apply to most languages.

2.1 Organise!

Code and data declarations should be presented in a file in the same way a journalist presents an article: the most important part comes first, with details presented later. A long program file, like a long article, should be divided into sections, where each section covers a particular aspect of the overall picture. Each section of code should include an introductory comment visually separating it from the previous section, and briefly describing its purpose. This puts closely related code close together, and establishes a context for each chunk of code. It also makes the program easier to develop and maintain, as it makes it easier to find code you have already written and need to modify.

Some languages permit you to define operations in any order, making it easier to organise your code as you would like. For other languages, you just have to do your best using forward declarations or other language-specific features to organise the code understandably.

2.2 Abstract!

If you find yourself writing similar code repeatedly, you should look for a way to abstract it into a separate piece of code that can be invoked from many places in your code. Use parameters to allow different uses. This has several advantages over copy-and-paste programming: the code is easier to understand because it is clear how different uses differ (it is much harder to see a small difference between two 20 line code sequences than between two 1-line function calls). Since the code only appears in one place, this also makes it much easier to make changes to the code.

There are many approaches to choosing the best abstractions to use for your task; we cannot cover them adequately here. One approach that may help is to try to choose the abstractions that are the simplest to explain.

2.3 Make code self-documenting

Each operation should have a well-defined purpose, and its name should reflect that purpose. This may affect the way you abstract a particular chunk of code. Names should also be chosen to distinguish similar functionality; for example, `biggest_region` and `largest_region` do not convey a distinction, whereas `tallest_region` and `widest_region` do.

The same is true of constants. Do not litter your source code with constants such as 12 or 7, instead define symbolic constants such as `MONTHS_PER_YEAR` and `DAYS_PER_WEEK`. Note that the names must be meaningful; using `DOZEN` in place of 12 is not helpful. It is generally not necessary to define symbolic constants for values with obvious meanings such as 0 or 1, nor where there is no useful name, such as the 2 and 4 in the quadratic formula.

A well-chosen name is better than a comment, because it appears everywhere it is used.

2.4 Be brief

Names need not be long. When a name is too long, code can become difficult to indent properly. Often you can rephrase, abbreviate, and omit words from a name, without sacrificing clarity.

By the same token, individual operation definitions should not be too long. It is difficult to understand a chunk of code when you cannot see it all at once, so it is best to keep individual definitions to less than 50 lines. If a definition gets longer than this, abstract out key parts of it into separate operations, if possible.

2.5 Be consistent

Be consistent in your coding, particularly in choosing identifiers. For example, do not call a line count `line_count` in one place in your code, and `num_lines` in another. Use the same variable names for the same kinds of entities throughout your code. Use the same abbreviations and capitalisation everywhere.

Many languages have established conventions for naming, capitalisation, and even code layout. It is best to observe these conventions, even if you do not like them, because it makes your code easier to read for readers who know the language conventions. Similarly, different languages have idiosyncratic ways to do things, and you should follow those conventions where appropriate. Where there are no widely agreed conventions, feel free to follow your own taste, but do so consistently.

2.6 Use the right tool for the job

Use the most appropriate language construct, primitive, or library operation for the purpose at hand. Do not re-invent the wheel — if an operation similar to what you need is already defined, use it rather than writing your own.

2.7 Keep it simple

Try to find the simplest, most succinct way to implement your operations. If performance or other considerations force you to complicate the definition, so be it. But it is usually worth trying the simplest approach first — you may be surprised how effective it is.

3 Assessment Criteria

We use the following criteria to assess your program:

1. Quality of file-level documentation

Each file should begin with a comment identifying the author and the purpose of the file. Students are not required (or encouraged) to include their student number. For example, for a simple program, a file header comment like the following should receive most or all of the marks for this criteria:

```
// Author:   Peter Schachte <schachte@unimelb.edu.au>
// Purpose:  Play rock-paper-scissors (RPS) against a human opponent
//
// We repeatedly play RPS against the user.  For each round of the
// game, the user enters one of "rock", "paper", or "scissors" to
// indicate their guess.  The computer then chooses one and decides
// the winner (paper beats rock, scissors beats paper, rock beats
// scissors).  If user and computer guess the same, it's a tie.  The
// computer than prints the result of the round and begins the next
// round.  The user quits by entering "quit" instead of a guess.
// Finally, the computer prints a summary of the results.
```

For a more complex project, more text would be needed, but not every detail needs to be included here. For example, this comment does not explain what kind of summary is printed at the end. Each detail should be explained in the part of the code responsible for that detail. In a multi-module project, each non-main module should begin with a comment explaining the purpose of that module.

2. Quality of function- and type-level documentation

Each operation (except possibly trivial helpers) and each type should have a description: at least one or two sentences describing what it produces or accomplishes. Any operations whose behaviour or correctness is complex should have a more thorough explanation. It should be possible to understand the purpose of each item from only the name and documentation, without having to read the code.

3. Readability of code

The code should be easy to read if printed out in 80 column lines. Wrapped lines will cost marks. Code should look neat whether displayed with 4 or 8 column tab stops. Code should be well organised and easy to navigate. Any file more that a couple hundred lines should be broken up into separate chunks of code, each devoted to some aspect of the file, making it easy to find one's way around. There should be no useless comments.

4. Understandability of code

The code itself should be easy to understand, with easily understood names for variables, constants, and operations. Any subtleties in the code should be explained. The logic of each piece of code should be clear either from the code itself or from the accompanying comments.

5. Appropriate abstraction

Each variable, constant, and operation should have a single purpose. Each purpose should have a single variable, constant, or operation that is responsible for it. Each design decision should appear in only one place in the code, to the extent permitted by the language. Each piece of code should be as simple and clear as possible (given any performance requirements). There should be no repeated chunks of code, and no single operation definition should be longer than about 50 lines if it can reasonably be avoided. Shorter is better.

6. Good use of language and libraries

The program should not implement operations supported by the library (unless required by the project specification). It should make the best possible use of the features of the language (to the extent they have been covered in the subject).

4 Rubric

Each criterion will be assessed on this scale:

- **5** → outstanding; practically flawless
- **4** → very good, but not perfect
- **3** → passable, but not great
- **2** → unsatisfactory, but on the right track
- **1** → very poor
- **0** → no effort

In general 5 is reserved for an unusually good job for that criterion, 0 for no sign of any work toward that criterion, and 1 for a particularly poor job. The majority of assessments are expected to fall between 2 and 4.