Effective Documentation: a Case Study

Joseph E. Taylor
Inter-Tel, Incorporated
7300 West Boston Street
Chandler, Arizona 85226-3224
Joseph_Taylor@Inter-Tel.com

David C. Pheanis
Computer Science and Engineering
Arizona State University
Tempe, Arizona 85287-5406
David.Pheanis@ASU.edu

Abstract

University students who study computer science in classroom settings have a difficult time grasping the importance of thorough documentation. This lack of understanding inevitably spills over into industry as students graduate and pursue careers. The resulting lack of documentation is unfortunate because thorough documentation greatly eases the burden of software upgrades and maintenance. This paper examines guidelines for thorough documentation and illustrates the main points with a case study. The case study is about a graduate student who has ported a large, complex program from an assembly language to C++ without even knowing the language of the original program. Such a task was possible only because of the thorough documentation of the original program.

1 Introduction

Computer programs in industry are typically large. Programmers maintain software over long periods of time — fixing bugs, adding new features, and re-releasing the software. Thorough documentation simplifies the maintenance of large programs. Proper documentation reminds programmers of their design decisions in previous months, or even years, and greatly eases the training of new programmers.

University instructors of computer science face a daunting challenge when trying to teach their students the importance of thorough documentation. This challenge is particularly difficult because the programs that students write are, of necessity, small. Additionally, the purposes of programming assignments in the classroom are to illustrate particular concepts that are new to the students. This purpose usually results in programs that are not especially practical for applications in industry. Furthermore, because semesters are short, instructors cannot require students to maintain and extend their original programs over a period of many years or even many months. Such long-term development and maintenance assignments would reveal whether the students produced thorough and proper documentation.

2 Case Study: An Unknown Language

ASM is an assembler that we use at Arizona State University (ASU). ASM accepts assembly-language programs for many Motorola processors in the M6800 family. We originally wrote ASM in the Macro-32 programming language (i.e., VAX assembly language). ASM is well written, modular, and well documented.

In an effort to move from the VAX platform to other platforms, faculty at ASU wanted to port ASM from Macro-32 to a high-level language.

As part of the research for his Master’s thesis, a student at ASU has ported ASM from Macro-32 to C++. He ported this large, complex program without even knowing the Macro-32 programming language. He read only the documentation of the code and the data structures to understand the functionality of each module, and he translated the functionality into C++ code. This translation method is a clear testament to the quality and completeness of the documentation in the original ASM source code.

3 Thorough Documentation

What makes documentation so complete and descriptive that a programmer can port a program without even knowing the original language of the program? Consider the following.

3.1 Six Questions

For complete coverage of an event, journalists answer the following questions when reporting:
1. Who?
2. What?
3. Why?
4. Where?
5. When?
6. How?

Thorough documentation answers these same questions to the extent that they are meaningful and relevant. Like a journalist, the programmer who answers these questions when documenting code gives readers a complete picture of the program. Future readers, including the original programmer, will more easily understand the design and flow of the program. More importantly, readers will understand the intent of the design and the reasoning behind the design decisions. Readers will not have to read the code to determine what the program does. Certainly, well-written English is easier to understand than well-written code.

3.2 Documentation Areas

There are four main documentation areas that programmers should consider when documenting their programs. They are:

1. the program overview
2. variables and data structures
3. subroutines, and
4. logical groupings of instructions.

We will first explain these areas and then use examples from ASM to illustrate these important documentation concepts.

4 Program Overview

A good program overview starts by describing the purpose and function of the program. The description includes program inputs, program outputs, and any run-time options that users may choose. It also includes a high-level description of program flow. For maintenance of larger programs, a history of major modifications is also helpful. Histories should include reasons why the programmers made the modifications.

ASM’s program overview contains such items as a copyright notice, an introduction, a history of changes and improvements with credit given to the implementers, installation instructions (for VAX platforms), implementation notes, and a list of desirable future improvements. The graduate student who ported ASM to C++ found the historical notes to be unexpectedly valuable. The history revealed how ASM has evolved over the past 15 years, and the considerations and reasoning behind some design decisions became apparent. This insight inevitably influenced the student to preserve those designs as the better choices among other options.

5 Variables and Data Structures

The documentation of variables in most programs is grossly inadequate. In a typical program, programmers must find and examine all references to a variable throughout the code to gain a clear and accurate understanding of the variable’s use, purpose, and function. Maintenance programmers would have a far easier time if the documentation with the declaration of the variable explained all the information that a programmer would want to know about the variable. Thorough documentation answers the six questions posed in section Section 3.1 above, namely,

1. Who (i.e., what routine) uses this variable?
2. What is this variable’s purpose and function, and what meanings do various values of the variable have?
3. Why is this variable necessary or useful?
4. Where in the program do the initialization and modifications occur?
5. When does the program initialize and modify or use the variable?
6. How does this variable contribute to the overall program, and how does the program use this variable?

Such comments can be lengthy, but one location (i.e., a variable’s declaration) contains all the information that a programmer would want to know about the variable. The programmer can conveniently read this documentation in one place and understand everything about the variable and the design of the program relative to the variable. Alternatively, a programmer
would need to search the source of the entire program to find all references to the variable, examine each of those routines carefully, and deduce the information that should have been with the variable’s declaration.

The documentation of data structures is probably the most important part of a program’s documentation because the data structures are the heart of a large program’s design. Diagrams or pictures (using ASCII characters) help programmers visualize the data structures clearly. Diagrams also help explain concepts that may be confusing or deviate from the “expected” approach. Diagrams can replace pages of textual explanation by supplementing clear, concise discussions. Diagrams along with text are usually more clear than abstract text alone.

A key data structure in ASM is the symbol table. The symbol table contains the labels and values that a user’s program defines. The symbol table is very complex because it must contain so much information, such as permanent labels, redefinable labels, absolute or relocatable properties of such labels, values of labels, and a reference list for each label.

At the declaration of the symbol table and symbol-table nodes, ASM contains seven pages of documentation that includes text, diagrams, and tables of values. There is a diagram of a symbol-table node showing field sizes relative to each other, positive and negative offsets from the base of a node, and fields that are common to other data structures in ASM. A symbol-table node contains some bit flags, and there is a table of values for all bit flags with the meaning of each value. There are also some fields that are optional and make the symbol-table nodes variable in length. The documentation identifies the conditions under which the optional fields exist.

The graduate student who ported ASM to C++ found this documentation very helpful. He had to read these few pages of documentation to gain a complete understanding of the use of the symbol table in ASM. The alternative — to decipher literally hundreds of pages of VAX assembly code to learn how ASM uses the symbol table — was far less attractive.

The assembler’s symbol table is a height-balanced, right-threaded, binary tree with variable-length nodes. There is no need for a diagram of the symbol table because a height-balanced, right-threaded, binary tree is a well-known data structure that programmers understand. However, the handling of redefinable labels in the symbol table requires modification to the binary tree structure. ASM’s documentation uses a simile to illustrate vividly how ASM inserts redefinable labels into the symbol table. To quote from ASM’s documentation:

The first node for a given redefinable label is in the binary symbol-table tree, and the rest of the nodes hang from the tree (like Spanish moss) in a linked list.

To proceed even further, this linked list that hangs from the base node in the binary tree is a mutation of a typical linked list in computer science. Indeed, the link fields of the nodes have different meanings during different passes of assembly. Therefore, ASM’s documentation contains a diagram of this substructure (i.e., this customized linked list). Plenty of explanation accompanies this diagram to give the reader a clear understanding.

After reading the documentation and diagrams of all the data structures, the graduate student who ported ASM knew how most of the assembler worked even before approaching the code. Indeed, he could have invented most of the code if given just the documentation of the variables and data structures.

6 Subroutines

For each subroutine, a programmer should include a clear and concise description of all functions that the subroutine performs. The programmer should also describe how each subroutine relates to the program as a whole and when and how to use the subroutine. A programmer should also include entry and exit conditions, including any global variables that the function uses or modifies. For assembly-language programs, a register map is particularly useful. A register map simply lists the registers that the subroutine uses and the purpose, meaning, and value for each use. Registers often contain variables, and the names of the registers are not mnemonic toward the variables they hold. Readers can refer to the register map when they forget what registers hold what values.

ASM follows these guidelines for its subroutines. Each subroutine contains a brief description of its purpose, including entry and exit conditions. Because ASM is written in assembly language, the subroutines also have register maps. For example, ASM contains a subroutine that defines a label in the symbol table. After describing the function of this subroutine, the comments give a sample calling sequence. The entry
conditions identify six global variables that the subroutine uses as inputs, and the entry conditions also explain what values those variables must have. The exit conditions identify registers that the subroutine preserves and registers that it modifies. The exit conditions also explain how the subroutine reports any detected errors and how it puts the given label into the symbol table, if appropriate. The register map identifies three registers that the subroutine uses and explains how the subroutine uses each one.

Such documentation throughout ASM ties all of the subroutines together into one coherent program. Thus, the graduate student who ported the program easily ported subroutines to C++ and related each of them to the overall program.

7 Logical Groups of Instructions

At the lowest level, programmers should format related instructions into logical groups with white space between one logical group and the next. Some groups logically contain only one instruction, but most contain several. A comment for each group should address the six questions from section Section 3.1, although the Who, Where, and When information is usually implicit in the group’s position in the program or subroutine.

Too often, programmers throw in some meaningless words or comments with little or no value next to their code because they know it is “good programming style” to include a comment. Such comments are usually just statement-level descriptions that do not demonstrate much thought. For example, a comment such as Assign zero to variable rel_mode for the C++ statement int rel_mode = 0; is all too common. Such a comment reveals absolutely no useful information to the reader.

The one question that programmers most frequently neglect to answer in their comments is, ironically, the most important question of all, Why? All too often a comment explains What a step does and perhaps even explains How the step accomplishes its task, but typical documentation usually fails to explain Why the step does what it does.

Thorough documentation answers the six questions of section Section 3.1. For example, ASM contains a global flag called RELFLG. Its declaration forms a logical group containing just one statement. The following comment accompanies this variable’s declaration.

Note how it answers the six questions of section Section 3.1.

Relocatable mode flag. We clear this variable to indicate an absolute assembly, or we set it to $FF to indicate a relocatable assembly. Subroutine RELOPT sets RELFLG to $FF for a relocatable assembly when the user uses an OPT statement to request a relocatable assembly. The absolute mode (which is the default) gives the user the ability to generate an S-format object module. The relocatable mode gives the user the ability to generate a relocatable object module, which is suitable for later input to the UniLink linkage editor. We initially set this variable at the beginning of each pass from the front end’s relocatable-mode flag, so the assembler properly starts each pass in the same mode.

True, such a comment is lengthy for just one variable declaration, but is there any doubt as to how ASM uses this variable? A person who doesn’t presently care about this variable can quickly bypass the variable’s entire comment. On the other hand, a person who needs to know about this variable has the necessary information readily available in a single location.

All of the source code for ASM places related instructions into logical groups. Only because the code follows this guideline was the graduate student who ported ASM able to port it from Macro-32 to C++. The student did not understand most of the code at the instruction level because he did not (and still doesn’t) know the Macro-32 programming language. The comments next to the logical groupings of code, however, told him what to program next in C++ and why to do it.

8 Conclusion

Thorough documentation is a concept that is difficult to teach in the classroom because its importance is difficult to understand until one actually works to maintain a large program over long a period of time. Good documentation greatly simplifies the maintenance of large programs. The ASU graduate student who ported an assembler presents a compelling example of what thorough documentation can accomplish. If he can port from a language that he doesn’t even know, imagine how easily a programmer who knows the language can maintain that same code.