Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

LA-UR--85-1708

DE85 012732

TITLE: A DEVELOPMENT METHODOLOGY FOR SCIENTIFIC SOFTWARE

AUTHOR(S): G. Cort, J. A. Goldstone, R. O. Nelson, R. V. Poore,
L. Miller, and D. M. Barrus

SUBMITTED TO Fourth Biennial Conference on Real-Time Computer Applications
in Nuclear and Particle Physics
Chicago, Illinois
May 20-24, 1985

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy.

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

A DEVELOPMENT METHODOLOGY FOR SCIENTIFIC SOFTWARE

G. Cort, J. A. Goldstone, R. O. Nelson, R. V. Poore,
L. Miller and D. M. Barrus

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Abstract

We present the details of a software development methodology that addresses all phases of the software life cycle, yet is well suited for application by small projects with limited resources. The methodology has been developed at the Los Alamos Weapons Neutron Research (WNR) Facility and was utilized during the recent development of the WNR Data Acquisition Command Language. The methodology emphasizes the development and maintenance of comprehensive documentation for all software components. The impact of the methodology upon software quality and programmer productivity is assessed.

Introduction

With the rapid advance of hardware technology, and the increasing complexity of software applications, modern software systems are becoming much larger and significantly more difficult to manage. For numerous reasons, these factors have had a particularly serious impact on the quality of scientific software. Whereas large engineering organizations can devote considerable resources to software management, most scientific software development projects are characterized by small staffs with very limited resources. That most scientific programmers have little or no familiarity or experience with software management issues exacerbates the problem. As a result, software developed in this environment is often fragile, haphazardly designed, difficult to use and impossible to maintain. Documentation, if it exists at all, is frequently inconsistent and inaccurate.

In order to address these problems, and with the ultimate goal of improving software reliability and maintainability, the Computer Group at the Los Alamos Weapons Neutron Research Facility (WNR) has established a comprehensive development methodology for scientific software. This methodology employs many of the strategies and techniques utilized by large projects to manage the software development process, but significantly reduces the associated overheads.

Assumptions and Implementation Strategy

The software systems that are developed and maintained by the WNR staff can be characterized as mission-critical. (This is particularly true of the real-time systems.) The operational lifetime of most of these systems is expected to be rather long--approximately ten years. The WNR methodology must, therefore, promote the development of highly reliable software that can be maintained by a small staff. We base this strategy upon three assumptions.

1. Large-scale methodologies are not suitable for use by projects of small or intermediate size. Very effective methodologies already exist for managing software development projects. Historically, these strategies have been pioneered by, and perfected for, large-scale software development projects which can devote considerable resources to applying them. These methodologies usually require a dedicated staff to perform time intensive activities within a highly stratified management structure. A commitment such as this usually represents only a small fraction of the total resources available to a large project. The WNR staff, however, consists of approximately four full-time programmers and a single manager. As would most small or intermediate size projects, the WNR effort would be overwhelmed by the institution of a large scale development methodology.

2. Coding is the least important activity associated with any development or maintenance operation. Coding should correspond to merely the translation of a sound design into an implementation. The most critical development activities are performed either before (specification and design) or after (testing) the coding phase. Most small projects (especially those of a scientific nature) emphasize the coding activity--often to the complete exclusion of the other phases. Reliability, consistency and maintainability are very difficult to introduce into an application during the coding phase, so many scientific applications are characterized by very low quality.

3. Documentation is fundamentally more important than code. Every software professional has been exposed to inadequately documented code, usually during a maintenance operation. The precedence of documentation over code, however, has far wider implications: over the entire software life cycle, complete documentation is crucial to the successful operation of any software system or component. In addition to adequate maintenance documentation, requirements must be fully documented (for comparison with validation test results). A carefully documented design is invaluable when the software requires enhancement or repair. Exhaustive documentation of test coverage, specific test cases and all test results is essential for validation and verification of subsequent versions. For these reasons, the WNR methodology emphasizes the development of complete, consistent and uniform documentation for every phase of the life cycle.

The WNR methodology implements these assumptions by applying policies of universal standardization and strict modularity. Standardization of all development/maintenance activities (and the results thereof) is the simplest means for promoting uniformity and completeness in the final product. This can be accomplished without adding appreciably to

the management overhead. Standards application and verification can often be accomplished with automated tools, further reducing the overhead.

In order for standards to be effective, however, a certain level of management commitment is required. Of primary importance is enforcement of existing standards. Standards must be compulsory and universally applied. Voluntary standards or selective application doom the effort to failure at its inception. In the very rare instances in which exceptions are made and standards violations are permitted, there must be significant other advantages to be gained (improved functionality, clarity of expression, etc.) other than mere convenience.

Standards, to be effective, must exist in written form. Unrecorded conventions that are presumed to be understood by all participants are often worse than no standards at all. Interpretations vary with individuals, and standards application subtly changes with time. Hence, the content and uniformity of resulting software varies radically with author and date of development.

The WNR facility standards address all aspects of the software development project. They establish development procedures, specify the components to be produced during each phase and define acceptance criteria. At a lower level, they promote complete and uniform documentation by defining a partitioned documentation template and rigidly specify the information required in each partition. Coding and style standards are provided to specify the implementation language, to identify illegal control structures, to establish formatting rules (indentation levels and case rules), to set naming conventions and to define modularity constraints.

The policy of strict modularity is defined in the facility standards and specifies limitations for the size and contents of a software module, as well as the allowed forms of communication between modules. The goal of this policy is to compartmentalize the software, thereby reducing side effects which are attributable to pathological inter-module connections.

The WNR methodology limits the number of software routines (programs or subprograms) per source file (module) to one. Module length is not to exceed 100 executable lines. Communication between modules must be performed exclusively through calling parameters. Global common data structures are explicitly forbidden. These rules promote the development of small, highly functional modules with simple interfaces. Reuse of existing modules is therefore encouraged and maintenance can be performed on individual modules without jeopardizing other unrelated components.

The Life Cycle Approach

The WNR methodology is based upon a software life cycle model. The model partitions a development or maintenance operation into three

phases: specification, design and implementation. The implementation phase is further subdivided into coding and testing activities.

The activities that may be performed during a particular phase of the life cycle are rigidly specified in the facility standards. During the specification phase, functional requirements are analyzed, general algorithms are established, and top level constraints are identified. The design phase addresses the detailed algorithmic and procedural aspects of the software product. The coding activity translates the design information into the appropriate programming language. The testing activity is required to generate a formal test plan which specifies the test coverage and which describes each test case in detail (purpose, inputs and results required to pass the test). The testing activity must also generate a test report that contains the actual results of every test case.

Work on a particular development/maintenance cycle proceeds sequentially through each of the phases described above. Generally, all work specified for a particular phase or activity must be completed prior to beginning the subsequent phase. (The development of the test plan may, however, proceed concurrently with the coding activity.)

Peer reviews are employed to verify the completeness, correctness and appropriateness of all work performed during a particular phase of activity. Work is subjected to a mandatory review at the conclusion of each phase. For extremely complex projects, intermediate reviews may also be required (e.g. a test plan review).

The reviewing body is composed of the entire programming staff, the section leader, and a representative of the user community. Collectively identified as the Configuration Control Board (CCB), these individuals determine the completeness of submittals by comparison with standard baselines. A baseline is defined in the facility standards for every phase and activity. Each baseline details specific components that must be completed to satisfy the requirements of the corresponding phase or activity.

Submittals are also reviewed for compliance with general facility standards (coding and documentation). Algorithms are critically evaluated and required changes are identified. The reviews therefore provide a means for identifying and eliminating errors at the earliest opportunity. This guarantees that each phase of the life cycle is addressed in the appropriate order and in a complete and consistent manner.

Documentation

Documentation is the cornerstone of any successful development methodology. The WNR strategy emphasizes the development of exhaustive documentation for each system component during each phase of the software life cycle. The software review process promotes the generation of complete, uniform, accurate and current documentation. Of equal importance to the small project is the requirement that the documentation be easily managed.

One of the problems associated with large-scale methodologies is that although they produce very comprehensive documentation for each life cycle phase, this documentation exists as many separate components (requirements documents, statements of work, design documents, user's guides, reference guides, etc.) and in many different forms (text, graphics or binary). The sheer number of components makes cataloging and tracking the various versions extremely difficult. Some entities (e.g. graphical data flow diagrams and structure charts) are tedious and time consuming to produce and very difficult to maintain. As a result many components become obsolete as the project evolves, thereby undermining confidence in all existing documentation. For small projects with limited resources, the effort required to support a documentation strategy such as this is prohibitive.

The WNR methodology is designed to drastically reduce both the number of distinct documentation components and their manifestations, without severely restricting the documentation scope. Only one manifestation is permitted: all documentation is required to be electronically readable and modifiable with a text editor. Documentation which does not meet this criterion (structure charts, for example) is reorganized to comply, or eliminated. Unwieldy media (e.g. paper-only copies) are thereby eliminated and the update procedure is simplified considerably. Because all documentation is computer readable, automated software tools can be employed extensively to streamline much of the effort.

The number of documentation components is reduced to three. The first component consists of all specification, design and maintenance documentation. This component is generated and maintained inline within the source code module. The second documentation component consists of a users' guide that describes the operation of the software and contains no design or maintenance information. The last documentation component contains all testing information and consists of the test plan (test description) and the test report (test results).

The inline documentation component consists of a standard documentation template that resides at the beginning of every software module. The template is divided into ten sections, each of which is dedicated to a particular category of documentation. These categories are organized into narrative documentation and tabular documentation classes.

The narrative documentation categories are formatted as paragraphs, and is divided into sections for module purpose, history, functional description, assumptions and limitations, special comments, references and pseudocode. These sections contain information documenting the requirements, previous maintenance activities, algorithm, procedure, general data structures and special features of the associated module. Automated tools are provided to support the generation and update of narrative documentation.

Several special components of the narrative documentation class are notable. The pseudocode category contains a concise, but detailed description of the module procedure. Pseudocode is expressed as

structured English statements associated with keywords that denote control information. As such, the pseudocode comprises a very high level, structured program-design language with fully bracketed syntax, and is used to express the detailed design of a module. Pseudocode is easily translated into code at implementation time, yet it is far simpler to comprehend for either the designer or the maintenance programmer. Tools are provided that format and verify the syntax of this documentation.

The second special construct provided as part of the narrative documentation is a module structure chart. This documentation uses a tree structure to represent the hierarchical organization of the module and all subordinate modules. This allows a maintenance programmer to determine the precise calling structure of a system or subsystem. Tools are provided to build the structure chart automatically and to format the structure chart in any of several ways.

The tabular documentation class is dedicated to describing specific data items (symbolic constants and variables), data types and subprograms. There are three tabular categories: interfaces (for documenting the routine's formal parameters and external files), global identifiers (for documenting objects that are visible outside the declaring routine) and local identifiers (for objects of local scope).

Each documentation table is organized into a series of columns. The names of the objects being documented are listed alphabetically in the leftmost column. Succeeding columns contain information which specifies base type, attributes, parameter passage mechanisms and other information. The rightmost column contains a definition of the object. Every identifier that appears in a module must be documented in one of the tabular categories. Automated tools are available to construct the tabular portion of the template and complete most of the entries in each table (including, in many instances, the description). These tools guarantee that all identifiers are indeed included in the tabular documentation. In addition, the tools delete tabular documentation for objects that no longer appear in the module, and thereby contribute to keeping the documentation current. In this manner a complete, uniform and accurate data dictionary is maintained within the source code.

The WNR methodology is designed to produce progressive and cumulative evolutionary documentation. The WNR philosophy organizes all documentation according to its intended audience. Documentation that is of principal use to designers, programmers and maintenance personnel is placed inline with the appropriate source code. Thus, all specification, design and maintenance documentation is maintained within the corresponding source module and the test plan is contained within the testing software. This close association of documentation with source code has the advantage of making all documentation immediately available with the associated sources. This enhanced access to the documentation makes documentation updates simpler to perform and less likely to be forgotten.

The users' guide documentation is targeted for individuals who do not require the detailed knowledge of a developer. Users' guides are therefore organized to contain practical information that is required to integrate the software into an application. Detailed information regarding the design and implementation of the software is deliberately omitted as irrelevant to the users' needs. As such, the users' guide documentation is maintained separately from the source code.

The above organization for documentation allows the various software baselines to be expressed entirely in terms of documentation components. In this manner, as the development progresses through each stage of the life cycle, each baseline contributes to completing the documentation for a particular system or module. This approach produces source modules that evolve from their documentation. Every module begins as an empty documentation template that is completed in stages until, as the final step, code is generated.

The specification baseline, for example, is comprised of specific categories of narrative and tabular documentation that define the requirements and constraints for a software system. The modules in which the specification baseline is built eventually evolve into the system's executive routines. Requirements and constraints are specified in narrative format in the purpose, assumptions and limitations and special comments categories. References are provided as needed. Top level interfaces are defined in the interfaces documentation table. The general algorithm is described in the functional description narrative category.

The design baseline is also constructed principally within the documentation template. Executive logic design is appended to the specification baseline. This corresponds to pseudocode that details the top level procedural flow as well as a preliminary structure chart to document the system calling hierarchy. Detailed subprogram design is then accomplished by completing all narrative documentation and the interface tabular documentation for every module in the system (as derived from the structure chart). This includes a functional description and pseudocode for each module. The users' guide is then composed for the software system. It should be noted that through this point in the development no code has been written.

Upon completion of the design baseline virtually all documentation which is required to operate and maintain the system is complete. Only the data dictionary documentation and the testing documentation remain to be developed. The former can be (to a large degree) automatically generated after the coding is completed. The latter is developed concurrently with the coding activity and as a result of executing the test procedures.

The advantages of this approach are numerous. Detailed documentation is available to guide the programmer in translating the design into code. The resulting code is therefore much more likely to reflect the requirements and design than if no such guidance were available. Several levels of procedural documentation (functional description and the more detailed pseudocode) are provided, thereby simplifying the

implementation and future maintenance operations. Problems of omission and inaccuracy which are normally associated with retrofitting documentation to existing code are avoided entirely.

The test plan baseline contains documentation for each test case to be executed. For each test case, this information includes a unique identifying number, a description of the function tested, the inputs required and a description of the expected outputs (for comparison with test results). The test data pack is also specified as part of the test plan baseline. The test procedure is ultimately derived from (and will reside in the same module as) the test test plan baseline.

The implementation baseline consists of all source modules (with completed documentation), the test plan and test procedure module, the users guide and the test report. In particular, all tabular documentation is completed. The implementation baseline thus emerges as a fully documented, progressively developed software product.

Configuration Management

The last element of the WNR methodology is a strategy for controlling access and tracking changes to the components of each software system. This configuration management strategy is based upon a philosophy of modular maintenance: during the conduct of any maintenance operation, only those modules that actually require modification are made available to the maintenance programmer. All maintenance operations, including a list of modules to be modified, must be certified by the CCB prior to the initiation of any maintenance activity.

These very strict policies derive from the mission-critical nature of the WNR software: failures in system components can result in total disruption of the facility, with serious economic, political and scientific consequences. By rigidly controlling access to system sources, errors introduced through inadvertent or unauthorized access are eliminated. Backup configurations are provided for use in the event of the failure of a primary system component, thereby allowing the system to remain operational (although at reduced capability) while the primary component is under repair.

The WNR configuration management system is built around an automated tool that provides a secure repository for controlled modules as well as a command language for automating configuration management activities. Only one individual, the Configuration Manager (CM), has access to controlled modules. This individual is responsible for moving software (implementation baselines) into the controlled environment after certification by the CCB, and for releasing modules which have been approved for maintenance to the designated maintenance programmer.

In order to request maintenance (either enhancement or repair) on a particular system or component, a work request must be submitted to the CCB. The request is reviewed for accuracy, relevance and importance. If approved, it is assigned to a programmer for determination of the work required and the modules affected. The analyst's

recommendations are reviewed by the CCB and, if approved, the designated modules are released to a programmer for modification. The standard WNR methodology is utilized to perform the maintenance activities and review the results. Upon completion and subsequent certification by the CCB, the new implementation baseline is admitted to the controlled environment.

Logs of work requests are maintained to track the status of all software products. Automated tools are provided to simplify the process of completing, submitting and cataloging these requests. Other tools are provided to automate the process of moving modules into and out of the controlled environment as well as for rebuilding systems and libraries affected by maintenance operations.

Conclusions

In the preceding sections we have detailed the features of a powerful software development methodology that is suitable for use by projects of small or intermediate size. The methodology is designed to maximize the reliability and maintainability of software components developed from it. This is done by emphasizing phases of the development life cycle that are generally ignored by small projects: specification, design and testing. Exhaustive documentation is progressively generated for each phase, and the coding activity is demoted to consume minimal project resources. A suite of automated tools is provided to support all phases of the development effort.

Although the WNR methodology is designed to minimize the overheads imposed upon programmers and software managers, application of the methodology undoubtedly increases the development time for a software product. Estimates of the additional time required for software development are necessarily subjective, but a conservative estimate for this methodology indicates a programmer overhead of approximately 200%. Utilization of the configuration management strategy described is expected to place a burden of an additional 15% upon one participant.

The predicted overheads are based upon the increased effort that must be expended by each participant during the specification, design and testing phases (the time required for coding should actually decrease significantly). Peer reviews also consume a significant amount of time, both in preparation and execution. In order to maintain a proper perspective, however, it must be noted that the resulting software product is of significantly higher quality than a comparable system developed with traditional methods. Documentation is complete and accurate. Formal testing introduces a level of reliability that cannot be attained through the ad hoc exercising that might otherwise be performed. In summary, although the development time is tripled, the resulting software is exceedingly more reliable, maintainable and robust. This enhanced quality is expected to manifest itself in much longer mean times between failures as well as simplification of enhancement/repair activities.

The WNR methodology may also be tailored to the requirements of individual projects, particularly in terms of the degree of implementation and accompanying overhead. The simplest subset to implement retains the policy of evolutionary documentation and combines the specification and design phases. No peer reviews or formal testing are performed. This implementation significantly reduces the overheads associated with the methodology, although the quality of resulting software can also be expected to be much lower. The addition of peer reviews provides a very powerful means for improving software quality at the expense of increased overhead. A formal testing program and configuration management procedures then provide the full benefit (at maximum cost).

Whatever the degree of implementation, the WNR methodology promotes enhanced software quality by shifting a major portion of the development effort to the early stages of the life cycle. Regardless of project size, increased attention to specification and design issues will always produce a better result.

Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy.