Session Number 3150

Adaptive Documentation: An Optimized Approach to the Documentation Dilemma

Gregory M. Dick, Frank W. Pietryga

University of Pittsburgh at Johnstown

Abstract

Instruction in computer programming includes both an introduction to program documentation and opportunities to practice program design and coding. A trade-off exists between these two programming elements (coding and documentation.) Assuming that student time is finite, time devoted to one must come at the expense of the other. A change in programming language from FORTRAN to C required that students devote additional time to coding because of the subtle language complexities of the C language. An approach that permits additional focus on coding while neither abandoning appropriate emphasis on documentation nor overloading the students, is presented. A related effort to improve programming instruction in MATLAB by carefully adapting documentation requirements is discussed.

Background

Instruction in computer programming has been a required component of the Engineering Technology curriculum at the University of Pittsburgh at Johnstown (UPJ) since its inception in the early 1970s. The primary goals of the course are to give the students a firm grounding in the basics of:

- problem solving
- algorithm development
- program design, coding, testing and documentation.

This paper focuses on the third of these.

The programming course at UPJ has always included programming projects. Nominally, one programming project is assigned each week. No assignment is made in those weeks during which an exam is being administered. In the second half of the course more substantial projects are assigned; these may be intended for one and a half or two weeks duration. The result is that students typically complete about ten programming projects.

Students submit a report documenting their experience with each project. The report includes:

- Discussion of Results – a brief discussion that addressed their perception of the success or failure of their program and other observations they may have made as they worked on the project.
- Answers to Questions – most assignments include a small set of questions pertinent to the project. The students submit their answers.
- Documentation Package – this includes:
    - Source Code – The code must be a stand-alone document. Thus, much of the documentation is embedded within the source file. It includes:
        - Title, programmer, date.
        - Program description. The program description includes an itemization of the program's inputs and outputs and a brief description of the function performed by the program. The description for substantial programs can be rather lengthy, but it is important that it be bundled within the code so that it cannot become separated from it.
        - Synopsis. The synopsis is an abbreviated version of the pseudocode for the program. It summarizes the algorithm used to implement the function of this program.
        - Executable code. The code should be clearly commented. Typically, each paragraph of code corresponds to one entry in the synopsis and is preceded by a comment that repeats that entry. Additional comments are included as necessary. Mnemonic variable names and Hungarian Notation are also required.
    - Pseudocode or Flowchart – Both approaches to algorithm development and documentation are taught.

Appendix A contains two typical source code listings. The first is a small program which consists of nine C statements and a documenting header that is eleven lines in length. For this example, the documentation is more than 50% of the file. The second example is typical of a programmed solution to an engineering design project. It is included to illustrate the scope of these project assignments. Note that documentation elements of the source file, i.e., header data dictionary, are significant components of the program. The addition of the other required report elements, i.e., discussion, answers to questions, and flowchart or pseudocode, would add two or three additional pages. Thus, report lengths of four or five pages are not unusual.

This course has evolved during the past thirty years; this is expected. The programming environment was originally punched cards submitted to a batch operating system. In the 1980s a time-sharing operating system (VMS or UNIX) was employed and source code was developed using a simple text editor. The current environment is supported on desktop PCs running Windows XP. Microsoft Visual Studio, an integrated development environment, is the platform. Additional details (syllabus, assignments, tests and quizzes, example programs and other handouts) about the current state of this course are available at the course web site[1]. Related information about the EET program is available in the College Catalog[2].

[1] http://faculty.upj.pitt.edu/gmDick/courses/sea/
[2] http://www.upj.pitt.edu/coursecatalog/default.cfm

More important to this discussion is the evolution of programming language. When this course was instituted in the 1970s, the language of choice for solving scientific and engineering problems was FORTRAN. FORTRAN was the basis of this course for many years.

The current language of instruction is MATLAB for mechanical and civil engineering technology students and C for electrical engineering technology students. The course focus for civil and mechanical students remains the solution of analytic problems. MATLAB was chosen because of its friendly programming environment, inherently powerful language elements and wealth of useful function and toolboxes. A primary motivation for choosing C as the language of instruction for electrical students was the need to prepare them for a subsequent course in embedded systems based on the Motorola MC68HC12 processor and the IAR Embedded Workbench integrated development environment. Similarities between the development environments provided in Visual Studio and Embedded Workbench assist in making the transition from PC based programming to the programming of embedded systems relatively trouble free. C is clearly the correct language for our students.

The transition from FORTRAN to C, like nearly any change, produces both intended and unintended consequences. The intended ones are good; the students are prepared to commence the study of embedded systems with C language skills in place. The unintended consequences emerge from the language complexities of C vis-à-vis FORTRAN. Students experience more difficulty in mastering these elements. Typical problem areas have to do with those language elements that generate subtle errors. Examples include, among others:

- Confusion of logical conjunctions (i.e., && and | |) with bitwise Boolean operators (i.e., & and |)
- Confusion of the replacement operator "=" with the relational operator "=="
- Misplacement of the semicolon punctuation mark.

Specific examples include:

```
if (a = b)
{
  conditional statements
}
```

This code phrase is syntactically correct and will compile and execute without error or warning. However, it will not test to determine if "a is equal to b" as the neophyte programmer intends. Another trap is:

```
if (logical expression)  ;  /* misplaced semicolon */
{
  apparent conditional statements
}
```

In this example the apparent conditional statements always execute. The if statement controls the conditional execution of a null statement which is located in the white space preceding the misplaced semicolon. This is not what was intended!

The important feature of these insidious traps, that's how students describe them, is that they are not the result of conceptually difficult ideas. Rather, they are simply examples of the subtlety of the language syntax. Mastering them does not require great intellectual effort; rather, what is required is simply repeated exposure – practice. If the goal is mastering these complex yet subtle language elements, then the obvious solution is simply more practice. However, it is not that simple.

The Problem

The language transition from FORTRAN to C has led to a problem. It is perhaps best illustrated by considering the following:

- Traditional programming assignments require both working code and appropriate documentation.
- Both components of the assignment require student time to complete.
- Student time is finite
- We might represent the above as:
  Code + Documentation = FiniteStudentTime
- The language transition from FORTRAN to C requires additional student practice designing and writing code.
- It's clear that *increasing* Code leads to a *decrease* in Documentation. This trade-off between Code and Documentation is the root of the problem.

Thus, the challenge is to increase students' efforts writing code while neither abandoning appropriate emphasis on documentation nor requiring exorbitant student time.

The Solution

A solution, which allows for increased programming practice while neither compromising appropriate emphasis on documentation nor requiring exorbitant student effort, is sought. The solution arrived at attempts to manage the trade-off between coding practice and documentation in a manner that optimizes learning. It is based on increasing the total number of assignments, using an approach that does not increase the total time commitment. Two different types of assignments are now used; some requiring complete documentation, others requiring minimal documentation. Specifically, the course now includes:

- Standard Projects as described above. These include a small report and completely documented source.
- Little Projects. These require no report and the source code need not include the program description and synopsis. This significantly reduces the amount of time required to complete these projects.

Little projects tend to be highly focused and emphasize a single concept, e.g., loops, if statements, case structure, file I/O, etc.) Standard projects are more involved and often focus on real world problems. A typical project is one that focuses on a geometric design problem which leads to the need to solve a transcendental equation. Thus, the actual programming assignment might be root finding using Newton Raphson iteration (see

Appendix A.2.) The project report would discuss both the design problem and the C program on which the solution is based.

The total number of programming assignments has been increased. The assignment mix is approximately two little projects for each standard project. The conditions described above continue to apply, i.e., no assignment during an exam week and more substantial projects in the second half of the course. This leads to six standard projects and twelve little projects for a total of eighteen total programming assignments. This is a significant (80%) increase and affords the students ample additional opportunity for practicing the skill of program design, coding and debugging. However, the continued inclusion of a number of projects which require complete documentation maintains an appropriate emphasis on documentation. The table below summarizes the change and demonstrates that the total student effort (measure by total number of report pages) is not significantly increased.

Table 1. Comparison of Previous and New Approach

	Previous Approach		**New Approach**	
	# Projects	# Pages	# Projects	# Pages
Standard Projects @ 5 Pages	10	50	6	30
Little Projects @ 2 Pages	0	0	12	24
Totals	10	50	18	54

This approach recognizes the fact that learning how to properly document a program requires less practice than code generation and debugging. Students must be exposed to an appropriate standard and be required to implement it several times in order to understand what is involved. Participation in group projects where each student implements only one module of a multi-module project further emphasizes the importance of proper documentation. This approach is seen as superior to the alternative of requiring the same level of documentation on all projects but one that is significantly relaxed from the previous approach. This sends the wrong message i.e., that incomplete diluted documentation is acceptable.

Assessment

This revised approach to programming assignments was implemented in the fall of 2003. Assessment data is sparse, but encouraging. Student performance on exams improved. This was most noticeable when examining the practicum portion of the exam. During this part of the exam students are given several small problems and are require to design, code and debug the program during the exam period. Exam scores increased, on average, more than 15% when the new approach was adopted. These data should be read with some caution. The intent was to draft exams of equal difficulty, but it is difficult to write exams which present precisely the same challenge. The apparent improvement indicated by exam scores was corroborated by instructor observations. The types of errors that this approach was focused at reducing were, in fact, reduced. This was evident during both classroom discussions and office consultations.

MATLAB - The Other Side of the Coin

The above discussion focuses on increasing learning by selectively decreasing the emphasis on documentation in the C language programming course. It is interesting to note that precisely the opposite has taken place in the MATLAB course.

It is our observation that one of the more troubling aspects of learning the MATLAB environment is mastering the concept of scalar, vector and matrix data types. This is most noticeable in the early stages of the course when students are attempting to grasp many new concepts. A lack of discrimination between scalars and vectors is seen to frequently lead to student difficulty. In this case, a selective increase in documentation has been found to effectively address this problem.

MATLAB, unlike C, is not strongly typed. Variable declarations are not required; the mere mention of a variable name causes its creation. Students are not required, by the language syntax, to specify *a priori* the type of a variable. This feature makes rapid development of problem solutions easy for the experienced programmer. However, the neophyte often is not fully aware of the type (scalar or vector) of the variable that has been created; this can lead to difficulties. The solution to this problem is to introduce an additional documentation element - the data dictionary.

A data dictionary is a table that lists a variable's name, type, usage, and description (see Appendix C – Example MATLAB Script.) The construction of a data dictionary during the design phase requires that the student focus on the variable type, vector or scalar, before proceeding to design and code the script. This tends to reduce the number of errors. However, the construction of a data dictionary also increases the amount of time required to craft a program and, for the more experienced programmer is not always necessary. The approach adopted:

- Requires student to include data dictionaries in their code during the early phases of the course, i.e., when they are most likely to make errors rooted in misunderstanding variable types.
- Relaxes that requirement later in the course after the students have mastered the concept of different variable types and are not experiencing difficulty.

Assessment

Assessment is both anecdotal and subjective. Instructor observations indicate that beginning student programmers make fewer errors when required to draft a data dictionary. Conversations with students indicate that they "get it" better when they use a data dictionary. During office debugging visits, their response to the question, "is this variable a vector or a scalar?" is almost never a blank stare (which was not unusual before this approach was adopted.) Students indicate that this approach helps them to grasp the concept. They also appreciate the relaxation of the requirement later in the course when they see it as unnecessary busy work.

Conclusions

Instruction in computer programming addresses both the design of good code and the documentation of that code. Program documentation is important both in its own right and as an important facet of pedagogy. In two instances involving two different languages (C and MATLAB) we have sought methods to improve overall instruction by adapting the documentation requirements to address specific pedagogical needs. Preliminary assessment indicates that these techniques are successful. Exam scores have improved and instructors' perceive that the students' mastery of the languages has also improved. Colleagues who have experienced similar problems, that is:

- Students struggling with C language syntax complexities
- Students experiencing difficulty because stemming from lack of an appreciation of the differences between scalar, vector and matrix variables in MATLAB

may wish to adapt these approaches. Both appear to improve student learning with only modest costs in student effort.

Biography

GREGORY M. DICK – Associate Professor and Head of Electrical Engineering Technology. Dr. Dick holds degrees from the University of Pittsburgh, Stanford, and the Pennsylvania State University and is licensed in the Commonwealth of Pennsylvania. He has taught at Pitt-Johnstown for 30 years. His areas of interest include Computing, Systems and Controls, Digital Signal Processing and the interface between technology and society.

FRANK W. PIETRYGA – is an Assistant Professor at the University of Pittsburgh at Johnstown. He graduated form UPJ in 1983 with a BSEET degree and completed his MSEE degree in 1993 at the University of Pittsburgh, main campus. His interests include power systems engineering, AC/DC machinery, power electronics, and motor drive systems. Mr. Pietryga is also a registered professional engineer in the Commonwealth of Pennsylvania.

Appendix A – C Language Examples

A.1 – square.c – Small example program

- Header = 11 lines
- Code = 9 statements

```
/* square.c
 * (c) 1996 g.m.dick Revised- 9/2003 c.j.s
 *
 * Program Description:
 *  Inputs:   a data value (iI)
 *  Outputs:  the square of the value (iJ)
 *  Function: this program reads a data value,
 *            computes, & prints the square.
 *
 * Synopsis:
 *  - Prompt and read input
```

```
 *  - Compute its square
 *  - Echo data, print square
 *
 */

/*----------------------------------------preprocessor directives */
#include <stdio.h>

/*-------------------------------------------------------------main */
int main (void)
{
    /* Declarations */
    int iI;
    int iJ;

    /* Prompt and read input */
    printf ("Enter an integer: ");
    scanf  ("%d",  &iI);

    /* Compute its square */
    iJ = iI*iI;      /* how else might we do this? */

    /* Echo data, print square */
    printf ("The square of %d is %d \n", iI,iJ);

    return 0;
}
```

A.2 – newtonsPicnicTableMacro.c – typical project example

```
/* newtonsPicnicTableMacro.c
 * (c) 2003 g.m.dick
 *
 * Program Description:
 *  - Inputs: required table dimensions (height, width)
 *            required manufacturing tolerance (max relative error)
 *  - Outputs: several manufacturing dimensions as specified in the
 *             data dictionary
 *  - Function: This program determines needed manufacturing dimensions
 *              from required table dimensions.  The key element of the
 *              design is the solution of the equation:

 *                  f(theta) = h*cos(theta) + b - w*sin(theta)
 *              This equation is solved using the Newton Raphson method.
 *
 * Synopsis:
 *  - read required table dimensions and manufacturing tolerance
 *  - seed the search with an initial guess for theta
 *  - while solution not found (i.e., error > errorMax)
 *    - compute new guess using Newton Raphson
 *    - compute new error
 *  - compute manufacturing dimensions
 *  - print results
 *
 * Reference: "Numerical Methods with MATLAB" Gerald Recktenwald,
 * Prentice Hall 2000, pp. 240 - 243
 */

/* ---------------------------------------- preprocessor directives */
```

```
#include <stdio.h>
#include <math.h>

// #define DEBUG
// #define DEBUG2

#define PI 3.1415927
#define OUTFILE "newtonsTable.txt"

/* macros that "simplify" some statements below */
#define F(Theta)           dHeight*cos(Theta) + dLumber - dWidth*sin(Theta)
#define FPRIME(Theta)      -dHeight*sin(Theta) - dWidth*cos(Theta)

/* ------------------------------------------------------------- main */
int main(void)
{
 /* ------ Data Dictionary ---------------------------------------- */
 /*    type   name               in/out unit  description              */
 /*    ====   ====               ====== ====   ========================== */

       double dWidth;            /* in  (in)  table width               */
       double dHeight;           /* in  (in)  table height              */
       double dLumber;           /* in  (in)  lumber width ("b" in the ref.)*/
       double dD1;               /* out (in)  manufacturing dimension   */
       double dD2;               /* out (in)  manufacturing dimension   */
       double dB2;               /*     (in)  manufacturing dimension   */
       double dC;                /* out (in)  manufacturing dimension   */
       double dA;                /* out (in)  manufacturing dimension   */
       double dThetaOld;         /*     (rad) old solution guess        */
       double dThetaNew;         /*     (rad) new solution guess        */
       double dThetaD;           /* out (degree) manufacturing dimension */
       double dError;            /*     (unitless) relative error       */
       double dErrorMax;         /* in  (unitless) max relative error   */
                                 /*                manufacturering tolerance */
       double dF;                /*      value of f(thetaOld) - function    */
       double dFprime;           /*      value of f'(thetaOld) - derivative  */
       double dAlpha;            /*     (rad) intermediate solution variable */

       FILE   *pFoutFile;  /* file pointer */

       pFoutFile = fopen(OUTFILE, "w");

       #ifdef DEBUG        /* if enabled ... quick test data */
       dWidth    = 32.0;
       dHeight   = 29.0;
       dLumber   =  3.5;
       dErrorMax = 0.0001;
       #else

       /* read required table dimensions and manufacturing tolerance */
       printf("Enter table width (in): ");
       scanf("%lf", &dWidth);
       printf("Enter table height (in): ");
       scanf("%lf", &dHeight);
       printf("Enter lumber width (in): ");
       scanf("%lf", &dLumber);
       printf("Enter maximum relative error (0.0 - 1.0): ");
       scanf("%lf", &dErrorMax);
       #endif


       /* seed the search with an initial guess for theta */
       dThetaOld = PI/3.0;       /* from rough graph */
```

```
	do  /* while solution not found (i.e., error too big) */
	{
		/* compute new theta guess using Newtom Raphson */
		dF		= F(dThetaOld);
		dFprime		= FPRIME(dThetaOld);
		dThetaNew	= dThetaOld - dF/dFprime;
		/* update error */
		dError		= fabs(dThetaOld-dThetaNew)/dThetaNew;

		#ifdef DEBUG2
		printf("%f %f\n", dError, dThetaNew); /* trace the solution */
		#endif

		/* prepare for next iteration */
		dThetaOld = dThetaNew;
	} while (fabs(dError) > dErrorMax); /* continue while error too big */

	/* compute manufacturing dimensions */
	dB2		= dLumber/sin(dThetaNew);
	dD2		= (dWidth - dB2)/(2.0*cos(dThetaNew));
	dAlpha		= PI/2 - dThetaNew;
	dA		= dLumber/tan(dAlpha);
	dC		= dLumber/tan(dThetaNew);
	dD1		= dD2 - dA - dC;
	dThetaD		= dThetaNew*180.0/PI; /* degrees */

	/* print results */
	printf("\n\nSpecified Dimensions: \n");
	printf("  Height (in) \t\t = %7.2f\n", dHeight);
	printf("  Width (in) \t\t = %7.2f\n", dWidth);
	printf("  Lumber width (in) \t = %7.2f\n", dLumber);
	printf("Computed Manufacturing Dimensions \n");
	printf("  Theta (degrees) \t = %7.2f \n", dThetaD);
	printf("  a  (in) \t\t = %7.2f \n", dA);
	printf("  c  (in) \t\t = %7.2f \n", dC);

	printf("  d1 (in) \t\t = %7.2f \n", dD1);
	printf("  d2 (in) \t\t = %7.2f \n\n", dD2);

	fprintf(pFoutFile, "Newton's Picnic Table \n\n");
	fprintf(pFoutFile, "Specified Dimensions: \n");
	fprintf(pFoutFile, "  Height (in) \t\t = %7.2f\n", dHeight);
	fprintf(pFoutFile, "  Width (in) \t\t = %7.2f\n", dWidth);
	fprintf(pFoutFile, "  Lumber width (in) \t = %7.2f\n", dLumber);
	fprintf(pFoutFile, "Computed Manufacturing Dimensions \n");
	fprintf(pFoutFile, "  Theta (degrees) \t = %7.2f \n", dThetaD);
	fprintf(pFoutFile, "  a  (in) \t\t = %7.2f \n", dA);
	fprintf(pFoutFile, "  c  (in) \t\t = %7.2f \n", dC);
	fprintf(pFoutFile, "  d1 (in) \t\t = %7.2f \n", dD1);
	fprintf(pFoutFile, "  d2 (in) \t\t = %7.2f \n", dD2);

	fclose(pFoutFile);

	return 0;
}
```

Appendix B – MATLAB Language Example

```
% piston.m
% (c)2000 g.m.dick
% This script determines the linear position of a piston within a
% cylinder as a function of crankshaft angle
```

```
% data dictionary
% ____________________________________________________________________
% variable    type        i/o | description
% --------------------------------------------------------------------
% system parameters
%   L1        scalar      in  | length of connecting rod (inch)
%   L2        scalar      in  | length of crankshaft arm (inch)
% simulation paramenters
%   thetaMax  scalar      in  | maximum chankshaft angle (radians)
% simulation variables
%   d         vector     out  | linear position of piston
%   theta     vector      -   | crankshaft angle (radians)
%   thetaD    vector     out  | crankshaft angle (degrees)
%   phi       vector      -   | auxiliary angle between connecting
%                             rod and center-line (degrees)
%   numPoints scalar      -   | the number of simulation points

%   synopsis:
%    - read system and simulation parameters
%    - compute piston position vs crankshaft angle
%    - plot results

% read system and simulation parameters
clear
L1 = input('Enter length of connecting rod (inch): ');
L2 = input('Enter length of crankshaft arm (inch): ');
thetaMax = input('Enter maximum simulation angle (radians): ');

numPoints = 100; % graphing parameter

% compute piston position vs crankshaft angle
theta = [0.0 : thetaMax/(numPoints-1) : thetaMax];  % theta vector
phi = asin((L2/L1)*sin(theta));                     % auxiliary angle
d = L1*cos(phi) + L2*cos(theta);                    % piston position

% plot results
thetaD = (180./pi)*theta;
plot(thetaD, d);
Title('Piston position vs Crankshaft Angle - g.m.dick');
xlabel('Crankshaft angle (degrees)');
ylabel('Piston position (inches)');
disp('Locate comment on graph with crosshairs.');
gtext(['d = ', num2str(L1+L2), '(inches) is top dead center']);
% gtext() is a 'fancy' alternative to text()
```