Implementation of a Software Configuration Environment

Dr. Bakul Banerjee

Switching Systems Division, Rockwell International Corporation
Downers Grove, IL 60515

Abstract
This paper demonstrates the techniques used and experiences gained while implementing a new system development process. The objective of the new process was to structure the system development environment to reflect standard software life cycle management methodologies, so that the reliability of the product can be increased while reducing management costs. Although a need to change the software process was felt for a while, the involvement with the development of IEEE Software Engineering standards provided the additional momentum to initiate the change. Throughout the change process, these standards were used effectively as selling tools to management as well as planning and training guides. The transition was made from the existing software development environment structured around the software features to one structured around the life cycle process and management of changes.

The design of the new software environment and associated management tools was based on simple software configuration management elements, except that they were accounted for in a reverse order. The decisions driving the design were made by following standard configuration management strategies: 1) Traceability management, 2) Baseline management, 3) Change management, and 4) Configuration identification.

The overall software environment, called Software Configuration Environment (SCE), composed of a set of individual environments, procedures, and tools. For each product release, one self contained environment was created per phase. These environments were placed in a horizontal pipeline. Each environment was designed to be a self contained set of UNIX® files under the control of Source Code Control System (SCCS) sharing a common history file. The SCCS version identifications were used to distinguish between types of deliverables, documents and code, as well as phases. Each product release was designed to be composed of all entities in the corresponding pipeline. Each new release was an extension of a previous release. This design could support various custom releases effectively. All associated information were tracked by two additional interconnected databases. The first one tracked the user level change requests and problem reports and the second one maintained records of application of resulting fixes in documents and code.

Key words: Configuration Management, Software Configuration Environment (SCE), Traceability

UNIX is a registered trademark of AT&T Bell Labs.
Introduction

Two major issues encountered by any management team for moderately large to large software projects, are:

- How to track the development efforts efficiently throughout the software life cycle?
- How to manage risks associated with changes to achieve highest level of output quality for every phase of the software life cycle?

As software projects get larger, it is critical to devise economical means of tracking changes to requirements and other outputs as well as propagate necessary changes to impacted life cycle documents smoothly. It is also important to allow developers the highest level of flexibility to implement and test changes adequately without impacting others adversely. Finding a balance between control and flexibility is a challenge to all software organizations. Today, various software CASE tools offer several life cycle management capabilities to fulfill just this need. However, for an existing software organization, introducing a brand new CASE toolset may not be the ideal solution. Major issues arise from the time and cost involving the following factors:

1. Migration of existing software base to the new tools environment
2. Retraining of software engineers to work under the new environment
3. Purchase and installation of new tools
4. Modification of special internal tools to fit the new toolset

In our organization, like many other organizations in a similar situation, complexities of the transition process was considered to be high, because: 1) the software was a part of a large telecommunication system with a distributed architecture, 2) a large portion of the software had to be reused, 3) due to the aggressive schedule plan, no time could be spared to prepare for the new tools. In the beginning, some soul searching was needed. It was decided that we need a model that will support the following capabilities:

1. To manage software deliverables as bundles associated with releases and milestones and to be able to propagate necessary changes to the bundle as the components progress through various life cycle phases and milestones
2. To allow modifications to the software at various stages of life cycle and assure the propagation and accountability of resulting fixes
3. To package software by functionalities to a level of granularity that will optimize the incremental software construction
4. To conduct verification and validation of software as automatically as possible during code implementation and testing phase.
5. To allow developers maximum flexibility to apply fixes in the phase that the problem was found
6. To do all of the above, to be able to trace requirements to software code so that the final product fulfills all original requirements

We realized early that it was best to use existing established life cycle model as a basis, and develop supporting procedures and tools to control and automate the model. The new process model had to be implemented, so that the changes were transparent to users. Since the project was already in progress, this was the toughest challenge to face in the whole project.
Software Configuration Environment

The new process model, called Software Configuration Environment (SCE) model, accepted engineering efforts to produce software deliverables (documents, code, and testing) as input. The model is described in Figure 1. Outputs were: Verified and validated software and various management aids. Most of the management aids were in the format of system generated status reports or on-line status accounting mechanisms. Additional advantage of this model was the capability of the process control data collection. Within this environment, meaningful statistical data could be collected for further process improvement analysis.

The objective of this environment was to organize software deliverables and tools to establish a trace among various components of the software deliverables. All items, that were part of the the software development process, were marked as Software Configuration Items (SCIs). These SCIs were packaged, controlled, and tracked. SCIs could be document or code packages composed of any number of files ranging from one to several thousand. Packages were kept under control by auditing them according to predefined package rules, when they were ready to be released to officially controlled area. In essence, we viewed the model as the solution to a configuration management problem and extended the framework to assure software quality, to effectively manage software projects, and to implement cost-effective verification and validation. The solution was devised using traditional configuration management strategies as well, except in reverse order. These strategies, ordered according to the hierarchy of increasing order of details, are described below.

1. Traceability management at the general project level (independent of the time frame)

2. Baseline management at the product packaging level (dependent on the time frame)

3. Change Management at the individual component level (highly sensitive to time)

4. Configuration Identification at the product planning level (somewhat sensitive to time)

Traceability Management

Our primary objective was to trace all requirements to elements of final software product. The goal of the model was to be able to organize and connect all SCIs according to an established hierarchical traceability rule. This allowed us to able to trace dynamic changes to an SCI to other impacted SCIs residing in all phases, past, present, and future, at all times. Consequently, if an SCI could not be connected by the existing traceability rule, exceptions could be flagged, reviewed, and tracked. Occasionally, if appropriate, new or modified rules were defined as well. The traceability management dealt with the product development at the project level. That is, it considered the global picture independent of the state of the project at a particular time or the state of any individual deliverable. Figure 2 shows the overview of the traceability management.

Baseline Management

The second objective was to develop the project by developing incremental baselines by milestones as well as by developing concurrent (but staggered with short time lag) releases to supply the market place with new releases in less than average time. The SCE model allowed for the assignment of each code and document configuration item to a particular milestone as well as to a release. These information were stored in a database, called the Software Configuration Database, and the software was
SOFTWARE CONFIGURATION ENVIRONMENT

Figure 1. Software Configuration Environment (SCE) Model

Figure 2. Organization of Software Configuration Items and Tracing
built and verified according to the package defined by the database. This allowed us to be able to track baselines through life cycle phases to reflect the true status of the code and document packages delivered. Baseline management addressed a narrower view of the project than the traceability management. It dealt with bundles to be delivered per software release, and at a lower level, per milestone.

**Change Management**

Our third objective was to be able to control, track, and apply all changes effectively. We sought to automate these change management problems with minimal amount of impact on developers and their interface with familiar existing tools. As shown in the Figure 3, changes in software were considered from two levels. They are:

1. Changes made to the software due to fixes applied in the same or later phases and automatic propagations of changes that occurred.
2. Changes made by developers as they released their SCIs (document or code) from their own development areas to controlled official areas.

The software development was divided into two major areas, officially controlled area and development area. Only four coding phases are elaborated in Figure 3. These phases are: Unit code development and testing, functional integration testing, stable base, and system test. The stable base always contained a relatively stable set of code, that could be picked up by developers for integration or unit testing. Release of SCIs from the unit development area to the unit official area was allowed at all times when the associated baseline was open in the phase. However, a fix in other phases could be applied only when a formal problem record was posted and tied to the fix. Changes across phases were controlled strictly with required approvals of supervisors. Although high volume change propagations to code residing in implementation, integration, and test phases occurred, available change management tools provided accurate management information. The change management mechanism handled the project at the phase level, one level down from the baseline management.

**Configuration Identification**

Our last objective was to identify each component of the deliverable software, namely, generic as well as third party documents, code, and data. We were in the process of developing a large system with a large amount of existing kernel code and some application code. Most of the existing code was not structured well and kernel code had complex interfaces. The associated documents were not partitioned adequately to support the new code to be developed. Within our life cycle model, controlling software promotion of code as well as documents was given a high priority. So, a large amount of effort was spent of simply organizing the documents and code and establishing relationships. It was also critical to train developers on the packaging of documents and code.

Configuration management addressed the project at the lowest level, namely, at the package level as well as bundling of packages within a bundle. Many unique characteristics of the project became visible at this level. Establishment of package rules allowed us to implement many rule checking audits that were used by developers as well as controllers to maintain the sanity of each released deliverable.
Figure 3. Software Coding, Fixing, and Testing

Figure 4. Toolset For Software Configuration Environment
Software Configuration Environment Design
Considerations

The major design considerations of the new environment were as follows:

1. The environment should be able to track the software requirements throughout the software life cycle for completeness, by accounting the status of the configuration items in the associated trace. A typical trace contained following items in the particular order shown in the Figure 2.

   Requirements documents
   Design and architecture documents
   Code modules as well as builds
   Test Procedures

2. The environment should support phased development, integration, and system testing, while providing developers a pointer to a stable set of code at all times that could be used as a stable base during dynamic unit and integration testing.

3. The environment should support application of fixes to the code in the phase the problem was found, so that, under certain rules, applied fixes might propagate automatically to code residing in previous phases.

4. The environment should have the capability of baselining a particular trace configuration per milestone and per release.

5. The environment should be able to provide status reports of verification and validation of the software product on demand.

6. The environment should have the forward (or backward) tracing capability. That is, given an SCI, the environment should be capable of tracing all corresponding software documents to their children (or parent) documents according to established hierarchical rules. The environment should also be able to produce reports identifying corresponding completeness status of each trace with respect to predetermined requirements fulfillment criteria.

7. The environment should be able to reuse other existing tools being used in the project. New applications of existing tools should be devised to compliment this environment. New tools development should be kept to the minimum and they should be introduced with minimum impact to developers.

Software Configuration Environment
Methodology Considerations

An informal process group collaborated together to develop the model, rules for the process, and recommendations for tools. Additional support was sought from lead engineers constantly. All new and enhanced tools were introduced to developers in phases as recommended by the process group. Continuous user support on procedures and tools were provided using hands-on as well as formal training. Rest of this section briefly outlines the methodology adopted. Figure 2 and 3, introduced earlier, show some of the details of the model.

The organization of files followed the software breakdown structure as well as the organizational structure. Documents and code were packaged at various levels of granularity. For example, requirements were broken down by features to be delivered, independent of the architectural considerations. Requirements were broken down into modules, called feature groups, allowing the system engineering group to control requirements at a high level as well as at a low level. The new organization allowed us to create complex documents authored by several people at the same time. Test procedures were packaged to correspond requirements to allow for verification and validation of
requirements. Direct traces were established between requirements and test procedures.

Since design components were not dependent on functionalities, but on the architecture of software and hardware, only an indirect trace could be established between requirements and design. However, this organization allowed us to establish a strong tie between design and code components. The first level of design contained architecture documents detailing the construction of the software and hardware. These SCIs outlined structures of the software system as well interfaces among components of software and hardware. The structure of the software build was outlined in these architecture documents, by defining the outline of the software build code packages. These build packages were identified as SCIs and were traced to the corresponding architecture document. This established the basis of change management of the software build. Detailed design documents were written from and directly traced to architecture documents. Design documents also outlined interfaces between various components of code. The structure of the code necessary to establish these interfaces were laid down as a part of the design document. Appropriate code packages were identified at the conclusion of the design. These code packages, called interface packages, were traced to corresponding design documents. Code modules written from each detailed design document delivered one or several capabilities desired. These code modules, bundled according to capabilities described by the design, were designated as capability packages and were traced to the design document.

All information related SCIs and their traces were maintained in the Software Configuration Database. One of the critical information maintained by this database was the relationship between milestones and SCIs. Using this relationship, the code was built incrementally to deliver small chunks of functionalities per well-defined baseline for a milestone or a release. The content of each baseline build was extracted directly from the database. This allowed us to build the software in such a manner that any build contained all SCIs to be delivered for that particular baseline and nothing less. However, developers could continue working on any current or future baseline and apply fixes to past baselines. The software construction process continued concurrently with the development of design. Excess code was allowed in a baseline only to provide some future hooks and was negligible in size. At the same time, the database contained all information of test procedures.

Each new release was an extension of a previous release. This design could support various custom releases effectively, while maintaining the fix propagation from earlier releases simple.

Figure 3 shows the structure of code related life cycle phases as multiple UNIX® environments with shared SCCS history. Sharing history among files in all four phases allowed us to propagate changes due to fixes effectively. Each environment was divided into two areas, official and development. Official area was always under control. A code package (or document for document phases, not shown in Figure 3) could be released to the official only after the prescribed verification and validation were complete. Each release promotion was associated with well-defined procedures for verification and validation. The fix propagation rules were prescribed and implemented mostly automatically via the database, except for the manual intervention by managers and supervisors when the disposition of resolutions

UNIX is a registered trademark of AT&T Bell Labs.
were not clear cut. The database also maintained information about all fixes.

**Tools Associated with Software Configuration Environment**

There were three major tools components of SCE that held together and implemented supporting procedures.

1. **Software Configuration Database**: a database system to track and manage information of all code and document packages and tied to the Software Configuration Database.

2. **Change tracking system**: a database system to track, report, and analyze change requests and problem reports.

3. **File control system**: a file management environment and toolset to manage files in controlled and development levels, as well as, at phase levels.

Figure 4 shows relationships among these three components.

The environment was designed so that users populated most of the database transparently during normal user operations. The release of a package to the official area initiated several audits to check internal consistencies in the code. Although it was not difficult to sell the idea of execution of associated audits, the underlying concept of custom packaging of documents and code was difficult to sell. However, in the long run, the packaging concept proved to be the best part of the model, since it dictated engineers to design the software in an uniform manner based on the peculiarities of their subsystems. Custom packaging of code required significant amount of preplanning on the part of developers. This activity resulted in a stronger design.

Tools were used, directly or indirectly, by all engineers and management. For example, Software Quality Assurance organization was able to use the tool to collect various process data at each stage of the software life cycle. Management used a variety of daily reports on the status of SCIs to conduct incremental checkpoint audits of the evolving software product.

**Standardization and Software Configuration Environment**

One of the conclusions of initial investigations, that led to the new process model, was the difficulty of implementation of standards and procedures. Although the organization was committed to software quality and the need for standardization was well-recognized, it was up to the individual engineers to implement standards. Little process imposed control was available to implement standards. The goal of the model was to realize the essence of several IEEE software standards. Standards were consulted regularly for planning and selling to upper management. Customization tips in newer standards proved to be useful.

The design of the process model depended heavily on the IEEE Standard for Software Quality Assurance Plans[1]. The new model aimed at solving the following existing process problems:

1. Outputs of various phases, particularly for code, could not be isolated easily.

2. The set up of file control environments did not allow problem fixing at the phase it was found. So, fixes could not be isolated from development.

3. The exit criteria from a phase could not be verified, as the transition between phases could not be partitioned effectively.

Our next concern was the verification and validation of the software. It was critical to make
the verification and validation process, as defined in the IEEE Standard for Verification and Validation Plan[21, as smooth as possible. So, we needed to automate most of the verification activities, so that software development could progress with minimal manual interventions. The new model allowed us to delegate most of the verification responsibilities to developers while rules were determined by the process group. The execution of the verification was made simpler by providing engineers with various custom audits.

The design of the process model was based on the IEEE Guide to the Software Configuration Management [3]. Since the change management was our primary concern, the design of our process pivoted around this standard. Personal involvement with the development of this standard provided insights about the importance of accommodating dynamic changes in the product. This standard also provided a basis for the source code packaging and modularization of the software. This standard was also used to establish the control of the software build process, promotion between life cycle phases, and management of all fixes and their propagations.

Project and Product Impacts

Experiences gained to minimize impacts of changes in early phases of implementation of the model were effectively used to minimize risks in later phases. At the planning stage of the process design, we considered both product and project issues and prioritized them. High priority product issues were:

- Since the product was based on a distributed architecture, the platform software, that established basic interfaces, might change often. So, we had to provide for robust interface control mechanisms.

- Since multiple releases of the product were being developed in parallel to meet narrow market windows, we had to provide a mechanism to isolate software components to minimize impacts on the cohesiveness of the product coexisting in different phases and impacts on work groups.

- Since the product itself was not fully planned out, we had to plan for major changes in higher level documents, namely, requirements and system architecture, to propagate to the code module level relatively easily. We had to provide for adequate tracking and reporting of these changes as well.

High priority project issues were:

- Limited resources were available to purchase, develop, or train a brand new set of software management tools. So, existing support tools had to be repackaged and a new software configuration database had to be developed to integrate existing tools to manage software components and change markers to them.

- Since most of the line managers were using the old process for several years, the change had to be implemented with minimum impacts on management style.

To prove the immediate pay-back, we were able to demonstrate following benefits early on.

1. The provision of tracing the software to the set of detailed deliverable capabilities kept various phases coherent and minimized the rework cost. To do so, we provided the code packaging and trace auditing capabilities and tools to manage them. Early prepackaging of the code reduced the time spent on rework of documents in previous phases.

2. The productivity during integration and system test was increased by more than 200% by providing the method of smooth transitions of smaller chunks of capabilities. It was done by integrating
the software according to the capabilities to be delivered per milestone, not by whatever code was done.

Benefits of the new methodology of software trace management and packaging within the SCE environment were realized within a short time. Major areas of improvements were:

1. The time to assemble the first software load was reduced by at least 300% from the time taken by an earlier load of similar size and complexity.

2. Contents of the build in terms of code packages and their status per each milestone was reviewed daily by the management team using simple reports. These verification reports improved the productivity significantly from the previous methodology. In the previous methodology, only report available was lists containing thousands of file names in no particular order and without any change information.

3. Since the rules for software packages were automated, code could not be promoted to the controlled area without the rule checking audits. This need of early planning for code packaging led to a better understanding of the software construction.

Conclusion

Many factors may be attributed to the success of the project. The most important of them was the attribute of customization. Within the SCE environment, the software environment, tools, as well as methodologies could be tailored to the particular management organization as well as the software delivery plan. Direct involvement with the planning process of the software, including definitions of requirements and architecture, formal change management strategies, and availability of flexible software tools, also contributed to the success of the project. This environment also laid down the groundwork for future software metrics programs for the project.

References