Project Coding Guidelines

Peter Schachte

July 17, 2008

> *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 departmental login id, and a one or two line summary of the purpose of the file. If you are using a version control system such as CVS or Subversion, it should also include the magic keyword to show the file date and version. This is very useful in distinguishing printouts of different versions of the file.

1.2 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 What kinds of 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 a typeless 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 What does it do?

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 they 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 Consistency

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 Department of redundancy department

Don't document anything that should be obvious to anyone who knows the programming language reasonably well. 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 Coding

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've 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'd 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 differences among the uses. This has several advantages over cut-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 Self-documenting code

Each function, method, or predicate 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 clear comment near the definition, because it necessarily appears everywhere it is used.

2.4 Brevity is the soul of coding

Names need not be terribly 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 60 lines. If a definition gets longer than this, abstract out key parts of it into separate operations.

2.5 Consistency

Be consistent in your coding, particularly in choosing identifiers. For example, don't 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 programming languages have established coding conventions for naming, capitalisation, and even code layout. It is best to observe these conventions, even if you don't like them, because it makes it easier to read your code 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 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's usually worth trying the simplest approach first — you may be surprised how effective it is.