Technique	Apply maintainability concepts to development of software in the early phases of the lifecycle.

Software Design for Maintainability

Establish Design Criteria for Maintainable Software Products to Increase System Availability and Decrease Overall Development and Operating Costs

Benefits	By designing software products that are maintainable we can update and enhance fielded software much faster and at lower cost. Software can be reused, thus alleviating costly update time. Also, any faults found in the software can be easily diagnosed and corrected, reducing downtime and meeting delivery schedules. Software maintainability ensures system availability by reducing system downtime.
Key Words	Maintainable Software, Modular Design, Object Oriented Design, Coding Standards, Naming Convention
Application Experience	International Space Station Program
Technical Rationale	Because of increases in the size and complexity of software products, software maintenance tasks have become increasingly more difficult. Software maintenance should not be a design afterthought; it should be possible for software maintainers to enhance the product without tearing down and rebuilding the majority of code.
Contact Center	**Johnson Space Center (JSC)**

Software Design for Maintainability

Technique DFE-6

Introduction

Recently, software products have exhibited a dramatic growth in size, complexity, and life cycle cost (LCC). In a large system, software LCC typically exceeds that of hardware, with 80-90 percent of the total system cost going into software maintenance to modify the delivered program to meet the changing and growing needs of users. The total system cost includes hardware, software, acquisition, development, and deployment costs.

Changes over the life of a program are inevitable, even if the program has met all its design requirements. Software changes are needed to adapt to increased functional requirements and different system configurations brought about by these changes.

One of the most urgent concerns in the computer industry is the need to maintain and enhance the software product at faster rates and at lower costs. To meet this objective, better, more maintainable software must be designed.

Maintainable Software

Maintainable software, which facilitates the correction of errors and deficiencies, can be expanded or contracted to satisfy new or changing requirements, which may include enhancing existing functions, modifying for hardware upgrades, and correcting code errors. The means of achieving this goal depends on not only one technique, but a combination of many tools and techniques, including the following:

a. Early planning: anticipating what and how programs might be modified at a later stage.

b. Modular design: defining subsets and simplifying functionality (i.e., one module performs only one function).

c. Object-oriented design: encapsulating both methods and data structures to achieve a higher level of independence than that of modular design.

d. Uniform conventions: facilitating error detection and debugging.

e. Naming conventions: providing understandable codes.

f. Coding standards, comments, and style: enhancing readability of the program.

g. Documentation standards.

h. Common tool sets.

i. Configuration Management

It should be noted that topics d through h can all come under the classification of configuration management, which as a subject in and of itself is very important to maintainability, but is listed by itself to show importance.

Early Planning

As with anything else, early planning puts design problems into perspective, provides good strategy, and is the most cost-effective way for modifying or adding features to software products at some later time. Early in the definition phase, expected changes should be identified and prioritized so that their considerations can result in an architecture receptive to change.

The system functionality should be decomposed into manageable segments for which software modules may be built. The format of these modules should be standardized so that code can be added, deleted, or modified to incorporate expected as well as unexpected changes. This helps to ensure that minimum interface alterations will be required to implement a change. Also, some particularly volatile areas should have their parametric values stored in databases to facilitate their change. Identifying expected changes early in the definition phase, and making allowances for unexpected changes, makes for a more maintainable software product.

Configuration Management

Configuration management of software is probably the single most important management and maintainability concept utilized in software development Utilization of coding standards, documentation standards, release standards, common languages and other methods will provide for good configuration management. A plan should be developed very early in the development cycle for managing the configuration of the software under development, and that plan should be followed rigorously. If configuration management breaks down, the code under development is doomed to be extremely troublesome when release for operations.

Modular Design

Modularized software is best structured so that high probability changes do not affect the interface of widely used modules. However, one of the most commonly encountered errors is when two or more simple functions are combined into one module because the functions seem too simple to separate.

For example, one might be tempted to combine synchronization and message sending and acknowledgment in building an operating system. The two functions seem closely related and for the sake of reliability one may insist on a "handshake" with each exchange of synchronization signals. If later an application is encountered in which synchronization is needed more frequently, it may be found that there is no simple way to strip the message sending routine away from the synchronization routines.

The irony of such a situation is that the mechanism could have been built effectively and separately from a simpler mechanism.

Modular design also provides reusable code and reduces redundant coding that might occur in many similar functions. For example, a generic input/output routine will save coding time and make it convenient for users to retrieve and use the module. Moreover, the modules can be saved in a software repository; e.g., FORTRAN Mat-Lib, and made available to the public.

Identifying potentially desirable modules, however, is a demanding intellectual exercise in which the software designer first searches for the minimal module that might conceivably perform a useful service and then searches for a set of minimal increments to the systems. Each increment is small-sometimes so small that it seems trivial.

The minimalist approach seeks to avoid a module that performs more than one function as discussed above. Identifying minimal modules is difficult, however, because minimal system programs are not usually requested. Minimal modules are useful if the software family is going to be developed but are not usually worth building by themselves. Similarly, maximum flexibility is obtained by looking for the smallest possible increments in capability.

Use of ADA Programming Language

The Department of Defense (DoD) undertook an effort some years back to develop a standard language that would provide for many things, including software maintainability. ADA programming language, the result of this effort, will provide software maintainability, as a rule, if utilized properly with other concepts listed here. Other languages can be maintainable as well, but much research and planning must go into the language chosen for development, and ADA is always a good place to start, especially in the development of government applications.

Object-Oriented Design

Object-Oriented Design is a more recent approach in restructuring programs. It is intended to make program tasks more independent of each other and therefore easier to maintain.

Normally, as taught in most computer courses, structured programming helps in dealing with the complexities and reduction of "spaghetti" in the code; however, structured programming is still based on the expected sequence of executing instructions. Attempting to design and debug programs by thinking through the order in which the computer does things ultimately leads to software that nobody can fully understand. With object-oriented techniques, a designer generates code based on objects and their behaviors. Objects, which might be real or abstract, could include an invoice, organization, order-filling process, or icon on a screen that a user points to and opens. An object's behaviors are expressed by its contained data structures and operations, which are also called "methods." Most systems can be built without having to think about loops, branches, and program-control structures, because the object-oriented technique is an event-driven programming approach in which events cause changes in the state of objects.

Each state change is usually simple to program by itself, so the program is divided into relatively simple pieces. Each object, in effect, performs a specific function independent of other objects. It responds to messages, not knowing why the message was sent or what the consequences of its actions will be.

Because objects act individually, each class which is made up of these objects can be changed largely independent of other classes. This makes the class relatively easy to test and modify.

Object-oriented systems are much easier to maintain than conventional systems. Space and other considerations make it impossible to discuss the object-oriented concept here. Readers to whom this concept is new are encouraged to pursue it further in object-oriented programming.

Uniform Conventions

Software coding standards and naming conventions are also important in producing maintainable code; therefore, they must be established during the development process. All production of new code should be done according to these standards, along with all program extension and repair work, during both the development and maintenance phases. All software development processes should be documented, consistent and repeatable.

The benefit to the maintenance programmer is that by learning the formal aspects and naming conventions of one segment of the system software, he or she will know these aspects and conventions for the other segments. Error detection and debugging are facilitated with more effort concentrated on understanding the logic of the program, even when working with a new segment.

Naming Conventions

The naming standards should encompass the systematic assignment of mnemonic terms chosen to suggest their own interpretation by carrying as much information about their respective variables as possible. It would be desirable, from a maintenance programmer's point of view, if there were a one-to-one correspondence between variable names throughout the program system. Global variables should be defined in a common glossary with their names the same in all routines. Local model variables are those having a meaning in the model or system specification but appear only in one routine; names of these may duplicate other local variable names, but not global names.

Coding Standards, Comments and Style

Since easy-to-read code is a definite plus to the software maintainer, one set of coding standards should be used to develop the documentation, flowchart construction, input/output processing, error processing, module interfacing, and naming of modules and variables. This consistency, which promotes general understanding, has prompted the Government to stipulate in software programming contracts the inclusion of the following software coding:

a. Presentation style: describes the rules and conventions for the format of the source code that may include paper listings, listings stored on electronic media, or both:

(1) Indentation and spacing.

(2) Use of capitalization.

(3) Uniform presentation of information throughout the source code; the grouping of all data declarations.

(4) Use of headers.

(5) Layout of source code listings.

(6) Conditions under which comments are provided and format to be used.

(7) Size of code aggregates (100-200 executable, nonexpandable statements).

b. Naming: defines rules and conventions governing the selection of identifiers used in the source code listing.

Documentation Standards

The reasoning for documentation standards is nearly identical to that for coding standards. Standardized document formats are necessary for good maintainability. Consistently formatted documentation simplifies the process of familiarization with any given project and assists the effort to make any given information easier to locate. It also promotes understanding of what the current status of a project is, what changes have been made, and the reasoning behind various activities during the development process.

Common Toolsets

A uniform environment and a standardized set of development tools adds several points important to increasing the maintainability of software. First, is the ability to reuse validated code. If changes must be made to the software, and the changes are similar to something that has been used previously, it is very convenient to be able to use code that has already been proven with the development tools being utilized. Secondly, a common toolset and environment allow for program and code portability between stations. As the environment is common, no conversions are necessary, and program compilation differences are negated. A standardized environment also reduces time necessary for training. Much less time is necessary to train people on one system than to train them for multiple toolsets. This also has the advantage of raising the average level of knowledge for that one toolset, thereby increasing efficiency. When one toolset is used for development, only one set of software resource libraries is necessary. All of this combines to decrease necessary resources needed to develop and maintain any given piece of software.

Conclusion

Software maintenance is a major cost contributor to software LCC. Maintenance is incurred to both correct faults and enhance capability. Therefore, designing code that minimizes maintenance costs will effectively reduce the LCC of operational software.

References

1. Capers Jones, "Programming Productivity: Issues for The Eighties," ITT Programming Technology Center, Stratford, Connecticut.

2. Tsun S. Chow, "Software Quality Assurance: A Practical Approach," IEEE Computer Society, Worldway Postal Center, Los Angeles, California.

3. J. Martin and J. J. Odell, "Object-Oriented Analysis and Design," Prentice Hall, Englewood Cliffs, New Jersey.

4. Parish Girish, "Handbook of Software Maintenance," John Wiley & Sons, New York.

5. "Third International Conference on Software Engineering," May 10-12, 1978, Atlanta, Georgia.

6. "Annual Reliability and Maintainability Symposium," January 26-28, 1993, Atlanta, Georgia.