NASA CONTRACTOR REPORT 177409

Software Maintenance in Scientific & Engineering Environments: An Introduction & Guide

(NASA-CR-177409) SOFTWARE MAINTENANCE IN SCIENTIFIC AND ENGINEERING ENVIRONMENTS: AN INTRODUCTION AND GUIDE (Sterling Software) 27 p CSCL 09B

N88-12954

Unclas
G3/61 0110673

David Wright

CONTRACT NAS2- 11555
February 1986

NASA

NASA CONTRACTOR REPORT 177409

Software Maintenance in Scientific & Engineering Environments: An Introduction & Guide

David Wright
Sterling Software
1121 San Antonio Avenue
Palo Alto, CA 94303-4380

Prepared for
Ames Research Center
Under Contract NAS2-11555

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

TABLE OF CONTENTS

SOFTWARE MAINTENANCE IN SCIENTIFIC AND ENGINEERING ENVIRONMENTS: AN INTRODUCTION AND GUIDE

1. Summary

The purpose of software maintenance techniques is addressed.

The aims of perfective, adaptive and corrective software maintenance are defined and discussed within this perspective. The consequences for maintenance within the NASA research environment are discussed.

Areas requiring maintenance, and tools available for this are listed, and suggestions for use made. Stress is placed on the organizational aspect of maintenance at both the individual and group level.

Particular emphasis is laid on the use of various forms of documentation as the basis around which to organize.

Finally, suggestions are given for how to proceed in the partial or complete absence of such documentation.

2. Introduction

In this report, maintenance is defined as:

> Any activity involving additions to, alterations mandated by system changes, or correction of, software previously accepted by a user as complying with requirements.

Far from being merely the correction of error, "maintenance" has come to encompass three areas of activity: the perfection, adaptation, or correction of existing code. The meaning of these terms in this context is as follows:

Perfective maintenance is the modification of software to reflect changes in specifications as the user's requirements change.

Adaptive maintenance changes the software to reflect changes in the programming environment, for instance changes to hardware, system software, or some essential application program such as a data-base, graphics or numerical analysis package.

Finally we come to what is closest to the common usage of maintenance: the correction of error to bring performance into line with requirements.

The aim of software maintenance is surely to permit the user continuous, unhindered access to properly working software. A quick scan of a piece of software will often give an experienced eye a fairly accurate idea of how easy it's going to be to maintain. That this is so is encouraging - it indicates that "maintainability" isn't some remote abstraction, but a recognizable characteristic of software.

It is now clear that this "signature" of maintainable software is the outcome of the style and techniques used to create it. Maintainability in software results from the application of a set of techniques involving modular design and structured programming techniques. This is coupled with a set of guidelines on the part of software maintainers and their managers, designed to ensure uniformity of procedure in assessing and making changes to software.

Maintenance techniques should have communication as their main aim. Often though, the literature gives the impression that the aim is the creation of an archive, with no indication of how or by whom it might be used.

This report argues that by ignoring the unapproachability of much documentation by the maintainer new to it, much of its the potential usefulness is diminished. The loss stems from the lack of focus or direction in the documentation. There is no lack of detail; indeed, the detail is usually overwhelming, and hence unusable to a newcomer without a great deal of additional work. For lack of the background usually acquired while creating the original software, problems presented for maintenance will have little initial meaning for the new maintainer. The effort to acquire this background also represents time (cost to the customer!) and frustration to the maintainer, who is eager to get started on problems.

The first recommendation is the creation of a "Programmers' Primer" in parallel with the original development effort, and whose continued currency would be a maintenance task. Such a document would explain the scientific or engineering content of the project at a level roughly that of a Scientific American article. A glossary of terms and abbreviations would be provided, and units of measurement specified and explained. On the organizational side, details would be provided of computers and languages used, and data structures and directories in which software was

resident. Finally, fairly high-level structure charts, at the large subsystem level, would orient the individual as to areas of maintenance. The names of those from whom more detailed information could be obtained should accompany the structure charts.

To the objection that the creation of such a document would take the time of expensive people, the answer is that the total of this time would be far less than the aggregate time wasted by successive generations of maintenance personnel trying to uncover this material themselves. There is also a high degree of probability that some such personnel would fail to completely reinvent the wheel, but would feel compelled to proceed with their maintenance effort anyhow. This leads to error, additional maintenance needs, and so on.

If the lack of an overall view and access to vital background information constitutes one type of barrier to the efficient transition between maintainers, another exists at the level of each module. A way is needed for the most recent maintainer to leave a record of the maintenance status of each group of modules for which there was responsibility. Details relevant to perfective, adaptive and corrective maintenance effort, designed to make accessible some of the day to day level of maintenance experience, would be captured in a brief though clear manner and made available to the next person with responsibility for these modules.

It is recommended that on leaving, each maintainer complete a brief status report on the modules being worked on. This would help continuity of effort.

A more general version of this technique is due to Molari (1981), who has devised a means of producing an Owner's Maintenance Guide for software. The Guide is produced by answering a series of questions drawn from a model checklist, reproduced with permission as Appendix 1 of this document. The questions should be tailored to specific projects; some additional ones will be suggested by the context of each project. The information provided by answers to the questions should enable maintenance programmers to locate the specific portions of the software and documentation required to investigate a particular problem. Each checklist covers a major subsystem of the software to be maintained. The person filling out the checklist should be able to do so in under an hour.

It is strongly recommended that all three of these techniques become accepted practice, since they permit the

maintenance task to be approached in a series of steps of increasing detail, which avoids the confusion of "information overload" if one attempts to jump at once into the middle of a complex situation.

3. Maintenance and the NASA Environment.

Glass and Noiseaux [1981] report that the relative amounts of time spent on the three types of maintenance, industry wide, are roughly:

Perfective:	62%	User's needs change.
Adaptive:	20%	Programming environment changes.
Corrective:	18%	Errors of logic, design or coding.

It's worth considering how the NASA computing environment affects software maintenance.

First, there's a great range of scope of projects, in both time scale and computational demand.

At one extreme is a project involving a summer student requiring access to a VAX and perhaps a specialized piece of output hardware such as an image display. In this case, it's unlikely that any software created will long outlive the project. Essentially such a project is a training exercise which also gives some help to a NASA staff member. Whatever the development difficulties, there will be no additional maintenance considerations.

At the other extreme are long-term, large-scale projects such as the wind tunnels, computational fluid dynamics, flight simulators and space science projects such as the Hubble space telescope and Pioneer. Here the computational demands are great and complex.

The large, long-term project involves two basic maintenance problems: staff turnover and hardware change. The development of adequate documentation forms the basis of an "institutional memory" to help new staff members, and is an important element in addressing the staff change problem. But given the lack of time or facility for formal training of new staff members, very often all they have to go on is whatever commentary there is in the code. All too frequently, specially in the very long-term projects (twenty or more years), the cumulative effect of undocumented or poorly documented code has been to render it close to unmaintainable, though some suggestions as to how to try are in the last section of this report. Trying to discern the

original intent of the code is complicated by the accretion of quick fixes and ingenious ways round the quirks of all the systems the code has ever run on. Personnel, computers, programming standards and techniques have all undergone several generations of change in the interim.

An important difference between the NASA environment and most, say commercial, computing environments is that for many projects at NASA, change is the norm. There is frequently no relevance to the notion of a "finished product" towards which one is working and which, when attained, will remain relatively stable for a long period. Also, hardware change will be more frequent, and more drastic, in a NASA-type situation.

Thus, when assessing the costs and cost-effectiveness of maintenance, one should compare NASA with other research and engineering environments rather than with traditional "data processing" environments.

4. What is there to Maintain?

Each stage of the development process has a product, which needs to be maintained to keep its usefulness. A list of such products is thus a list of items to be maintained:

- Requirements
- Algorithms
- Design
- Programs
- Data
- Documentation for all these

It is recommended that all documentation be stored in text-file form, for ease of access, change and archiving via routine backup. Appropriate file protection will guard against inadvertant loss or overwriting.

It is recommended that all requests for change to any of these be date stamped and signed, and confirmed with the requestor before being acted on.

It is recommended that requests so confirmed are reviewed by a change control group and assessed for feasibility in principle, feasibility within given time constraint and availability of maintenance staff.

It is recommended that requests for perfective, adaptive and corrective maintenance be kept in separate but parallel

directories, with one individual having specific responsibility as librarian.

It is recommended that any request for change of any kind be entered in a Change Log, and that the log also note subsequent actions on the request.

It is recommended that all programs written conform to User or Developer programming standards as to modularity, structure, and documentation.

4.1 Requirements Maintenance

Requirements should as necessary be reviewed with the end-user to ensure that ambiguities and uncertainties are resolved. The original and all subsequent, revised requirements should be kept clearly separate, since it's important to know which requirements changes correspond to what software modifications. Sometimes an older set of requirements needs to be re-implemented, perhaps at another site on an older machine. Or perhaps errors are discovered in some set of requirements, so it becomes important to remove or change the corresponding code. A record should be kept showing which modules address which requirements. This will provide at least a pointer for maintainers.

4.2 Algorithm Maintenance.

An algorithm for a particular purpose will result from the design stage of the software development process. Algorithms may be specially written for a particular project, may be obtained from some source of relevant techniques, or be a specialization of a general technique. In any case, the maintenance concern is that they remain appropriate for present purposes, which may have changed since the original design. Hence the importance of documentation containing:

- Original requirements.
- Original specifications (if these were generated).
- Source of the algorithm.
- Modifications or specializations of off-the-shelf software noted against specific requirements.
- Original design, expressed in a suitable pseudocode.

- Date-stamped notes of change requests, indicating requestor and reason for change.

- Copies of responses to such requests, including pointers to corresponding changes in the implementation of the algorithm.

4.3 Design Maintenance.

Probably the most effective way of recording design is some form of high-level pseudocode. There are many pseudocodes available, most of which look on the page not unlike the computer language on which they will be implemented. Clear indication should be given of which modules in the actual programming language the designs correspond to.

4.4 Program Maintenance.

Our aims here are to ensure:

- Semantic correctness: does the code embody current requirements? There are no logical proofs of whether it does; hence the importance of defining adequate test data. Test data should exist to permit regression testing as needed.

- Compliance with software standards as to:
  Modularity.
  Structure.
  Documentation: naming conventions for programs, modules, identifiers; in-line commentary; directory and file structure, and backup procedures.

It needs to be clearly borne in mind that the only realistic aim of software maintenance is to minimize error. It can no more eliminate it altogether than medicine can eliminate fatality from disease. Only the odds can be changed.

In order for program maintenance to proceed efficiently, the following documents should be available:

- Requirements.

- Current specifications.

- Notes (formal Technical Memos as well as "lab notebook" type notes) of the original designer(s) and developers.

- Indication as to software source, if obtained "off the shelf".

- Location and justification for modifications to such software.

- Relevant language and system manuals for the computer on which the software is now run, and those from which it has been ported, if any.

- Cross-reference listing of the most recent version, which if adequately maintained will contain a modification history.

- Test data used to verify past versions.

- A log of all past problems, tied to the specific modules found at fault.

- An account of how these problems were resolved.

- A statement of directory structure, access rules and configuration control protocols for change.

- A set of the user's software standards or (if none) those of the maintaining organization.

The approach to program maintenance is parallel in structure to that of algorithm maintenance, and is addressed in the same way - keeping clearly separate the three strands of perfection, adaptation and correction. With programs, there are the additional strands of explicit system dependence, hardware and software, as maintenance considerations.

4.5 Data.

The only reason programs are written is to manipulate data. This point is so commonplace as to be embarrassing to make, but experience shows that the design of data structures receives scant attention during the development phase, considering its importance to run-time considerations such as time loss due to unnecessary i/o interrupts. For each of these areas of concern there is associated documentation to maintain.

- Data structures: File and record structures;

- Scope of definition of variables in languages permitting this;

- Data structure declarations: the location, type and size of identifiers set up by PARAMETER, COMMON, and EQUIVALENCE statements and their definition (initialization) via DATA statements.

- Data flow diagrams or their equivalent if available.

- Data bases. This term is used too widely to have anything like a precise meaning. It could equally refer to a ten year accumulation of experimental data tapes, or to a specialized program for maintaining large amounts of data on disks. But whatever your project's usage, the need for maintenance is there!

4.6 Documentation

- Original requirements, specification and design documents.

- Problem reports/complaints from the user.

- In-line commentary in code.

  Each module should contain a standard header showing module name, aim, explicit system dependencies, language dialect, identifier declarations, and data dictionary briefly defining the meaning and usage (input, output or scratch) of each identifier. There should also be a cumulative record of modifications, giving brief reasons and name of author.

- Program Runners' Guide.

- Tutorial material.

As has be seen from the repeated references to the use of documentation in the maintenance process, properly designed, written and maintained documentation is THE key to success in the maintenance effort! It cannot be too strongly emphasized however that unless the documentation itself is kept current with changes to the software, maintenance will become an extremely costly and unsatisfactory activity.

Indeed, there have been cases where reasonable but false assumptions made for lack of relevant documentation have resulted in the introduction of error where none was before. Also, errors have been introduced by the use of plausible but wrong documentation.

Any change to software, however well planned, has the potential for introducing error. A basic reason is that even the most skilled user of a keyboard has a nonzero error rate! A file being edited must be changeable - for worse as well as better.

5. The Application & Host System Hardware and Software Dependencies.

It is well known that the system (computer/operating system) under which most applications run both make possible and limit what the applications are capable of. The list below indicates some of the areas of dependency:

- Hardware dependencies, e.g.:
  computer word length;
  instruction set;
  speed of execution;
  hardware versus software
  implementation of floating
  point arithmetic;

- i/o devices available.

Hardware dependencies show up as maintenance problems when we attempt to port software between computers. At once, certain features of the source and target computers' hardware design become important. For instance, real-time applications are sensitive to instruction execution speed. Another such feature is word length, which affects such issues as the range and accuracy of numerical representation, and also the internal representation of various kinds of data. A related issue is how the bits within the word, however many there are, are grouped. For instance, the CDC 7600 employed 60 bits to a word; the DEC VAX series uses 32. The CDC sub-divided a word into ten six-bit character units; the VAX, into four eight bit character units.

Another dependency is the order in memory in which bytes are accessed within a word, even if the number of bits per word is the same. If you're converting from a system that accesses left to right to one which goes in the opposite direction, then a utility is going be needed which reflects the order in which bits read from one machine are interpreted on the other.

- Operating system dependencies, e.g.:
  System utility calls of any sort;

Use of command files to pass parameters to, or otherwise control the running of programs.

- Compiler dependencies, e.g.:
  non-ANSI standard language features provided by a particular vendor, such as identifiers of more than 6 characters, and
  LOGICAL*1 data type in VAX FORTRAN.

A decision will be required on whether to exploit these, or to remain strictly within ANSI FORTRAN. Strict conformity may yield ready portability, but at the cost of the advantages afforded by many of the non-standard features. How often is porting envisaged? How helpful are some of the non-standard features? Again, costs and benefits can be weighed.

To some extent this kind of maintenance can be anticipated by designing software to be "portable", that is, containing minimal explicit dependence on a particular computer, operating system release or compiler. Any apparently unavoidable remaining dependencies should be noted clearly in the in-line commentary and in the external documentation.

- File handling dependencies.

- Maintenance tools provided by the computer maker or outside vendors.

6. What Constitutes "failure"?

After acceptance by the user, any report of unusual or unacceptable output constitutes a failure - though it may be of communication (say the result of a poorly written operator's manual) rather than a flaw in design or code.

7. Some Reasons for Failure of "working" software

- Unanticipated change in input data.

- Hardware failure.

- Upgrade or other hardware change in existing system.

- Change in existing system software, e.g. new operating system release.

- Any allegedly "transparent" change to either hardware or software.

- Failure of direct portability to system other than that on which developed.

- "Hidden" dependencies, e.g. changes to any other system which provides input data to a program. If, as is often the case in NASA, the input is provided by another organization, then it is important to be able to contact them. Additionally, if possible, knowledge of relevant aspects of their code, such as data editing ranges, may be needed.

- Code rendered brittle or fragile from too many earlier "quick fix" undocumented patches.

8. Who should perform Maintenance?

The maintenance function is a crucial one to the effective use of some very expensive equipment serving highly sensitive areas, such as air traffic control, automated chemical plant or defense communication network. Whatever shape the software was accepted in, it is up to the maintainers to make it work as it should. This is not a task for beginners, since it involves being able to follow not only what should happen, but in what ways it has failed to do so. Technically it is a difficult area to work in; as one is always dealing with problems, it can be very stressful as well. As it will look back in terms of systems and languages in use, rather than being on the leading edge of development, strong inducements may be needed to attract the necessary talent. This will tend to make it more expensive than development work, but is unavoidable.

Maintenance must be made an attractive area in which to work, attracting the good people essential to do it justice, or it will become an area of nagging drudgery shunned and dreaded by all, and hence staffed essentially by junior-level conscripts. This will result in increased expense, if only because more people will work longer to attempt the same ends, with a much lower likelihood of adequate work.

9. Maintenance: Quality Assurance

Besides the importance of accurate, complete documentation, there are two keys to quality assurance in the maintenance area. The first is to minimize the amount of maintenance required. There is always the chance that some small change of apparently local scope will have unforeseen side effects. Error will be introduced elsewhere where formerly there was none. This is possible in particular if in starting a maintenance task, the documentation has been misinterpreted. One way to detect if such error has been introduced is to perform regression testing whenever change is made.

The formation of a maintenance quality circle, involving users as well as software personnel, may also be found helpful. This has the effect of encouraging the users and producers of software to gain insight into each other's problems and difficulties. It also encourages the exchange of ideas and information.

The second key is the personnel doing the maintenance. This was discussed above.

10. Maintenance: Techniques and Tools.

Many of the tools useful in maintenance are those used in development, since in many ways maintenance is the continuation of development by another name. The emphasis is on the carefully controlled introduction of any change, and care at all times to avoid inadvertant change to files.

Design reviews and walkthroughs are familiar development tools, which work just as well in a maintenance environment.

The same is true of code inspections.

Protection against inadvertant change to files is provided by configuration control. This is the technique by which file creation, access and change is managed. There are commercially available packages designed to establish configuration control, but a particular project may find it more desirable to develop its own. However, the effectiveness of configuration control depends crucially on members of the development (and later maintenance) personnel routinely following whatever guidelines are established in this area.

It is recommended that all these techniques be employed during the maintenance as well as the development phase of the software life cycle.

A difficult question of maintenance is to decide when to rewrite from scratch rather that fix what exists. Empirically, it seems that if 50% to 75% of the modules in a system are in need of maintenance, then a complete restructuring is in order. For an individual module, if 15% or more is in need of change, rewriting is in order.

11. Missing Documentation

Given the considerable emphasis on documentation as the basis for maintenance, what does one do in its near absence? The first suggestion that comes to mind is simply to rewrite, but this assumes the original requirements are accessible. In their absence, all that can be done is a painstaking reconstruction of the current state of the system from available listings and directories of files.

The first thing to do is establish if the listings correspond to current source on object code. This can be done by examining the dates of file creation in many instances. Hopefully source code, or at least listings, will be found for all object modules. It is then necessary to take whatever commentary exists in the code, and attempt to understand how and what it's doing. The help of earlier developers or maintainers should be enlsited if they are available. Work areas should be scoured for old notebooks or anything which may be of use. Users and former users may be of help in this reconstructive effort.

In summary, find out what there is, and if possible what it does. The situation will be serious but may not be desperate. The cost of doing this may be considerable, but apart from abandoning the project for which the software was developed, there's not much else to be done.

12. Maintenance: "Defensive Development".

If the occurrence of the word "development" seems strange in the context of a document on maintenance, consider that in a very real sense development never stops. In a formal or legal sense, of course, development could be said to stop at the point of first acceptance by the customer of the software as working. But software truly isn't like other products in this regard, partly because it's much more flexible by nature. If a company buys a digital voltmeter, the manufacturer isn't going to be asked to turn it into a compact disk player! Yet the cumulative changes to originally quite modest software can completely change the

product. It is in this light that the term defensive development is used. Knowing that change will be required, how can the demands that it will put on the existing software be met? Some of the adverse effects of unpredictable change will be mitigated if you anticipate that:

- People will depart projects at short notice. Develop a checklist to be turned in by the department. The checklist should include:

  (1) A list of modules for which the individual was responsible, and indication where these lie in the hierarchy of module dependency.

  (2) Any working notes on current modifications as yet incomplete, including the requestor's new requirements.

  (3) Cross-reference listings of current modules, and indication as to what module(s) call them.

  (4) Library listings showing current location of source, object and linked code.

  (5) Logs indicating completion of documentation of previously modified modules.

Ensure that the checklist is indeed completed. Exchange it for the final paycheck and handshake, if need be!

- Computers will come and go - sometimes with amazingly little notice to your particular project, which may not loom large in the vision of those responsible for choosing the next machine. Will you, for instance, be stranded with fifteen years of data tapes recorded at a density unreadable by the new machine's drives? An explicit list of machine dependencies would help avert little surprises like this. Your group's concerns with a proposed choice of machine or peripheral grouping are more likely to be attended to if accompanied by a list of concrete reasons for alternatives.

- Of course, with new machines come new operating systems, linkers and compilers. Again, an awareness of dependencies allows one to plan ahead. Helpful manufacturers will continue to upgrade operating systems. What vulnerabilities has your software to new releases? They lie in the direction of system calls, command file techniques and file management systems.

APPENDIX I

MODEL CHECKLIST FOR PROGRAM MAINTENANCE GUIDE

In responding to the checklist, please:

- Brief.
- Be accurate.
- Be specific.
- Define locations in the software by module name, if applicable.
- Please specify any other documents where this information may be found. If possible, specify the section.

1. MAINTENANCE DUE TO SYSTEM SOFTWARE MAINTENANCE

1.1 If this software has to be recompiled/assembled, how should it be done (e.g. commands or catalogued procedures to be used)?

1.2 If this software has to be re-linked, how should it be done?

1.3 If this software has to be moved to another similar machine, how should it be done?

1.4 What parts of the operating system affect this software aside from the compiler/assembler and linker (e.g. data base/data communications monitors, device drivers)?

1.5 Are any software "packages" used (e.g., sort routines)? Are they specific to a certain system release?

1.6 Are there any backups to the software or data files? Should any be done regularly?

2. MAINTENANCE DUE TO SYSTEM HARDWARE MAINTENANCE

2.1 What peripherals are required by this software?

- Terminal
- Printer
- Digital tape
- Other

2.2 For each peripheral, please indicate:

- Special features the software relies on, or special hardware models or types it relies on.
- Whether a different model could be substituted without affecting the software.
- What software modules perform I/O to the peripheral (or "many").
- What the software does if the peripheral is not ready or is missing (e.g., messages, abort, wait, etc.).
- What is the minimal hardware configuration on which this program will run (memory, peripherals, disk storage space, etc.)?

3. MAINTENANCE DUE TO OPERATIONAL PARAMETERS (REAL TIME ENVIRONMENT)

3.1 How are logical unit numbers assigned for the peripherals? How would they be changed?

3.2 Are any peripherals logically "attached" for sole use by this program? By which module? How could this be changed?

3.3 Are any disk files used by this software? Where are the names stored in the program? Where are the formats described (specify document, if applicable)? Which software modules do the I/O?

3.4 What would you do if the program were modified and became too large?

3.5 Are there any speed requirements that are met by the program with a small margin of safety? What would you do if the program were modified and no longer met these requirements?

3.6 Are there any internal or external parameters that can be changed to "fine-tune" the operation of the program? (Names, locations)

3.7 What is the fastest or most efficient way to run the program (e.g., how to set parameters, how to input data)?

4. MAINTENANCE DUE TO SOFTWARE STRUCTURE

4.1 Where is most internal data initialized?

4.2 Are data, variables, or arrays used for more than one purpose? Where?

4.3 Where does the program begin and exit? (Module names)

4.4 What are the largest arrays? Where are most internal tables? (Names, sizes, modules names)

4.5 What language features are relied upon that are ANSII-standard or that do not follow local standards? (See Appendix for list of non-ANSII features.)

4.6 Are there any diagnostic or debugging features built into the software? If documented elsewhere, please specify the document.

4.7 Where are input parameters checked for range and consistency? How could these restrictions be changed? How could a new parameter be added?

4.8 Where is most of the time spent in the program? (Module name, section)

5. MAINTENANCE DUE TO EXTENSIONS

5.1 What extensions or modifications would you suggest or consider for this software?

5.2 What extensions or modifications have others suggested to you for this software?

5.3 For each change listed above, please classify it as:

quick (1/2 week),

medium (1/2 month), or

long (more than 1 month)

Also classify it as:

easy (can be done by anyone who knows the language),

moderately hard (would take some study of most of the software internally), or

complex (need to understand details of linkage to operating system, drivers or hardware)

6. OTHER MAINTENANCE

6.1 What other kinds of maintenance do you think may be required the most? Please include periodic maintenance, extensions, corrections as a result of experience, etc.

REFERENCES

Glass, R. and Noiseaux, R.: Software Maintenance Guidebook, Prentice - Hall 1981

Molari, R.: Producing an Owner's Maintenance Guide, Informatics Inc. 1981

Federal Information Processing Standards Publication (FIPS) Publication 106: Guideline on Software Maintenance, U.S. Department of Commerce 1984

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
CR177409		

4. Title and Subtitle	5. Report Date
Software Maintenance in Scientific & Engineering Environments: An Introduction & Guide	February 1986
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
David Wright	
9. Performing Organization Name and Address	10. Work Unit No.
Sterling Software 1121 San Antonio Avenue Palo Alto, CA 94303-4380	K1707
	11. Contract or Grant No.
	NAS2-11555
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics & Space Administration Washington, DC 20546	Contractor Report
	14. Sponsoring Agency Code

15 Supplementary Notes

Point of Contact: Robert A. Carlson, MS: 233-15, Ames Research Center, Moffett Field, CA 94035
(415) 694-6036 or FTS 448-6036

16 Abstract

The purpose of software maintenance techniques is addressed.

The aims of perfective, adaptive and corrective software maintenance are defined and discussed within this perspective. The consequences for maintenance within the NASA research environment are discussed.

Tools available to the maintainer are listed, and suggestions for use made. Stress is placed on the organizational aspect of maintenance at both the individual and group level.

Particular emphasis is laid on the use of various forms of documentation as the basis around which to organize.

Finally, suggestions are given for how to proceed in the partial or complete absence of such documentation.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
software maintenance, quality assurance	Unclassified; Unlimited Subject category 61

19 Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
UNCLASSIFIED	UNCLASSIFIED	23	

*For sale by the National Technical Information Service, Springfield, Virginia 22161