Documentation: The Necessary Evil

Robert M. Gignac
Motorola Information Systems
9445 Airport Road
Brampton, Ontario (Canada)
L6S 4J3

Abstract

The systems we are developing today will become our 'foundations for the future'. The cornerstone of our foundations should be an array of clear, concise, and well written documentation. Do not misinterpret the title, documentation is far from 'evil', the keyword is 'necessary'. The 'evil' designation is in the eyes of the analysts and programmers in charge of developing our systems. Of the many steps involved in a project, documentation is the one we all agree is required, but is the one nobody wants the responsibility of writing, let alone maintaining. To further undermine our foundation, documentation is the area most likely cut from a project as costs rise and deadlines draw near.

1.0 Introduction

A tremendous amount of time, resources and knowledge must be expended in the development, programming and testing of computer based systems. The results of these efforts must be carefully organized so the myriad of details related to the programs within these systems can be recorded in a clear and orderly fashion. This means significant information about the programs and the systems must (not should!) be written and stored in a clear and concise fashion. The creation and maintenance of this information is called DOCUMENTATION. Documentation is a vital part of every system, and in order to build 'foundations for the future', documentation must be one of the cornerstones of that foundation.

Unfortunately, as vital as documentation is, it remains possibly the weakest link in the computer systems field. If you need confirmation of why this is so, perform the following informal survey of your programmers and analysts: ask them to list their 15 favorite tasks on a sheet of paper and when they are done, sit down and review them. More often than not, the words 'documenting systems/programs' will fail to appear on the paper. Why? Documentation isn't trendy, machine oriented, technically stimulating, and most programmers and analysts consider it beneath them to document. When your staff approaches the task of documentation with this attitude it is easy to see why documentation has been neglected, completely ignored, or written in a haphazard manner in many DP shops.

There are two hazards encountered when writing about the subject of documentation. The first hazard is the fact the term encompasses a variety of different forms, all having different meanings to different people. G. Prentice Hastings and Kathryn King in their book _Creating Effective Documentation for Computer Programs_ make reference to the following levels of documentation: Reference Charts, Operator's Guide, User's Training Document, Student Workbooks, Reference Manuals, Management Guide, System Administrators Guide, Logic Diagrams, Microcode List, Technical Manuals, User's Manual, and Installation Guide. I am not about to cast judgement on how much of this documentation is really required or even practical to create. A decision on this matter would depend on the size of the system you are creating and the needs of your individual company. The second hazard arises from the subjective nature of documentation. Like anything subjective, changes in documenting procedures may make things better, worse, or leave them much the same - and often there are no easy answers about what to do. Every change or implementation of new ideas involves trade-offs - What type of documentation? Structure? Amount? Who will write it? - which are significant in determining what needs to be done. This paper is, therefore, by no means an attempt to provide definitive 'yes/no' answers. However, if this paper proceeds to equip you with the knowledge required to reach your own conclusions about why documentation is not evil, it will have done its job. The scope of this paper will be two areas of the documentation field that I feel are key, yet are often left incomplete, incorrect or nonexistent: Program Documentation and Application Software Documentation.

2.0 Program Documentation

It remains a fact that even today many programs are delivered for which there is little or no documentation to accompany them, or the documentation that does exist bears no resemblance to the source code. On the surface you could ask "So what?", because the program/systems will run whether or not the documentation exists. As long as the program runs, this remains true. Unfortunately, I have seen very few programs that ran forever without 1) going crash in the night, or 2) requiring some form of maintenance. Addressing the first issue, we all know that it will be late one Friday night that the XYZ analysis program which has run flawlessly for 27 months will abort with some cryptic Image error, requiring someone to dig out the program documentation. If we can find it, we are a step ahead of the game. If it really matches what the program is doing, your maintenance staff will love you. More likely, you will have documentation that is either: a) incomplete, or b) lacking the revisions Peter Programmer made last June before he left the company. Addressing the second issue, I have yet to see in my relatively short experience in the field (6 years) any program that has run for 2 years or more without ever requiring some form of modification. Fixes to old bugs, fixes to new bugs caused by fixes to old bugs, report format changes, company policy changes, etc. The reasons why you might change code are endless, but you can be sure that if your program is important it will need to be changed at some point. There is also a third possibility, which I have seen only once, that may be valid in your shop -- extremely accurate documentation of very poorly written programs. While not as severe a problem as the first two, it deserves some recognition as well.

Given all these potential problems, don't despair, you won't be the first to encounter them (or the last). However, to ensure that they don't happen more than once, we should discuss some solutions. The most obvious (and since it is obvious, it is probably the least effective) is to write standards and directives for what you expect to find in your shops program documentation and make compliance with these directives a requirement for continuing employment. We will assume you are doing this already, and if it is working, fine -- but if it isn't, we require some additional tactics. More humane and perhaps more sensible might be to seek out programming methodologies with built-in documenting enhancements. But don't be mislead here: structured programming by itself (regardless of language) won't solve the documentation problem (despite what your programmers say...). Choose whatever documentation method you feel is best for your shop, then it is up to management to insist that the guidelines are followed. It still appears that after all this time there is no mechanical substitute for old-fashioned management control.

2.1 Program Documentation - Implementing Change

In order to get your DP staff to follow the new management guidelines, you will have to get them to change the standard and widely held view of documentation: "Those who can, do, those who can't, document". Do not expect this change to happen easily, as people resist change in any number of ways, and for many different reasons. Key among them are lethargy and fear. Phillip Metzger in his book

Managing Programming People outlines the following options as possibilities:

1) Serious Threat: "Do it or I'll kill you"

2) Appeal to Self-esteem: "You don't want the people in Linda's department to make us look bad, do you"

3) Opportunity: "We finally have a chance to look into this new opportunity to improve ourselves"

4) More Opportunity: "Here's a chance to blaze a trail for the rest of the department"

5) Bribery: "Do this and I'll remember it when salary review time rolls around"

On the surface, these options appear quite humorous, but they are some of the methods you may have to use if you are to change the opinions of your staff towards documentation. Believe it or not, people will actually resist the opportunity to increase their skill levels or improve their credentials in this area (ask my former supervisor). In part, this may be due to the mistaken belief that good documentation skills are not seen as a marketable asset, as are courses in structured design, database methodology, 'C' programming, etc. As an analyst or manager, you may face an additional problem. You may have programmers reporting to you (whom you can convert), but you in turn report to someone who may hold the same beliefs as your programmers. It will probably be easier to get your technical people to try new things than it will to get management to join. Management's reasons may be as follows:

1) If the group spends too much time on this, other projects may fall behind (yet time for this should have been planned in advance...)

2) How do we know it will do any good?

The best way to answer managements concerns would be by analogy. You have to look at program documentation as insurance. If nothing ever goes wrong at your site then the effort may appear to be wasted. On the other hand, when things do go wrong (and they will...) your first line of problem solving will be a referral to the program code and the accompanying documentation in order to: a) find the cause of the problem, b) fix it, and c) get the system rolling again. It is because of this fact that documentation is often viewed in nebulous terms, you can't tell management how much value documentation has until a crisis arises, and by then it is too late to start documenting.

2.2 Program Documentation - Assembling the Material

Hopefully, we have determined that program documentation is actually required. We will assume it doesn't exist, and that attitudes on the part of programmers and management can be changed. Just what kind of information should we create? The following chart summarizes one of many possible alternatives for creating a program documentation manual.

1) Title Page	The title page should contain the program name, system of which the program is a part, original programmers name, and the date released to production.
2) Revision Page	The revision page is required to document the history of the program. Contents should include the name of the original programmer, date released to production, estimated time to complete programming. On this page in chart form, provisions should be made to document all subsequent revisions, descriptions of them, names of the parties responsible for them, and the date the revised program was released to production. In order to ensure this page is always up to date, do not allow programs to be moved into the production environment until it has been verified that the revision has been documented.
3) Abstract	The program abstract should contain a general purpose and description of the program, frequency of use, input and output files, subprograms called, and a list of programs that are prerequisite for this one to run.
4) System Flowchart	The system flowchart documents the flow of data through the system, providing a visual means of identifying input and output files used in the required steps of the total processing cycle. (There will be those who feel this process is becoming obsolete.)

5) Logic Description	This section should provide a detailed description of the program including special editing performed, sequence checking, reasonableness checks, tables used in the program, special forms required, and operator instructions. The detailed program logic must be illustrated using program flowcharts, decision tables, or pseudo code.
6) Test Data	A listing of the test data used in testing the program, and a sample of the program output should be provided in the documentation manual. Test data should be sufficient to test all routines within the program.

At Motorola Information Systems we are using a combination of the items mentioned in the above list as shown in Fig. #1, #2 and #3 (see appendix). These documents must be completed for every program released to production, and no program will be run in the production environment before these pages are verified to be complete. These pages, as good as they may be, are only half of the battle. The remaining program documentation must be carried out in the program code itself.

2.3 Program Documentation - Using your code

As mentioned earlier, some programming languages lend themselves to documenting. COBOL for example can be somewhat self documenting if certain standards for documentation are enforced. Require COBOL programs to contain a purpose, description and brief history in the REMARKS section of the code. Require that sections or paragraphs be commented if the paragraph performs complex routines or calculations that would not be obvious to someone unfamiliar with the code. Require that all COPYLIBS and SUBPROGRAMS be identified as in Fig. #4 (see appendix).

In case you feel that I am the only person 'crazy' enough to feel this way, I offer the following quote: "In my opinion, there is nothing in the programming field more despicable than an uncommented program. A programmer can be forgiven many sins and flights of fancy; however, no programmer, no matter how pressed for time, no matter how well intentioned, should be forgiven for an uncommented and undocumented program". This quote comes directly from Edward Yourdon, author of <u>Techniques of Program Structure and</u>

Design. Do keep in mind however, that commented code is not an end to itself, as good comments are not a substitute for bad code, nor is good code a substitute for lack of comments. Program code obviously tells us what the program is doing, but it cannot tell us why it was done in a certain fashion. Before you decide that code documentation will be the key to solving your problems, be prepared to hear the following excuses from your staff:

- I don't have enough time

- My program is self-documenting

- Any competent programmer can understand my code

- This is a one shot program, its not worth it

- The program will change dramatically during the testing and debug phase, so any documentation will be useless by the time the program is finished (if this is the case, perhaps you should question their design skills before they start to code?)

- I understand the code, I'll be here to fix it

- My programs will take too long to compile

- Who will read the stuff anyway?

We have all heard these arguments before, and perhaps we have even used one or two of them on occasion. In order to combat them we may choose any of the techniques for change outlined earlier by Metzger, or we may choose to implement documentation as a philosophy across a department. Actually putting program documentation methods into practise is not that difficult if it is kept in mind at all times (perhaps a system welcome message that reads 'Have you documented your code today?'). Good documentation habits are generally best exemplified by personnel who work for consulting firms. Reassignment to another task is a common occurrence, and for the success of the project (and perhaps the firm), it is imperative that the next person be able to pick up the system where the last one left it. Arguments will be raised here as well, because people feel they work in a relatively stable environment, so this precaution is not necessary. One only has to look at DP turnover rates to see why it is necessary. The average length of employment with one firm is less than three years, and the easiest way to turn programs or systems over to new people is with decent accompanying documentation.

2.4 Program Documentation - Reducing Maintenance Costs

Programs spend most of their life being maintained. Often, considerably more time and money is put into extending and changing programs than was spent at the initial development. If this surprises you, it shouldn't. New systems and their associated programs change the environments in which they are used. In turn this changes the way they work, and when work habits change, changes in the system are a natural result. Barry Boehm in his book Software Engineering Economics reports that DP shops are currently spending over 50% of their budgets on maintaining their existing systems (see Fig. #5 in appendix). Over the past 10 years this figure has increased by about 25%, and will probably continue to increase in the future. If you wish to reduce your costs (and who doesn't?), you can use reducing your maintenance costs as a selling point for program documentation. Below is a list of why maintenance costs are so high, and it is easy to see how program documentation may reduce these costs.

- Often programs are released to production that still have a significant number of bugs. Due to this, what is often called maintenance is really just an extension of the testing phase.

- When maintenance is required, the original programmer has often left the company, or has been reassigned to a different project.

- Programmers do not often view maintenance as glamourous work.

- Most people have difficulty understanding other peoples code.

- Documentation that accompanies most programs is just short of awful. Some testing in university settings has indicated that maintenance programmers would be better off removing all of the comments accompanying a program and then trying to find bugs or implement improvements. Because of this, many firms are now paying the price for poor documentation standards of the past, as their maintenance times and costs increase.

2.5 Program Documentation - A Dissenting Opinion

As with most concepts in the systems field, there are people who are 'for' the concept and those who are 'against' the concept. I feel I would be remiss if I didn't at least address the viewpoint of the 'against' delegation. John

Boddie in his book Crunch Mode, approaches program documentation as follows, "On some projects there is a rush at the end to produce 'program documentation' -descriptions of the code in the system. This is done in the name of maintenance. What it is, really, is stupidity.". On this issue I must disagree. If the project was properly planned and the documentation completed at each step in that plan, they wouldn't be running around at the end of the project trying to complete program documentation. Mr. Boddie goes on to state that the original programmers design documents, plus the comments that were put in the code should be adequate enough for the maintenance staff to pick up the system and maintain it. "These comments are the 'program documentation'", and project leaders will insist on it as good programing practice. Unfortunately, in the past, project leaders have not insisted on this, and many do not to this day. As for the design documents and program comments being adequate program documentation, could you imagine trying to piece together the relationship of a complex system from the program design documents and the source code?

Don't let your staff, or your management try to avoid the issue of program documentation by using any of the excuses mentioned in this section. Any program or system that is of any value (and why would we bother to create them if they weren't?) will remain active for some period of time, increasing the odds of some other individuals coming into contact with it. Perhaps one of the best ways to impress the importance of this on a young programmer is to give them a 'rats nest' program to maintain, debug and modify (you know, the kind we all used to write). If this is done to them early in their career it can have a strong and beneficial impact on their programming habits. This in turn will only make things that much easier for you to convince them of the benefits of program documentation, and they in turn may help you to convince the rest of your staff.

3.0 Application Software Documentation

Application software documentation (often referred to as the 'users manual') serves as the primary interface between the end user and the application software. Despite the importance of this documentation as a factor in both program and system success, software maintainability, proper system use and user satisfaction, application software is often paid little more than lip service by DP departments. Application documents have long been considered evils of doubtful necessity, and because of this, the manuals that are produced often try the users patience. It would appear that when programmers are good, they are very good; but when they write, they are terrible. The reason for bad writing getting out is the same as for bad programs getting out: tasks aren't planned well enough, plans aren't executed well enough, and the results aren't tested well enough. For these problems to exist, the finger must point at management for letting poor writing and documentation get by them.

3.1 Application Software Documentation - Inherent Problems

The Audience:

It is generally well known that most occupational reading is 'reading-to-do' rather than 'required reading'. Many people only use application documentation to improve performance of seldom performed tasks. Due to this, if the documentation is difficult to understand, users may abandon the written material in favor of alternative methods: trial and error, consulting more experienced users, or forgetting about the whole thing if possible. Users view the application documentation the same way we view documentation from companies whose software we use. If the documentation is poorly written or contains errors and inconsistencies, we attribute the same negative quality to their software. If we feel this way about the documentation we use, why shouldn't the end users feel the same about the documentation we provide for them?

Structure Differences:

The problem that arises here is due to the way documentation is created, especially in smaller DP shops. In many DP shops, the programmers or analysts are responsible for creating the application documentation. Unfortunately, when there are multiple systems being developed by different people, there will be different styles of documentation produced. Differences will appear in terms of layout, scope, wording, technical orientation, etc. Ideally, having access to a technical writer would help simplify the problem. In reality, since most shops cannot afford this luxury, documentation standards should be communicated to all responsible parties so consistent documentation will be produced.

Inadequate Current Documentation:

One of the most prevalent problems with the current state of documentation is the amount of it that is missing (whereabouts unknown), incomplete (lacking relevant information), inaccurate (missing latest revision), or obsolete (program no longer in production). Existing user manuals often lack a sufficient number of relevant examples to accommodate the needs of users. Error codes may go unexplained, and recovery procedures in case of error may be inadequately described.

Resistance to Document:
This topic has been discussed so many times that we should be able to abandon it by now (see section 2.1 on implementing change). DP personnel involved in the software development process are often those responsible for documenting the systems due to their higher understanding of the end product. The problem is that writing is one of the least interesting software related activities and little linkage is perceived between improved documentation and the organizational reward structure. Another part of this resistance to document may come from the basic educational system our programmers are now coming from. In a College or University setting there is no incentive to document your programs as you are the only person who ever has to deal with them. Users manuals are not required, and the whole issue of how educators view documentation can be summed up in a discussion I had with one of my college professors during a 3rd year systems design course. Being naive as I was at the time, I asked Ivan Chapman just what we would do with ourselves once we had been hired by a firm and had completed computerizing every possible activity known to mankind. His response: "Then you document". This attitude clearly makes documentation look like an unnecessary task, something to do once everything else has been completed. It is only in the newer systems texts that the concept of complete system documentation is being covered, and in fact, there is now an entire body of texts dedicated to the topic of creating documentation for computer systems. Eventually, this trend will filter its way into the educational system, and when it does, we should finally be able to hire programmers who do not view documentation as undesirable.

Inadequate Managerial Planning:
Often there is a perceived lack of managerial guidance, policy, support or review of documentation efforts. Efforts to minimize the software development time and cost may occur at the expense of perceived minimum benefits from documentation activities.

Lack of Testing:
You must schedule writing, editing and rewriting of documentation as carefully as you schedule design, programming and testing for the code. Documentation should not be left to the last two days before system delivery. If possible have your documentation written by people who like to write and are proficient at it rather than by the programmers who wrote the code (and who probably don't want

to anyway). As well, you must test your documents. No, you didn't misread that last sentence -- you must test your documents. This can be done through the use of structured walkthroughs and review sessions. The user manual is really the only tangible item that you deliver to your users, and how they view it will often be how they view your system. Test your document by giving copies of it to your systems people who had nothing to do with the design or programming of the system and see if they can follow the logic. If everything makes sense, turn them loose on the test system, as systems people love to try to crash software and they may try things the users wouldn't think of. As well, use key personnel from the user areas if possible, as their understanding of what their system is supposed to do may expose flaws in the design, or highlight areas that need to be more clearly defined in the manual.

3.2 Application Software Documentation - Putting it Together

In order to create good user documentation, we must begin by asking questions. Who will be reading this? How much to they know already? What do they need to know to do their job? What aspects will be confusing to them? Given the task we have to complete, what information should we provide the user with? The following chart summarizes one of many possible alternatives for creating application software documentation.

1) Introduction	The introduction should contain the purpose of the system, objectives it accomplishes, and relationships with other systems.
2) Equipment	If possible, provide pictures of the work environment the user will be in. There are still many cases where we install systems in areas where terminals, printers and modems are foreign objects.
3) Operation	Provide the user with brief descriptions of the following items: Terminals, keyboards, printers and modems. Descriptions should include how to turn all equipment on/off and operating features of each device.
4) Using the Terminal	This section will cover the basic operations required before the system is active. It should cover basic user questions such as: How do I sign-on the

	system? What are function keys? Who do I call if it doesn't work? How do I sign off the system?
5) System Features	This section will comprise the majority of the user manual. Explanations of the system menus, types of operations available, explanations of how each transaction works, limitations and security in the system, processing flow, numerous relevant examples, where to turn for help, descriptions of all forms/screens/reports used, and what the user responsibilities are.
6) Error Recovery	Even though it is difficult and time consuming, all possible errors should be described in tabular fashion, listing symptoms and cures. Problems indicitive of hardware should be separated from those associated with system problems and application problems.
7) Hardware Maintenance	It is surprising that many people feel computer equipment needs no care. This section might include basic maintenance the user can carry out (cleaning screens and keyboards, adding paper to the printer, changing printer ribbons, etc). As well include a list of items to be referred to the service department and appropriate contacts when things go wrong.

As I stated earlier, the above list is only one possible setup for a user manual, your own needs will dictate your end result. Once the design has been chosen, there are still various hurdles to overcome in this process that will directly affect the creation of your manual, and they are listed below:

1) The orientation of user manuals should be to work functions where the terminal is just a tool, instead of a manual solely about terminal procedures.

2) Some familiarity with the subject matter should be presumed. This allows entire sections to be devoted to specific tasks, such as, "How to perform a Query", "How to perform a Delete", etc.

3) For ease of training, pictures of screens, keyboards, printers, and other equipment should be included near the beginning.

4) References to other manuals are confusing; any situation that is not 'normal' to a user usually results in a request for assistance.

5) Jargon, mnemonics and excessive abbreviations should be avoided.

6) If the same physical screen layout is used to perform more than one procedure it is better to repeat it than refer to another section; this ensures that a single section can cover an entire procedure.

7) Any reference to function keys should be emphasized by bold type or preferably, a drawing of a key top. Rather than, "When a field has been entered - press 'SEND'", it may be more effective to have:

   "When a field has been entered - press $\boxed{\text{SEND}}$

3.3 Application Software Documentation - Perceived Benefits

In the abstract for this paper, I mentioned there are benefits to be gained by maintaining accurate documentation. While one may not be able to assign a dollar value to all of them, the list below covers some of the key benefits.

Cost Savings:
While good documentation will in fact save you money, it is not always obvious how much. As mentioned earlier in the analogy regarding insurance, the cost savings may only be realized once things start to go wrong. In the case of a software firm which produces documentation to accompany its products, the cost savings may be viewed as money not lost through sales. If you had purchased a piece of software and the documentation was so poor it made the program unusable, would you recommend it to someone else? Software with good documentation gets recommended, therefore, if you produce software for the marketplace, it is worthwhile to spend the time and effort to produce quality documentation.

I would like to be able to tell you that every hour you spend in documenting programs/systems would yield you a cost savings of $15.00-20.00 but I cannot. Well designed documentation will help facilitate efficient and effective software development and will decrease training, operation and maintenance costs. In addition, having current documentation of software under development can reduce the risk of duplication of effort by your staff.

Managerial Benefits:
Documentation will increase the flexibility of managers in dealing with turnover or reassignment problems with respect to both end users and system staff. Given the traditional rates of turnover, the benefits should be obvious.

Software Marketing Tool:
Presence of comprehensive, understandable documentation attests to the quality of the related software, and can lead to favorable user beliefs concerning system integrity and reliability. The best conceived, written and implemented system will fail if the accompanying documentation renders it useless. On the other hand, excellent documentation can make a somewhat limited system appear to be far better than it is. Although some would feel this applies only to companies producing software for the marketplace, it impacts on systems developed for internal use as well.

Improved Communication:
Documentation can serve as an important tool for communicating within and between phases of a software project that is split among different groups. Documentation can be used as a quick refresher of both user and DP staff memories, and serve to lessen the potential for conflict and misunderstanding between users and DP staff, as well as between different groups on a software project.

Vehicle for User Participation:
Documentation provides a common baseline for discussion within and between groups. In fact, many DP shops (Motorola included) are currently riding a trend to allow the end users to participate in writing the users manuals for systems they will be using. Participation such as this can stimulate user feedback, morale, commitment and confidence in the software the end user will eventually receive.

4.0 Where do we go from here?

Regardless of your role, be it the programmer of a specific software application, programming mangers, DP director or a technical writer, you must have a good understanding of the five primary ground rule for documentation.

1) In order to solve a problem rather than contribute to it, you must first recognize and acknowledge that it exists.

2) Both technical and user documentation must include sufficient information to be used as both reference and instructional material in order to be considered valid.

3) Writers of computer documentation are instructors. Therefore, they must understand the needs of the people they are going to write for and the level of detail required to satisfy them.

4) Every project must be treated as a training assignment in order to maximize the instructional value of the content and the context of the documentation.

5) The eventual success of documentation depends on the writer's abilities and the company's willingness to provide end users with sufficient detailed and instructional information.

It would probably come as a major surprise to many writers, DP managers and programmers that their documentation often fails to meet the needs of the user. In many cases the documents were never tested or subjected to a formal/informal review process. A study cited by Hastings and King revealed that over 85% of all supportive documentation offends the intellect of the end user while failing to do what it is supposed to -- instruct.

Data processing departments generally sense that something is wrong when systems start to fail after implementation, but rarely do they associate this problem with their documentation. This should not be surprising because these are the same people who have the attitude, "Documentation is just a necessary evil and no one is going to read it anyway!". Take a minute to think about that, for if the people who create the documentation have this attitude, who would want to read the results of their documentation process?

4.1 A little commitment please...

It cannot be stressed enough that the documentation effort must be treated as an integral part of the system development process if it is to support the product/systems. DP departments must have a commitment to turning out quality documentation for every system they produce. Unfortunately, this commitment must be more than a mental attitude; it is knowing the tasks to be performed and who will be performing them when the project is started. True commitment requires taking the time to give everyone involved a complete understanding of what is expected of them and refusing to accept anything less. As overused as the term "management control" seems to be, the push for better documentation must come from the top, it will not happen if left to the programmers. Once a dedicated effort is begun to improve the documentation standards, enforce proper and consistent compliance, improvements will certainly be seen. Documentation is not 'evil', but a necessity that we can no longer afford to ignore.

DOCUMENTATION: THE NECESSARY EVIL (APPENDIX)

Fig. #1

Motorola Information Systems
Program Documentation

```
------------------------------------------------------------

Program: SMCRP075 (Daily Widget Counting)    Eff: 01/01/88
                                            Page: 1 of 3

Input:   INTRANS.FLS.PROD (Verified Transactions)
         METTRAN.FLS.PROD (Metric Conversion File)

Output:  AUDTRAN.FLS.PROD (Audit Transactions)
         REPORT;DEV=LASER,10,4;CCTL (Widget Count Report)

Database Files:

         SYSDB.DBM.PROD (System Database)  Read Add Chg Del
         - SYS-CTL-DTL  (Control File)      X

         MTLDB.DBM.PROD (Materials Base)
         - MTL-IMF-MST  (Item Master File)  X    X
         - MTL-SCF-DTL  (Cost File)         X        X
         - MTL-OBS-DTL  (Obsolete File)     X            X

         PCSDB.DBM.PROD (Production Base)
         - PCS-PIF-MST  (Purchased Items)   X        X

Frequency: Daily

Prerequisite: SMCAN070 (Production Analysis)

Special Forms: N/A

Additional Resources:

         SORTFILE;DISC=450000;DEV=14




 Written by:                       Date:

Approved by:                       Date:

Approved by:                       Date:
```

Fig. #2

Motorola Information Systems
Program Documentation

Program:	SMCRP075 (Daily Widget Counting) Éff: 01/01/88
	Page: 2 of 3
Purpose:	This program will access the validated transaction file and access the database to verify inventory levels in the distributed stockroom. Items that fall outside control levels will be reported.
Input:	INTRANS.FLS.PROD (Validated Transactions) METTRAN.FLS.PROD (Metric Conversion File)
Output:	AUDTRAN.FLS.PROD (Audit Transactions) REPORT;DEV=LASER,10,4;CCTL (Widget Count Report)
Reports:	Daily Widget Count Report - 4 copies
Langauage:	Informix-4GL
Estimate:	2-3 days
Frequency:	Daily

Process Flow:

Access validated transaction file, search item master for matching key, if exists, update quantity counts, check for cost changes, see if item exists as suspected obsolete. If below safety stock levels access purchased item file and issue order message.The printed report will contain the Item-nbr, Qty Used, Qty on hand, Qty on order, Value of stock in-house and on order, and a warning message if the item is suspected obsolete.

Written by:	Date:
Approved by:	Date:
Approved by:	Date:

Fig. #3

Motorola Information Systems
Program Modification Tracking

Program: SMCRP075 (Daily Widget Counting)   Eff: 01/01/88
Page: 3 of 3

System Applied To: Codex Canada ____ MCS ____ INT'L ____

Module: ____________________

Modified Program/Form/Menu/Job: ________________________

Screen (If applicable) : ________________________

Program (If applicable) : ________________________

MSR Reference Number : ________________________

Effective at Release : ________________________

Discontinued as of : ________________________

Due to (KPR,MSR,Release): ________________________

Production Release: Codex Canada ____ MCS ____ INT'L ____

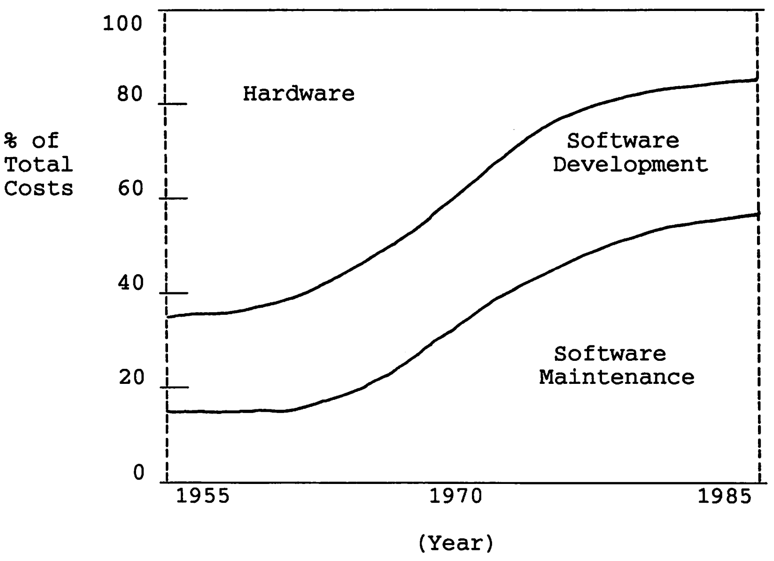
Release	Applied		Responsible	Which Accounts?
	Y/N	Date		

Fig. #4

```
$CONTROL DYNAMIC,BOUNDS
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SMFDR225.
 DATE-WRITTEN. MON. SEP 17, 1987,   2:12 AM.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. HP3000.
 OBJECT-COMPUTER. HP3000.
 SPECIAL-NAMES.
    CONDITION-CODE IS CC.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
$PAGE
**********************************************************
*                INVENTORY CONTROL SUBSYSTEM             *
*                                                        *
 01  PROGRAM-IDENTIFICATION.
     05 PROGRAM-NAME          PIC X(8) VALUE "SMFDR225".
     05 PROGRAM-VUF           PIC X(8) VALUE "A.12.01".
*                                                        *
*    PROGRAM NAME - SUPPLY/DEMAND INQUIRY                *
*    MODULE       - MCS                                  *
*    VIEW FORM    - CUSO                                 *
*    FUNCTIONS    - ADD,CHG,DEL,INQ                      *
*    SUBCOMMANDS  - NONE                                 *
*    SUBPROGRAMS  - SMFMI200, SMFMI210                   *
*                                                        *
*      THIS PROGRAM MAINTAINS INVENTORY COUNT RECORDS    *
*                                                        *
*    FILES           TYPE     GET PUT DEL UPD SRT I O I/O *
*   --------------- -------- --- --- --- --- --- - - --- *
*    MTL-DMF-DTL     IMAGE     X       X   X             *
*    MTL-IMF-MST     IMAGE     X           X             *
*    MTL-OIF-MST     IMAGE     X   X   X   X             *
*    SYS-MSF-DTL     IMAGE         X                     *
*    SYS-TGF-MST     IMAGE     X   X       X             *
*    COUNTERK        KSAM                             X  *
*    AUDITFLE        MPE                           X     *
**********************************************************
```

Fig. #5

Hardware/Software Cost Trends



Source: Barry Boehm, Software Engineering Economics
Prentice-Hall, 1981

References

Boddie, John	Cruch Mode Yourdon Press Englewood Cliffs, New Jersey 1987
Gore, Marvin Stubbe, Jim	Elements of Systems Analysis Wm. C. Brown Company Debudue, Iowa 1975
Hastings, C. Prentice King, Kathryn J.	Creating Effective Documentation for Computer Programs Prentice-Hall Inc. Englewood Cliffs, New Jersey 1986
Hedin, Anne	Unburden the User Data Processing Digest Vol. 31 No. 3 (March 1985) Los Angeles, California
Metzger, Phillip	Managing Programming People Prentice-Hall Inc. Englewood Cliffs, New Jersey 1987
Shelly, Gary B. Cashman, Thomas J.	Business Systems Analysis and Design Aneheim Publishing Company Fullerton, California 1978
Yourdon, Edward	Techniques of Program Structure and Design Prentice-Hall Inc. Englewood Cliffs, New Jersey 1975