software development
The lifecycle starts here.

Writing Understandable Code

January 2002

Sure, you know that you should comment your code, but how well do you actually do it? Here's one developer's perspective on documenting your application's raison d'être through comments, naming conventions and clean constructs.

By David Michael Zokaites

A few years ago, while working on some source code that someone else had written, I experienced the frustration of almost—but not quite—knowing which algorithm a function utilized. I could discern that it provided a polynomial fit to input data, but I couldn't determine precisely which curve-fitting algorithm was implemented. A suite of system tests found a calculation error in the function. But without knowing precisely which algorithm was implemented, the most expedient solution was to rewrite the offending function with a Newton-Gregory forward polynomial. If the developer had only provided a few more comments or perhaps an algorithm reference, I could have made a small correction instead of going the hard and expensive way—by rewriting the function from scratch.

Writing comprehensible source code gives software engineers an integrated guide to the many details of the systems they build, and best of all, the guide lives in the programs themselves, through comments, naming conventions and system descriptions. Understandability is important at many levels, from microscopic analysis of individual instructions, to awareness of the purpose of a particular function, to comprehension of the overall system design. Through the simple act of writing intelligible source code, you can increase the efficiency of software development, enhance maintainability and improve overall productivity.

Software Evolution

Software is dynamic: The mark of a useful system is that it changes and grows, and is refined and corrected, throughout its life cycle. Indeed, the moment a software system ceases to change is the moment it's unplugged and begins to fossilize with the other discarded dinosaurs. The best way to extend the life of software systems is to provide documentation: written descriptions of program operation and design on scales from micro to macro. Without documentation, complex software systems collapse into morasses of unintelligible code.

In the business of system development and maintenance, it's frequently necessary to thoroughly understand all the details of long-forgotten code. I've often had the challenge of trying to decipher another person's application—and I've also had the humbling experience of trying to understand code that I wrote years ago. On these occasions, I realized anew the importance of improving the understandability of my programming, and hastened to increase the detail and extent of in-code and supporting documentation. A decade's worth of such experiences has gradually resulted in my code being exceptionally well-documented. I now believe that if it takes more than a moment to understand a code segment, either more documentation is in order or the program structure should be revised. If I have to get back into this code five years from now, I'll want to know exactly what everything does, and why—and I'll want to know it fast!

Programmers can effectively store their reasoning in the very lines of code they write. That knowledge store should be as lucid as possible for maximum efficiency. Enabling some unnamed future programmer to quickly grasp this knowledge is simply the most sensible choice. Not only does understandable code help future bug-fixers and functional enhancers, it also assists current developers in clarifying and deepening their own understanding of the work in progress.

Natural Language

Software must be understandable to two different types of entities for two different purposes. First, compilers or interpreters must be able to translate source code into machine instructions. Second, people need to understand the software so they can further develop, maintain and utilize the application. The average developer overemphasizes capability and function while undervaluing the human understanding that effects improved development and continued utilization. There should be a description—in clear view—within the programming medium.

As I gradually improved my in-code documentation, I realized that English is a natural language, but computer languages, regardless of how well we use them, are still "code." Communication via natural language is a relatively quick and efficient process. Not so with computer languages: They must be "decoded" for efficient human understanding.

The Aesthetics of Software

People who read my code … wait a moment—did I say "*read* my code?" Now that's a remarkable way to approach software—not to debug, analyze, program, or develop, but simply to *read.* The act of reading allows me to approach my code as a work of software art: I strive to make the overall design, algorithm, structure, documentation and style as simple, elegant, thorough and effective as practical. Yes, this takes time, but when I'm rushed, I usually dash off the wrong implementation of the wrong design, and the darn project takes twice as long as it would have had I done it right in the first place. A disciplined, focused approach clarifies my thinking and improves my implementation. In keeping with a reasonable attempt for excellence, I proofread my applications.

My goal is to find a balance, describing all salient program features comprehensively but concisely. I explain each software component's purpose, the algorithm used, arguments, inputs, outputs—even the reason for `#include`-ing a particular header file. I document each section of each function so that the overall program flow is readily understandable. Locating the particular section under study thereby becomes an easy task.

In the world of software maintenance and development, it's often necessary to locate particular sections of code to augment or correct. A system component that's optimal (or good enough) today may well require modification in a few weeks, months or years. If all salient features are clearly indicated, the software change process will proceed much more efficiently.

The Name Game

All variable, function and source file names should be succinct and descriptive, and accurately reflect the purpose of the variable. I believe that even the lowly loop index deserves a practical, descriptive variable name. While loop indices "i," "j" and "k" are short and easy to type, they offer no clue about what is looped over. Better loop indices (say, for image processing applications) would be "imageLine," "imageSample" and "imageBand". And attempting to locate a loop index with a text editor usually finds many undesired matches for the letter *i.* To avoid an excessive number of matches with text editors, I tend to shy away from using simple nouns (that is, "line") for variable names and instead use a compound word (that is, "imageLine" or "iLine").

Descriptive naming can be applied to more than variables and file names. Symbolic constants, also known as *macros* or *enumerated types* (implemented via `#define` and `enum` in C), are easier to work with than "magic numbers" scattered throughout source code. Using these constructs allows programmers to consolidate the definition of important numbers in one location and to describe their meaning with comments. Using the constant MAXCOLUMN in place of the number 2000 gives you source code that's understandable and easy to maintain.

For large applications and systems, it can be difficult to create short names that are sufficiently descriptive. One solution is to limit the scope and visibility of the various name spaces. Encapsulation reduces the area of applicability for various program objects; this allows you to create a relatively short, yet distinctive name for the given object.

Constructs, Clear and Simple

The size and complexity of software components have a huge impact on source code understandability. We work much more efficiently on simple, modular components than we do on rambling, convoluted, cryptic code constructions.

The most basic practice, for those programming in procedural languages, is to avoid the infamous `goto` statement—well structured and designed programs simply have little need for it. "Spaghetti code," riddled with `gotos`, is notoriously difficult to decipher.

Similarly, understandable code employs very little typecasting. It's much better to declare variables of the required type than to transform one type into another. Passing a long list of variables into a function is also a cause for confusion. Functions with long argument lists are (most likely) poorly designed. Possible underlying causes for such lists include the under-utilization of classes (or data structures) to consolidate information about an object and functions that attempt to solve divergent needs.

Change Logs

As a piece of code winds its way through various versions and releases, it may experience a myriad of changes. Software developers and bug fixers, particularly those who work with concurrent development paths, multiple

supported platforms and release levels, must be able to reconstruct software evolution and capability. To this end, a well-maintained change log can provide key insights into the operation of a function at any point in its life cycle.

Once you've decided to maintain change logs, the next question is where to keep them. The two most common locations are in the source code at the top of each file, and in databases (or files) maintained by the configuration management tool. I prefer the latter, because embedding change logs in source code clutters it with infrequently accessed details.

Algorithms and Intricacy
Modern applications tend to have elaborate structures and deeply nested logic. Scientific applications have advanced mathematical/statistical techniques; business applications have lengthy lists of business rules; word processing applications have complex interactions of multiple formatting specifications. Whatever the domain, modern software is complex. Some form of algorithm specification is a definite prerequisite for the timely understanding of software by programmer and user alike.

For small applications, it's often practical to provide a complete derivation of the algorithm in a comment block at the end of the source file. Alternately, if a derivation appears in the published literature, it would be sufficient to cite the journal article or the page numbers of a book. For larger systems, logic flows can be described in an Algorithm Theoretical Basis Document.

Reviews and Tests
Literary works almost always benefit from review. Sometimes, all that's missing is an occasional comma; at other times, an insightful editor revises entire chapters or transforms lame endings into blazing finales. As with literature, source code benefits from peer review. You may only need to clarify a few comments and make variable names more descriptive, but when a second pair of eyes finds fundamental implementation flaws at an early—and inexpensive—stage of development, the extra process is worth it.

In the final analysis, however, there's only one way to fully understand a software component: by testing it. Verifying logic, output and operability with a robust suite of tests is a critical phase of the development process. Applications that have not been fully tested will probably not work as intended—if they work at all. After developing a unit of code, I verify that unit with a suite of test scripts that are configuration-managed alongside the source code. Where and when appropriate, I also develop debugging/verification blocks of code (or functions) that remain in the source files (usually bracketed by `#ifdef DEBUGGING`).

Ubiquitous Understandability
Comprehensible source code has a tremendous influence on software development, maintenance and enhancement. Essentially every aspect of a program—including algorithms, inputs, outputs, procedures, derivations, techniques and system interfaces—is worthy of documentation. Indecipherable systems that can't evolve tend toward obsolescence, whereas in-code natural-language commentary, coupled with good design and thorough system documentation, makes code much easier to work with.

Don't Be Shy
A profusion of comments provides an easy-to-follow natural-language narrative.

My programming and placement style gradually evolved to emphasize this narrative description of processing flow: I generally provide a header-type comment for every few lines of code, and my programming statements are (usually) indented and placed directly under the comment. Although I generally indent source code from comments, at times, comments are better placed to the side of the code to emphasize program structure. The following source file illustrates this coding and commenting style:

```
          /* this file contains the source code for die() */

/* include files */
#include <stdarg.h>   /* for variable argument processing */
#include <stdio.h>   /* for terminal output and remove() */
#include <stdlib.h>   /* for calling exit() */
#include "underStand.h" /* application header file */

 /* function: remove temporary files, write message to stderr, and exit */
void die (const struct FILELIST *toRemove,/* input: list of files to remove */
```

```
    const char *format,       /* input: format of message to print */
    ...)              /* input: arguments for print format */
{
  /* declare variables */
   va_list argPointer; /* argument pointer for variable arg processing */
   int fileNum;    /* for loop index, file number */

  /* get variable number of arguments for vfprintf() */
   va_start (argPointer, format);
   va_end  (argPointer);

  /* remove temporary files */
   for (fileNum=0; fileNum < toRemove->count; fileNum++)
     if (remove (toRemove->name[fileNum] ))
      fprintf (stderr, "Program underStand encountered an error while "
      "attempting to remove temporary file %s\n", toRemove->name[fileNum] );

  /* print an error message */
   fprintf (stderr, "Fatal error encountered by program underStand:\n");
   vfprintf (stderr, format, argPointer);
   fprintf (stderr, "\n");

  /* exit */
   exit (1);
}
```