



# ETP Cover Sheet

Title of ETP	Documenting Computer Programs
Name of IISME Fellow	Tim Brown
Fellow's year-round email	tbrown@srvhs.org
Sponsor Company	PG&E
Name of Mentor	Dave Bradley
National Board Certificate Area	Technology
I give IISME permission to share my ETP as part of its professional development program for teachers.	Signature of the Fellow: Timothy F M Brown

Category	<p style="text-align: center;"><i>Curriculum</i></p> <p>Subject: Math      Science      <u>Technology</u>      _____</p> <p>Level:    Elem      Middle      High      Other</p> <p style="text-align: center;"><i>Staff Development</i></p> <p style="text-align: center;"><i>Describe</i> _____</p> <p><i>Other</i></p> <p style="text-align: center;"><i>Describe</i> _____</p>
Objectives	<p>?? Students will explain the necessity of software documentation</p> <p>?? Students will apply the 18 criteria of good software documentation</p> <p>?? Students will evaluate other's software documentation using the 18 criteria and the rubric</p>
Abstract (50 words or less)	<p>This ETP consists of:</p> <p>?? a commentary on the essential nature of complete, clear and timely documentation</p> <p>?? 18 good internal documentation practices</p> <p>?? suggestions for external documenation</p> <p>?? a rubric for assessment of computer projects</p> <p>?? a "saved by documentation" article</p> <p>?? recommendations for printed documentation and presentations</p>
<b>Describe how your ETP aligns with the National Board Standard stated in your proposal.</b>	Technology ETP standards are not yet established
Resources Needed	Normal Computer Science classroom resources
Evaluation/Assessment Measures Used	Included rubric
Formatting specifications	PC <u>X</u> or Mac _____ <b>(Must be in Word or Text Format)</b> Software used    Word 2000, PowerPoint 2000
Submitted Copy	<b><i>Soft and hard copy due to peer coach by the end of the summer fellowship. Also, a copy of the cover sheet signed by a school site administrator submitted to IISME Oct.1, 2002 to receive \$300 grant.</i></b>
Mentor signature and comments	
Administrator's signature and comments	

# ***Documenting Computer Programs***

**An IISME Education Transfer Plan  
by *Tim Brown*  
*San Ramon Valley High School,  
Danville, CA***

B4

***Sponsored by:***



## **Table of Contents**

Table of Contents.....	2
Why Document?.....	3
18 Good Documentation Practices.....	4
Suggestions for external documentation- the user's guide to the student's project.....	6
Rubric for Assessment of Computer Programming Projects.....	7
Appendix A: Saved by Documentation.....	10
Appendix B: Some Recommendations for Printed Documentation and Presentations.....	12

# Why Document?

- ?? People other than you will read your code and they must be able to understand it and, if necessary, modify or extend it.
- ?? A clear coding style can improve your efficiency as a programmer as you can quickly determine the purpose and reusability of a code that you wrote in the past.
- ?? Explaining algorithms forces you to think about how the program should work.
- ?? Documenting valid value types and ranges pretty much specifies the data required for a good test data set.

# 18 Good Documentation Practices

1. Document the program itself. Include the following:  
    /\* Programmer Name:       **Julia Sorensen**  
      Class:                **APCS Period 3**  
      Program Name:        **Project 8-5**  
      Description:         **The user guesses the computer's 2-digit number.**  
    \*/  
    Your project will not be graded unless it is clearly identified!
2. Use self-documenting identifiers. The names of variables and constants should tell the reader what they contain, the names of functions should tell the reader what they do.
3. Be consistent. If **i** is used as a loop variable in one function, don't use it as something else in another function. If you use comments at the end of a line, line them up if possible. Be consistent with your comments; use either complete sentences or understandable explanatory phrases.
4. Use **white space**. If your IDE doesn't do it for you, type spaces around all operators. Instead of **total=amount+tax** use **total = amount + tax** which is much easier to read.
5. Use **identifier conventions**. Ours are:
  - a) variables start with a lowercase letter: **salesTax**
  - b) functions start with an uppercase letter: **GetInput()**
  - c) constants are all uppercase letters with underscores to separate consecutive words: **TAX\_RATE**
6. Use **indentation** to show program structure. Be consistent in your choice of indentation, such as one tab stop, especially with structures like *if* statements, loops, etc. Indent all lines inside control structures for readability. A new level of indentation should be used at every level of statement nesting in your program.
7. Separate functions by putting **blank lines** above and below them and do the same for control structures within a function. Also add a blank line between your variable declarations and the body of the function.
8. Do not use "**magic numbers**": Raw numbers in code seldom communicate their purpose. Use constant values or enums to encapsulate the meaning of the number in your code.
9. **Write self-documenting code**: Documentation occurs in two places: explicitly and implicitly. Code that is well written is its own comment. Poorly written code that needs a long comment should be re-written. On the other hand, complex algorithms are, by their very nature, complex; these are the types of functions that need explicit comments.
10. **Avoid obscure programming language constructs and reliance on language-specific precedence rules**. It is often better to force precedence by use of parentheses since this

leaves no doubt as to meaning. In general, if you had to look up some rule or definition, your readers most likely will too. Whenever you do write a difficult expression, or some other tricky business, if you found it difficult to write you should expect that it will be difficult to understand. So, add a comment.

11. **Comment to clarify, not to belabor the obvious:** Include a comment at the beginning of each function to explain its purpose. If a function contains several major steps or parts, there should be comments (called signpost comments) separating the function into each major part. However, do not comment every line of code.
12. Never use **goto** statements.
13. Use **multiline if** statements rather than single-line if statements to make your code more  
Use **if (a > b)**  
    **c = a;**  
    **else c = b;**  
rather than **if (a > b) c = a; else c = b;**
14. Use perfect spelling and grammar throughout your code, especially when a user can view it. If necessary, copy and paste your code into a word processor, like MS Word, and run a spell-check before you turn it in.
15. The functions of a program define the structure of that program. Every function should be preceded by a comment describing the purpose and use of that function. Describing that task before writing the function can help you think through the logic, especially if you describe what data is passed to the function and what data it passes back.
16. Functions should be short enough so that their full meaning can be grasped as a unit by the reader. Normally, one page, about 50 lines, of code represents the limit of intellectual complexity that a reader can grasp. Each function should have a comment block similar to that of the program header, with standard sections such as function name, parameters, description, and calling routines.
17. If you decide to modify a function during program development, be sure to modify the comments as well to keep them accurate
18. Some programmers like to surround the entire comment block by asterisks to set it off. I don't recommend this practice, nor putting comment braces at the beginning and end of each line in the comment block. While doing so will make the program more attractive, this practice also makes it difficult to modify the comment block and you want your documentation should be easy to maintain!

# Suggestions for external documentation- the user's guide to the student's project

The Software Turnover Template used by the New Customer Connections division of Pacific Gas and Electric provided this idea for this template.

The User's Guide should:

- ?? let the user know the software's system requirements,
- ?? help the user get the software installed and running,
- ?? let the user know what the capabilities of this particular version are
- ?? make suggestions for changes

Section 1: Give a general description of the purpose of the software.

Section 2: Explain any environment requirements:

- PC/Mac?
- Processor limitations
- Operating System
- Required hardware such as sound card, scanner, etc.
- Other required software such as Excel, Winamp, etc.

Section 3: Explain the installation and startup procedures.

Section 4: List any known limitations and defects

Section 5: List any suggested modifications or extensions

# Rubric for Assessment of Computer Programming Projects

- ?? It is important for students to be provided with specific expectations for assignments. They need to understand that evaluation is based on fulfilling requirements, not on whether the teacher likes the student or not. When the students are provided with expectations in the form of a rubric, they are able to assess their own work. This ability generally increases the students' motivation to produce higher quality work.
- ?? It is also helpful to refer to a rubric after an assignment has been evaluated and handed back to the students. If a student asks, "Why did I get this grade?" the teacher can refer the student to the relevant category of the rubric.
- ?? A rubric can also be used for peer evaluation, allowing the students to more fairly and objectively grade one another's work.

# Scoring Rubric for Computer Programs

Criterion	Exceptional	Good	Adequate	Unsatisfactory
<b>Correctness</b>	Program executes without errors, handles all special cases, and contains error-checking code. Thorough and organized testing has been completed and output from test cases is included	Program executes without errors, handles most special cases and thorough testing has been completed	Program executes without errors, handles some special cases and some testing has been completed	Program does not execute due to errors, no error checking code is included and/or no testing has been completed
<b>Algorithm</b>	Program algorithm is easy to understand and maintain Programmer has analyzed many alternate algorithms and has chosen the most efficient and has included the reasons for the solution chosen	Program algorithm is efficient and easy to follow. Programmer has considered alternate algorithms and has chosen the most efficient	Program uses an algorithm that is easy to follow but it is not the most efficient Programmer has considered alternate algorithms	Program uses a difficult and/or inefficient algorithm Programmer has not considered alternate algorithms
<b>Readability</b>	The code is exceptionally well organized and very easy to follow.	The code is fairly easy to read.	The code is readable only by someone who knows what it is supposed to be doing.	The code is poorly organized and very difficult to read.
<b>Reusability</b>	The code could be reused as a whole or each routine could be reused.	Most of the code could be reused in other programs.	Some parts of the code could be reused in other programs.	The code is not organized for reusability.
<b>Documentation-Identifiers</b>	All identifiers are clearly and effectively documented including descriptions of all variables	Most identifiers are clearly and effectively documented	Some identifiers are documented	No identifiers are clearly and effectively documented
<b>Documentation-comments</b>	For each function, the specific purpose is noted as well as the input requirements and output results. All non-simple code is clearly explained.	For each function, the purpose is noted. Most non-simple code is clearly explained.	For each function, the purpose is noted. Complex code is clearly explained.	Functions and non-simple code are not documented.
<b>Delivery</b>	The program was delivered on time.	The program was delivered within a week of the due date.	The code was within 2 weeks of the due date.	The code was more than 2 weeks overdue.
<b>Efficiency</b>	The code is extremely efficient without sacrificing readability and understanding.	The code is fairly efficient without sacrificing readability and understanding.	The code is brute force and unnecessarily long.	The code is huge and appears to be patched together.

Criterion	Exceptional	Good	Adequate	Unsatisfactory
Error Handling	Complete Testing and error handling	Attempts were made to catch all errors, though some remain due to limited testing.	Errors were not handled correctly so that program fails under some input.	No error handling
Use of object oriented programming methods	Good use of structures including advanced features of structures such as overloading.	Used structures appropriately, but did not apply any advanced methods such as overloading	Good choice of structures but poor implementation of them.	Did not use structures appropriately. Poor choice of structures or implementation of them.
Use of complex data structures	Appropriate use of advanced data structures.	Appropriate use of vectors/arrays. Use of these within a struct.	Some use of vectors/arrays.	Did not use vectors or arrays where needed.
User input	All input is prompted appropriately and appropriately edited.	All input is prompted appropriately and some input is edited.	Some input is prompted but not edited	Input is poorly prompted and not edited
Output	All output is clear, well labeled and attractively presented.	All output is clear, labeled and fairly well presented.	Most output is clear and labeled but not attractive.	Output is confusing and unlabeled.
PowerPoint Presentation	Presentation including pictures and text. Used slide transitions or animations. Spoke clearly and comfortably.	Presentation including pictures and text. Used slide transitions or other features but poorly. Speech was clear but not comfortable.	Presentation including text but no pictures. Used slide transitions or other features but poorly. Speech was too informal or included a number of partial sentences.	Limited PowerPoint presentation, basically with no changes from the template. Nervous, uncomfortable presentation.
Group dynamics	The group draws on strengths of each member and all feel valuable.	Good methods of communication have developed. Respect of members.	No clear leadership or organization. Some members do most of the work.	No appreciation of different member's abilities. Members drop out of group.
Individual Student	Student was actively involved in all parts of the project and also worked independently on parts of the solution. Communicated well with the rest of the group.	Student was involved in all parts of the project, but at times had limited input. Observed at times when action was needed. If the student had to miss team meetings, then the student helped the team out with individual work.	Student missed some days and often was just an observer.	Student had little involvement and when was involved was just an observer.

# Appendix A: Saved by Documentation

## A Real World Story Courtesy of Addison-Wesley's *Innovations* magazine

Not too long ago, we were using a commercial “terminal emulator,” a program that simulates in a window an old-fashioned terminal in which to run a command interpreter or telnet session. By default, the emulator didn't keep text that scrolled off the top of the screen, but a pull-down menu of options allowed one to specify a number of lines of “buffer” to store text that would otherwise be lost. At one point, we were using the program to look at a large file, so we went to the menu and set the number of lines of buffer to 999, the largest value that would fit in the three-digit field. When we clicked OK, the window vanished, leaving only its title bar! Clearly, the program had a bug and, although it accepted the large value, it didn't handle it correctly.

Because the button to activate the pull-down menu was missing, there was no way to restore the buffer size and, we hoped, recover the window. So we deleted the window using the delete button on the title bar.

But we still wanted to use the program, so we started it again. Unfortunately, the program had recorded the large buffer value in permanent storage, so the new instance of the program was also represented by just a title bar. What had started as an attempt to read a file had now turned into a puzzling debugging problem. How could we get the program back?

We spent a long time searching unsuccessfully for the location of the file that recorded the size of the buffer, hoping to reset it. Then we tried to find some other way to access the program's data, such as by a separate interface to its parameters, but that too was fruitless.

Then we realized that the problem was likely to be in the display code for the text of the window itself and that, even though it was invisible, the pull-down menu button might still work if we could only find it. So we carefully and methodically started clicking the mouse along the bottom edge of the title bar until--aha!--the options menu popped up. We were then able to restore the buffer size, whereupon the program returned to the screen and we were able to continue working.

After this adventure, we read the documentation for the program, where we learned that the valid buffer size was at most 399 lines. We did a little more experimentation, now that we knew how to get the program back, and indeed 399 lines worked fine, while 400 caused the window to disappear.

As with any good programming war story, there are lessons in our little adventure.

This silly bug would have taken far less time to fix than it cost us to uncover; who knows how much time has been wasted by other users? It would have taken even less effort to have prevented it in the first place by a simple check for valid input. Writing software correctly is the

best way to avoid problems, and one of the most important parts of being correct is defending against invalid input.

The program wasn't tested thoroughly either. An effective way to test code is to exercise it at its natural boundaries. The program's author must have known that 399 and 400 were critical values, since 399 represented the biggest possible buffer and 400 the smallest illegal value. If the program is going to fail, it will likely fail at one of those, along with small values like 0 and 1. It's easy to test such cases, often while the code is being written.

But when software has bugs, there are ways to track them down and fix them. Debugging is a process of narrowing down the possibilities, finding the critical input values that cause the problem, eliminating things that can't be related to the problem. Each debugging run is an experiment that tests a hypothesis. This example was hard because we didn't have source code, and the error symptom was that the display shrank to nothing!

A program should do sensible things, so that for typical applications it can be used without documentation; this is especially true for programs with graphical interfaces, where no one ever reads the instructions. The program could have been written without limits on the buffer size (easiest to use, and no documentation needed), or it could have included a "maximum 399" label near the input box (documentation visible when it's needed), or it could have truncated a too-large value to the maximum (silently enforcing a limit), or it could have checked the input and popped up a message box when the input is invalid (irritating but valid). Any of these choices is clearly better than proceeding with bad input.

This article originally appeared in Addison-Wesley's *Innovations* magazine.

Copyright © 1999 Lucent Technologies. All rights reserved.

# Appendix B: Some Recommendations for Printed Documentation and Presentations

Courtesy of Rubric, *Tips for Creating World-ready Documentation*,

## Text

1. Develop a standard glossary, and translate consistently across documentation sets.
2. Take into consideration relevant external glossaries, such as industry-standard terminology. If dealing with an after-market product, consistency should be maintained with the main product.
3. Establish standard linguistic style guidelines.
4. Maintain consistency between the documentation and the software's UI.
5. Use clear, concrete language.
6. Avoid slang and idioms.
7. Define acronyms and abbreviations.
8. Avoid cultural references, such as gender-specific roles, humor, ethnic or historical references.
9. When choosing to include cultural references, encapsulate such content into well-defined locations.
10. Customize formats to fit the target country, for example date and time formats and currency formats.
11. Confirm that your localization partner has a leveraging strategy, including any translation memory implementations, for reuse of translations and graphics that have already been localized. This strategy should accommodate both new product releases as well as updates during the localization process.

## Layout

1. Establish standard layout guidelines, clearly defining the use of design styles
2. Consider the development of a universal template to reduce layout tasks and time during localization.
3. Use standard fonts that are well supported by most output devices and are easily available. If special fonts are used, it is critical to communicate this to the localization team up-front, prior to localization.
4. Save call-outs, as well as text within tables, as text rather than as graphical elements
5. Leave sufficient white space to accommodate text expansion during translation.
6. Anticipate font type and size modifications for some character sets.
7. Anticipate vertical expansion of the text for some character sets, for example, Traditional Chinese.
8. Avoid formatting characters and paragraphs manually. Instead, define an appropriate style to automate the task and ensure consistency.
9. Automate header and footer text, such as chapter titles and page numbers. This information can change during repagination of the localized documentation.

## Graphics

1. Use standard, easily available applications to create graphics in target languages.
2. Avoid using text in icons and graphics, if possible.
3. If text is included, try using callouts or captions with the text component in the DTP application rather than in the graphics file. Otherwise, if the text is within the graphic itself, save the text as editable text, not as pixels.
4. Assume that callouts and text within graphics will increase in length when translated. Leave sufficient white space to accommodate this text expansion.
5. Link graphics rather than embed them whenever possible.
6. For screen captures, note the computer configuration and software settings, such as color depth and screen resolution. Give this information to the localization team.
7. For screen captures requiring complex setup and navigation, identify these and supply regeneration instructions to the localization team.
8. Avoid using culturally specific icons and graphics.
9. Avoid using representations of people and animals in icons and graphics.

Courtesy of Rubric, *Tips for Creating World-ready Documentation*,  
<http://www.rubric.com/>.