CSE 560 **Winter 2005**
Prof. Paul Sivilotti

Writing A Programmer's Guide

Distributed: Friday, January 7th.

Almost invariably, someone (perhaps the original authors) will need to modify the program. This may be because the original requirements change, some extra functionality is needed, or repairs are required for mistakes made in the original implementation. The programmer should be able to turn to the *Programmer's Guide* for the information they need.

Thus, the programmer's guide should clearly describe to the prospective reader *how your program works*. It should describe your program concisely so that when the reader looks at the program, he/she knows **where** to look for a particular structure, variable, function, error condition, etc.

The reader needs to find out fairly quickly how much work the change will take. Having to turn immediately to a long program listing will discourage our reader, and probably will result in your program being set aside in favor of another (and someone else getting credit for writing a versatile program), or will result in an unnecessary new system which will be costly and wasteful of resources. Instead, the programmer's guide captures the *design* details of the program – the blueprint by which the final program was written.

It is in this part of the writeup that you should describe your data structures, the algorithms you have chosen, the module structure, the way errors are handled, etc. This is not an appendix to your program. The user has not looked at your program yet, but rather is trying to find out whether or not to look at it, and on what parts to concentrate. You should not force the user to turn to the program to make sense of the writeup, but there certainly will be details in the code that are not in the programmer's guide.

Note that, in addition to the user's guide, the programmer's guide should be a working document for your group. In a strict sequential development model, the programmer's guide would be written *before you have begun the implementation*! While this strict ordering is too much to ask for in general, most development efforts begin too early, when not enough of the design has been thought through. So: resist the temptation to begin hacking code immediately. Rather, your efforts in the first week should be in understanding the requirements and carrying out the design, to increasing levels of detail. As your design becomes more detailed, carrying out the implementation should become simpler.

The following sections describe suggested contents for the programmer's guide. These sections should not be interpreted, however, as a blueprint for your Programmer's Guide. In other words, you do not need to have one section in the Programmer's Guide corresponding to each section listed below. Organize the document in a manner that best suits its contents, which will depend somewhat on some of your early design decisions such as implementation language and platform.

Front Matter

Like any technical document, the front matter should help structure the entire document.

The title page should include at least the Title, the group name, the primary author (recall that everyone must take a turn with at least 1 programmer's guide or 1 user's guide), and the date.

The table of contents and lists of figures or tables should help to guide the reader to a specific chapter, section, subsection, and page of interest. Choose a hierarchical depth that is appropriate to the content of your document.

The introduction should provide an overview of the document contents, supplying background information and making explicit any assumptions that will be made regarding the reader's background. The choice of these assumptions is important. Do not assume familiarity, for example, with the code! But you may assume some level of technical sophistication.

Overview of System and Data Flow

The first sections should give a higher-level decomposition of the system design and functionality. Pictures and prose should be used to describe how the larger conglomerations of components relate to each other and how information flows between them.

Implementations will most likely span multiple files. Your programmers guide should explain the directory structure. Note that both mappings are important: (i) What does file X contain? and (ii) In which file can I find element X?

It is also important to explicitly identify and document all of your adopted design conventions. For example: What are the naming conventions for variables, constants, functions, classes, files? What are the parameter-passing conventions? What are the memory-management conventions? What are the error-catching and error-message generation conventions?

Finally, module inter-relationships should be described. In an object-oriented design, the inheritance and encapsulation relationships between classes should be diagrammed. A diagram of run-time instantiations of the static class hierarchy is also useful. In procedural designs, the calling relationships between functions should be diagrammed. Pictures are very helpful here, but are not enough.

Data Structure Descriptions

All major shared data structures should be carefully documented. In documenting data structures, it is important to describe the role the structure plays in the execution of the program (e.g., an object called "pc" may represent the program counter of the virtual machine), as well as its implementation (e.g. pc may be a record having two fields, one called "length" and the other called "value"), and any invariants (e.g. pc has a value in the range 0 to 65,535).

That is, both the concrete (component-side) and abstract (client-side) aspects are important since the maintainer of your software may need to work with either (or both).

Module Descriptions

In documenting modules, it is important to show which modules invoke which others, as well as how individual modules work and, lest we forget, what they do. Modules that encapsulate a data abstraction should separate the specification (i.e., definition) and algorithmic (i.e. implementation) details. Parameter lists are an essential part, but not the whole, of module documentation. Module interface descriptions must include what a module assumes about its calling environment ("requires") and what it, in turn, guarantees to perform ("ensures").

The description of each module should follow a parallel structure. I recommend a structure based on the following template:

```
Procedure Name:
Description:

Calling Sequence
    Input parameters:
    Output parameters:
Requires:
Modifies:
Ensures:

Error Conditions Tested:
Error Messages Generated:
Original Author:
Procedure Creation Date:
Modification Log:
Who    Date    Why
```

You can include this template in the implementation itself, which will facilitate keeping it up-to-date and accurate (ie, as someone modifies the module, the can append a line to the modification log.)

Finally, algorithmic details can be well-expressed with pseudocode.

Data Element Dictionary

This section is used to describe each shared variable used in the program. You do not need to include variables and constants that are declared and used locally within a single function or module. (These variables and constants should be documented with that module.) The following format can be used.

Variable Name	Type	Declared In	Used By	Purpose

Code

An essential part of the documentation of any program is the source code itself. Despite the foregoing, *all that you have ever learned about comments, choice of variable names, blocking structure, etc. still applies.* Do not forget that someone modifying your program needs to be able to read it. *The code should be organized so that individual modules and data structures can be found easily and read quickly.* It is a good idea to adopt a precise coding standard, such as can be found under the class web site for C++ and for Java.