Quality Assurance Framework for Distributed Collaborative Software Development

by

Charles K. Njendu
Bachelor of Science in Civil Engineering
Mississippi State University, 1997

Submitted to the Department of Civil and Environmental Engineering in Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING
IN CIVIL AND ENVIRONMENTAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 1998

Signature of the Author [illegible]
Department of Civil and Environmental Engineering
May 22, 1998

Certified by [illegible]
Feniosky Pena-Mora
Assistant Professor of Civil and Environmental Engineering
Thesis Supervisor

Certified by [illegible]
Professor Joseph M. Sussman
Chairman, Department Committee on Graduate Studies

Quality Assurance Framework for Distributed Collaborative Software Development

by

Charles K. Njendu

Submitted to the Department of Civil and Environmental Engineering on May 22, 1998 in partial fulfillment of the requirements for the degree of Master of Engineering in Civil and Environmental Engineering

ABSTRACT

This thesis focuses on quality assurance within the sphere of distributed collaborative software development. Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures. SQA includes the process of assuring that standards and procedures are established and followed throughout the software acquisition life cycle.

A distributed collaborative software development endeavor was conducted at MIT in Cambridge, Massachusetts and at CICESE (Centro de Investigacion Cintifica y de Educacion Superior de Ensenada) in Ensenada, Baja California, México during the 1997-1998 academic year. This endeavor was in the form of a Distributed Software Engineering Laboratory (DISEL). This thesis analyzes and assesses the process and method applied within DISEL as well as other existing processes and methods.

The analysis covers concepts, functions, standards, models and plans developed for the software process from a quality assurance perspective. The assessment involves questioning the SQA plans and methodology implemented, what could be implemented more efficiently and effectively the next time around and how this will be done.

To do so, this thesis then develops a quality assurance framework for collaborative software development based on the results of the analysis and assessment. The framework developed exposes several important attributes necessary for a successful software quality assurance plan for distributed collaborative software development environments. These are ISO quality and the Capability Maturity Model guidelines; excellent communication channels with an organic structure; the existence of a well-structured data, documentation and code repository; independence of SQA team members from the project where possible; a shared culture; incentives; and rapid prototyping with iterations i.e. iterative enhancements.

The resulting framework will be useful in improving the quality assurance process of the next cycle of collaborative software development in the distributed environment present at MIT and CICESE.

Thesis Supervisor: Feniosky Peña-Mora
Assistant Professor of Civil and Environmental Engineering

ACKNOWLEDGEMENTS

I would like to thank the many individuals and acknowledge the many factors that contributed to the completion of this thesis.

For guidance and support I thank my advisor, Feniosky Peña-Mora who also had to put up with a lot of my doubts and taught me the ability to think of the big picture framework as well as make the small details work in order to make all the last minute direction changes that I needed to perform.

Thank you to David Marks, Charlie Helliwell and Cynthia Stewart for keeping me on track within the M.Eng. Program requirements. To fellow M.Engers, Amy Hwang, Bernard, Derrick and Jeri. Thanks guys.

To the other Information Technology team members including Bob Yang, Simonetta Rodriguez, Emilie Hung, Christine Su, Kareem Benjamin and Gregorio Cruz, I am grateful to have been a part of this pioneering effort to apply information technology in achieving Casual Contact and Social Interaction in the Virtual World.

To "mum" (Mrs. Sarah Githinji), thanks for your support and enthusiasm.

To Peter Israelsson, Thu Nguyen, Vittorio Agnesi and John Laplante, for being good friends. To Tim Wuu and William Gilreath, it is great working with you. I learnt a lot about Information Technology, both from a practical and theoretical aspect respectively.

Ana Pinheiro and Amparo Flores, thanks for your support and invaluable friendship and Ricardo Petroni, thanks for being who you are.

Thank you Simonetta for pulling me through a few tough times and for the enlightening and intellectually challenging conversations on a variety of topics that I had the opportunity to have with you. I hope these will continue.

Conny, this thesis and a good part of my quality of life would have been impossible without you.

Liz, Mary, Grace and Kenneth, you are the most supportive siblings a brother could have.

To Ayn Rand, Bill Gates, and the United States, for most of what I value, this life.

My late Mum, belated thanks for all those years. I love you.

Dad, thank you for everything that I have achieved so far. My gratitude for this opportunity to have studied, lived and learnt from the most prestigious and alive institution in the world, MIT. Thank you for having enabled me to be a part of the greatest civilization ever, the United States. Thank you for my life.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

This thesis focuses on quality assurance within the sphere of collaborative software development. Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures. SQA includes the process of assuring that standards and procedures are established and followed throughout the software acquisition life cycle.

A distributed collaborative software development endeavor was conducted at MIT in Cambridge, Massachusetts and at CICESE (Centro de Investigacion Cintifica y de Educacion Superior de Ensenada) in Ensenada, Baja California, México during the 1997-1998 academic year. This endeavor was in the form of a Distributed Software Engineering Laboratory (DISEL). This thesis analyzes and assesses the process and method applied within DISEL as well as other existing processes and methods.

The analysis covers concepts, functions, standards, models and plans developed for the software process from a quality assurance perspective. The assessment involves questioning what was implemented, what could be implemented more efficiently and effectively the next time around, and how this implementation should be handled.

This thesis then develops a quality assurance framework for collaborative software development based on the results of the analysis and assessment. The resulting framework will be useful in improving the quality assurance process of the next cycle of collaborative software development in the distributed environment present at MIT and CICESE.

1.1 IMPORTANCE OF COLLABORATIVE SOFTWARE DEVELOPMENT RESEARCH

Software systems are increasing in the level of complexity while facing the associated problems of higher costs, lower quality and frequent changes and re-workings. With geographically distributed talent comes the need for inexpensive, convenient, natural ways for software personnel to collaborate in distributed environments. Computer Supported Collaborative Work (CSCW) is one result of this need (SEI Collaborative Skills in Software Engineering, 1998). Due to the inability to map all aspects of physically co-located collaboration onto the distributed environment, software developed in a distributed environment has the added complexity brought about by this distance.

These distributed collaborative teams are being used to increase worker involvement, improve quality and productivity by making use of geographically dispersed core competencies, and reduce the complexity of the organization involved by helping flatten, downsize and decentralize it. A good part of what hinders the development and successful implementation of collaborative software development is the infancy level of the interpersonal skills software professionals and the way they handle the different phases of the development process such as requirements analysis, peer reviews and project management (SEI Collaborative Skills in Software Engineering, 1998). This deficiency affects any work activity involving two or more people, and is particularly evident in team work (such as systems work) and group interactions (such as group meetings) (SEI Collaborative Skills in Software Engineering, 1998). In addition, depending on the constitution of the team, additional hindrances to the development of these interpersonal skills may develop. In the DISEL project these included issues such as cultural and language differences as well as long distance communication protocols.

Lack of interpersonal skills is seen as the most consistent barrier to successful software development (SEI Collaborative Skills in Software Engineering, 1998). This barrier is especially evident in collaborative software development where excellent interpersonal skills are at a premium due to the added complexities of working at a distance.

The issues involved "can be characterized as an inability of groups to capitalize effectively on the cumulative talent and technical skills resident in the participating individuals, because they have no commonly understood, accepted, or enacted ground rules for working together on their shared technical tasks" (SEI Collaborative Skills in Software Engineering, 1998). In addition, when two or more people come together to work on a stated objective, there stands the possibility, as emerged in the MIT distributed class, of different people having differing expectations and goals within their involvement in the project.

1.1.1 DISTRIBUTED SOFTWARE ENGINEERING

Software consists of the programs involved, their data and documentation (Jalote, 1997). The sub-set of software development involved is collaborative software development (distributed software engineering). Working definitions are as follow. Software development or engineering is a broad discipline that include and refer to an integrated set of methods, procedures, and tools for specifying, designing, developing, and maintaining software (Flecher and Hunt, 1993). Software Engineering, in a working framework, involves having a philosophy and principles from which a Software Engineering approach, environmental support, a methodology and techniques and tools are derived (Flecher and Hunt, 1993).

1.1.2 THE NATURE OF SOFTWARE, SOFTWARE PROCESSES AND PEOPLE

No technology and in fact no human endeavor has as complex a nature as software and its interaction with humans. For the developer it is the source of fun as it brings together the realization of youthful dreams that are the stuff of magic. This "magic" involves the program's ability to print, move physical entities whereas it is no more then an abstract construct of the developer's mind (Brooks, 1995). To the user it is the ability to perform a multiplicity of tasks with one machine but with program obsolesce occurring at a rate that is unimaginable in previous technologies (Brooks, 1995).

Software is an extremely tractable medium placing very few bounds on what we can express and create. The products created are some of the most complex artifacts of human achievement with even greater complexity if their relationships and interaction are taken into consideration. It is an abstract construct in which one is always learning and creating from pure imagination to produce an abstract that moves concrete things. Software is complex, does not conform (i.e. there are no fundamental principles such as in mathematics or physics), requires constant change, and is invisible, non-visual and indivisible (Brooks, 1995).

This nature coupled along with a similar complexity in the people involved (such as communication issues) for software engineering processes and approaches results in software engineering systems differing from traditional engineering ones. However a system is only complex as long as the user does not understand it. When the user understands the system, it is no longer complex and becomes extremely simple.

It is the nature of humans to handle system complexity (whether in politics, economics or other) by finding and defining the principles or philosophy behind the system as simply as possible. There are standards, procedures and processes that can be followed in software engineering, with some being more applicable to collaborative software engineering then the entire area of software engineering to ensure quality. These standards, procedures and processes, which we can prematurely define as part of software quality assurance (SQA), ensure that the project is not late, over budget or that the software product is not too slow or misses the mark (client requirements). The very existence of successful products implies the existence of good SQA.

1.2 DISTRIBUTED SOFTWARE ENGINEERING LABORATORY

Defining and previewing the fundamentals of collaborative software development help set the context within which DISEL was conducted at MIT and CICESE and allow for a more concrete look at the DISEL development effort with these fundamentals in mind. It allows an understanding of the motivation for the particular effort that took place in DISEL, enabling assessment of the particular Digital Environment in which the effort took place and how this environment was to be altered, was altered and affected the participants from the perspective of SQA.

1.2.1 THE MOTIVATION

The task was to enhance the existing infrastructure of the typical digital environment for engineering development over geographically distributed areas. The aim was to map the physical occurrences of casual contact and the resulting requirements of social interaction and personal expression onto the virtual world. The objective was to have a digital system involved for casual contact, personal expression and social interaction that was as seamless and natural as that experienced when running into a colleague, say, on the way to a coffee break and having an unexpected and fruitful interchange. This was a continuation of absracting human interaction in the physical world onto the digital world. The interaction developed was to replicate or replace the interaction that occured when company people and colleges students are worked in close proximity to each other. The resulting system was named Cliq!. It was an entirely new paradigm.

The niche for Cliq! was evident from the creativity, brainstorming, idea definition, formulation or simply pure remembrance that occurred within casual interactions in the physical realm. Immense possibilities were opened up by the ability to map this peculiarity onto the virtual world where most of humanity was increasingly spending its time. In addition, as the capabilities of the virtual world became more similar to the physical world, new virtual behaviors came into existence mapping old physical ones such as the requirement for a new type of etiquette (netiquette). Initially, like all software paradigms, the digital system developed required a lot of commitment for regular casual communication to occur. It was a cumbersome but improving environment that is necessary in order to establish the same type of effortless physical environment where physical casual contact, personal expressions and social interaction occur.

The improvement of productivity over dispersed environments was one of the requirements for Cliq!. As an aspect and extension of casual contact, this was done by improving the ease of interaction between people and recreating a digital environment that imitated the physical one quite well, for example, by re-producing certain expressions and hand gestures that helped communicate more effectively. One goal was to have a seamless and natural system where random interaction could happen with minimal investment. Another goal of the system was to have the ease of a social environment, and not to be a monitoring process. The previous system included semi-fixed microphones and a video/audio set-up, as described in more detail below. This setup not only involved an enormous investment in time but required two to three specialists to be interacting with the system due to its instability and complexity.

A principal part of the process in developing Cliq! was the conceptual development of casual contact within a digital environment for engineering development that re-created the physical and social interactions occurring in physically co-located engineering working groups.

1.2.2 THE DIGITAL ENVIRONMENT

Prior to the DISEL effort, the Design Studio of the Future had developed a basic infrastructure that allowed teams to communicate electronically sharing audio, video, applications, and documents. The heart of the environment was a network of computers running Microsoft Net Meeting as its principal communication software tool (Microsoft, 1997). Audio and video equipment included semi-fixed microphones and video camera feeds of the room activities onto the web and across to the other ends of the distributed environment. It was set up for two users with one node at CICESE in Baja California, Mexico and the other node in Cambridge, MA, USA.

The hardware used in the environment included a video camera to capture the speaker's and listener's appearance and motion, a microphone to capture the speaker's voice, speakers to broadcast the sound from the other site, a video projector to broadcast the image from the other site and a wireless pointing device (mouse). The software used in the environment included Microsoft Net Meeting that was used to transfer the sound and synchronized the Web browser. InPerson on a Silicon Graphics machine was used to transfer the video image (Silicon Graphics, 1997). The operators in the environment included the video camera operator who captured the video image of the site. The audio quality was extremely poor especially at times when traffic on the Internet was high. Due to the capacity of the network, there was a problem with the communication process. When the transfer rate decreased, the sound quality dropped rapidly. The audio contained most of the message from the other site. Sometimes the speaker did not know whether the listener over at the other side could hear the sound clearly, resulting in lessened communication

The microphones involved were cumbersome, restricting natural body movement and expression due to their semi-fixed state. In addition, one had to lean in extremely close to the microphone in order to have them pick up what one was trying to communicate. Video quality was also a problem as well as the fixed nature of the one camera installed upon a tripod that was recording and transmitting the proceedings of the sessions to the other side. This reduced the scope of sight of most participants to what this single camera was focused on and its view range. Also, server stability depended on a lot of factors, some external to the immediate system. For example, due to El Nino on the West Coast of the U.S.A. and Mexico, lines of communication were disrupted resulting in no web-based communication between the two locations. Finally, to run this setup on either side required a team of two to three specialists continuously interacting with and monitoring the system due to its instability and compatibility problems.

Therefore, the previous system was cumbersome, not natural, and, due to the investment in time and patience required, an inconvenience to use unless this investment was greatly out-weighted by the objective one was trying to achieve. The Digital Environment was a secondary option to most means of communicating and interacting. Thus it was a challenge to implement casual contact and social interaction in such an environment. However, the Design Studio of the Future had the basic infrastructure of electronic communication environment to enable this effort, noting that the requirements were the provision of a natural way over networks for casual contact, social interaction and personal expression. What was needed was a system that could allow casual contact in this dual distributed environment and be extended to handle six dispersed groups in working or educational environments. These environments would be physically located in Japan, Chile, Australia, Switzerland, Mexico and the U.S.A.

CHAPTER 2
CONCEPTS AND FUNCTIONS

An understanding of the fundamentals of software engineering with a focus on collaborative software development validates the following observation. Collaborative software engineering development, like any other engineering discipline, can generally be divided into two aspects. These aspects are the product created and the process involved in the creation of this collaborative software product. Consequently the roles involved, activities performed and documents generated can be classified within two sets. One set is involved directly in the collaborative software-engineering product and consists of the requirement analysis, design and coding. These people may be termed as "those who do it", directly and visibly creating the software product. The other set is involved in ensuring that the processes established are optimal in delivering the required product, the software quality assurance. This set of people may be term as "those who help `those who do it,' do it", indirectly creating the software product.

2.1 DEFINITION OF QUALITY ASSURANCE

Software quality is an attribute and can subjectively be defined as "The degree to which a customer or user perceives that software meets his or her composite expectations" and objectively as "The degree to which the attributes of the software enable it to perform its specified end item use" (McManus and Schulmeyer, 1987).

Software quality assurance is an activity and is defined as "...a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirement" (McManus and Schulmeyer, 1987). An important point to note concerning software quality assurance is that "Unlike hardware systems, software is not subject to wear or breakage; consequently its usefulness over time remains unchanged from its condition at delivery. Software quality assurance is a systematic effort to improve that delivery" (McManus and Schulmeyer, 1987).

The conceptualization and development of quality assurance for distributed software systems is more marked and necessary due to the additional complexity. As software projects get larger, certain characteristics and needs become evident such as the need to have distinct phases within a given development process. Usually, each phase has a specific deliverable at its end. For example, the analysis phase has the requirement analysis document while the design phase has the design documents. Therefore, any quality assurance process should go throughout the whole development phase and allow for phasing.

2.1.1 THE NEED FOR SQA

The typical well-implemented SQA plan is justified by the following common occurrences in most software projects. As the project progresses the costs of detecting and correcting errors increases geometrically with project progress. Studies show that the cost of correcting errors in the maintenance phase can be anywhere from 30 to 270 times as costly as the costs incurred if these errors were uncovered in the design or analysis phases (Sinclair, Vincent and Waters, 1988). In addition, most of the effort undertaken by Information System departments occurs in system maintenance while half of the cost of removing errors in the testing and development phases come from errors inserted in earlier stages of the project (Sinclair, Vincent and Waters, 1988).

The primary goal of quality assurance is to avoid the two common pitfalls of software development that result in the above problems. These pitfalls are either a product that is robust and stable but does not meet the end user's needs or a product that meets the end user's needs but is not robust or is unstable.

A focus on product quality is usually half of quality assurance. The other half is process quality. The quality assurance team may be aware of the above two common fallacies and intend not to fall in them but under a time crunch, the process is usually forsaken in an attempt to obtain a product by any means necessary. Product and process quality are the most pursued and common objectives in software quality assurance. However, there is a third aspect to good quality assurance especially in a distributed, collaborative environment; the people involved. In most quality assurance plans people are only implicitly considered in the process aspect and usually only from a project management perspective. However, as noted in Chapter 1, quality issues related to people are just as essential to ensure software quality assurance. The people under consideration include the client (purchaser) and the producers (project team).

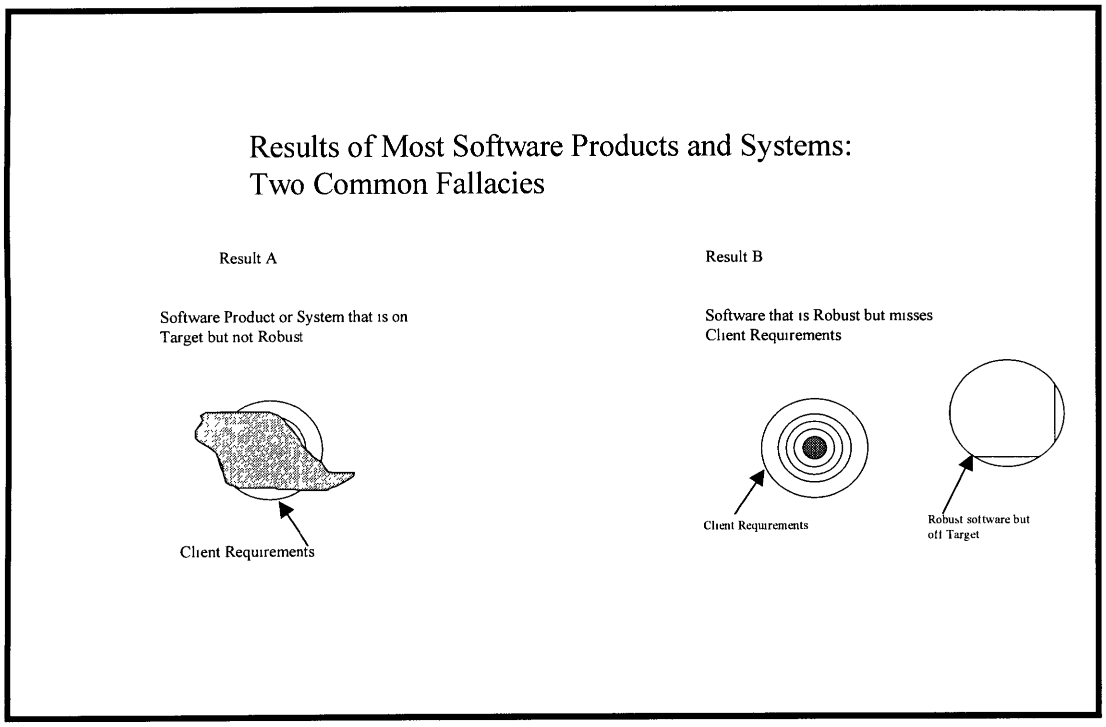

Results of Most Software Products and Systems:
Two Common Fallacies
Result A
Software Product or System that is on Target but not Robust
Client Requirements
Result B
Software that is Robust but misses Client Requirements
Client Requirements
Robust software but off Target

Figure 2-1: Results of Most Software Products and Systems

2.1.2 QUALITY AND ACCEPTABILITY

Acceptability is a key part to defining quality (Sinclair, Vincent and Waters, 1988). Problems with meeting acceptability may be anything from taking the requirements out of context to satisfying the objective aspect of the requirements without consideration of the subjective aspect or vice versa. To illustrate this concept, take as an example a contractor for a building who is given the following key requirements. The contractor is to build a home for a family and include perfect amenities. The contractor then delivers a Spanish architecture type building with large windows and white walls furnished with amenities like air-conditioning. However, if the location of this building is Greenland the large windows would clearly be unacceptable as it was built out of context. In addition, air-conditioning is not a priority in Greenland, subjectively or objectively. Unfortunately, acceptability is usually not as clear in the software industry as it is in the building industry before one has the final product.

2.1.3 QUALITY AND ASSURANCE

Quality and assurance are two separate distinct items (Sinclair, Vincent and Waters, 1988). However, the phrase as a whole does refer to an existing activity, that is the act of ensuring or assuring quality in the project.

2.1.3.1 Quality

Quality is when the software product meets the requirements at the unit and, especially, system level. This implies two actions, that the requirements established by the client are meet and that both functional and quality specifications and characteristics are meet. The quality unit function involved is circular in nature and consists of the following units: documentation, discussion and agreement (Sinclair, Vincent and Waters, 1988). It is an iterative unit function that occurs within each stage or phase of the project development. First, the requirements are documented and processes, standards and procedures are applied as necessary. Then a discussion occurs with most if not all of the members involved in the project (such as client, peers, auditors) and if a consensus is reached, any necessary modifications are made to the initial document and the unit process comes full circle (Sinclair, Vincent and Waters, 1988).

In order to satisfy the client's and software quality characteristics the discussion should result in the following. Identification of particular attributes that may be termed as constituting acceptable software (Software Quality Factors), identification of sub-attributes

(Criteria) that, if present, verify the existence of Software Quality Factors, and the provision of a weighed checklist based on "Itemized Requirements (the smallest measurable unit) of the Criteria" (Sinclair, Vincent and Waters, 1988). Clarification and analysis of the Quality Factors and their Criteria can then be done.

2.1.3.2 Assurance

If quality is the goal then assurance is the means or processes involved in the achievement of these goals. Similar in part to the quality function, the assurance process is made up of a process, documentation, review and comparison. The process involves the execution of an action (corrective or initiating). This action is then documented to capture its form (an example was the bug-tracking system that was developed in the DISEL effort). The documentation is then analyzed to ensure completeness. If satisfactory this documentation is compared with chosen standards, procedures and requirements to achieve acceptability. If not acceptable the loop is repeated starting with the process until acceptability is obtained (Sinclair, Vincent and Waters, 1998). The assurance process is similar to the quality function and can be overlaid.

2.1.3.3 The Assurance Function and Model

Both the quality function and the assurance process when focused upon are unitary and iterative. Both are singular units of a larger system as detailed in figure 2-2. As illustrated there are four fundamental phases within any software product life cycle. These are analysis, modeling, development, and realization. Within each phase the assurance process occurs. These phases are a high level view of other sub-phases that may or may not occur in software development. These sub-phases may include preliminary analysis or preliminary design. Regardless of the level of abstraction that a phase consists of, the assurance process is applicable to the phase and, on the development project level, the assurance system may be implemented. When the linearity of the phases and the circularity of the quality function and the assurance process are combined, the assurance system provides a generic Software Quality Assurance system life cycle.

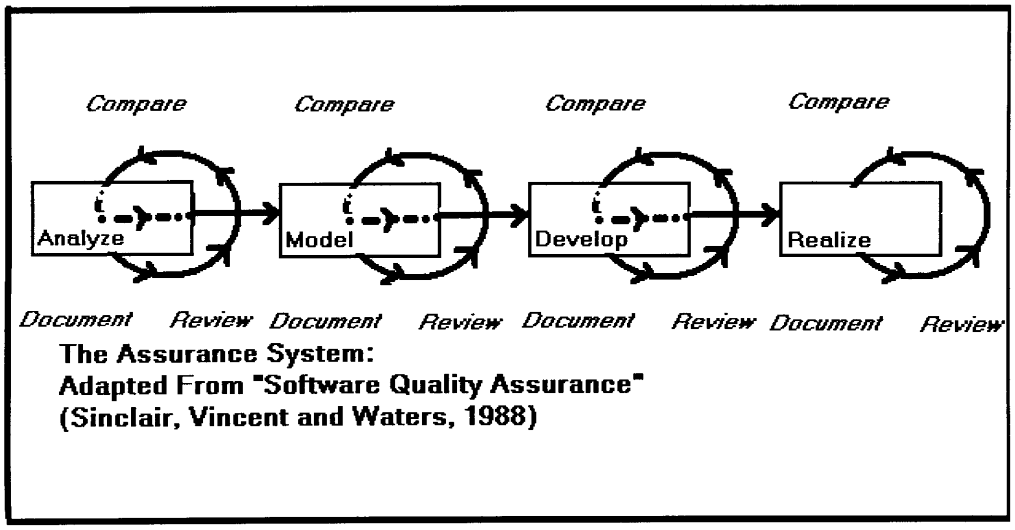


Figure 2-2: The Assurance System

2.2 QUALITY ASSURANCE

A key benefit of quality assurance is the differentiation between quality and quantity. Quality is intangible to most software development teams compared to quantity and this intangibility is one of the problems involved in resisting SQA and not seeing it as a value-adding discipline to the profession. Other hindrances to accepting the SQA function such as programmer's dislike of bureaucracy and paperwork may be countered by developing incentives for employees to accept SQA or developing an SQA plan that is an incentive by itself.

The principal loss attributed to an SQA plan is a perceived loss of productivity to the extent that the plan is unjustified (Sinclair, Vincent and Waters, 1988). This perception results due to the instinctive manner to view productivity from a quantity aspect. However, productivity in software development has to account for the maintenance, testing and debugging which as already noted, consumes most of the resources in a software product life-span. Productivity may be high during development and implementation without an

SQA plan but if the costs of maintenance, testing and debugging are added into the equation, overall productivity falls sharply, usually to a level lower than that realized with an SQA plan.

Some other factors that influence the quantity of code produced, besides the SQA plan, include the quality of programs, the power of the programming language, the programming environment and programmers experience (Sinclair, Vincent and Waters, 1988). In addition, productivity may be increased using editor enhancements (an editor example being Emacs, which was used in the DISEL endeavor), computer sub-second response time, more automation of the software project process, using on-line programming vs. batch programming, languages and program generators (Sinclair, Vincent and Waters, 1988).

2.2.1 QUALITY FACTORS AND ITERATION

Quality factors are the standards applied in the iterative process using the quality unit function and the assurance unit process. These quality factors such as reliability, maintainability and so forth can be decomposed further into attributes known as criteria and that also help describe the quality factor. Each criterion has associated with it one, or several metrics that taken as a whole quantify the criterion. This structure is illustrated in Figure 2-3.

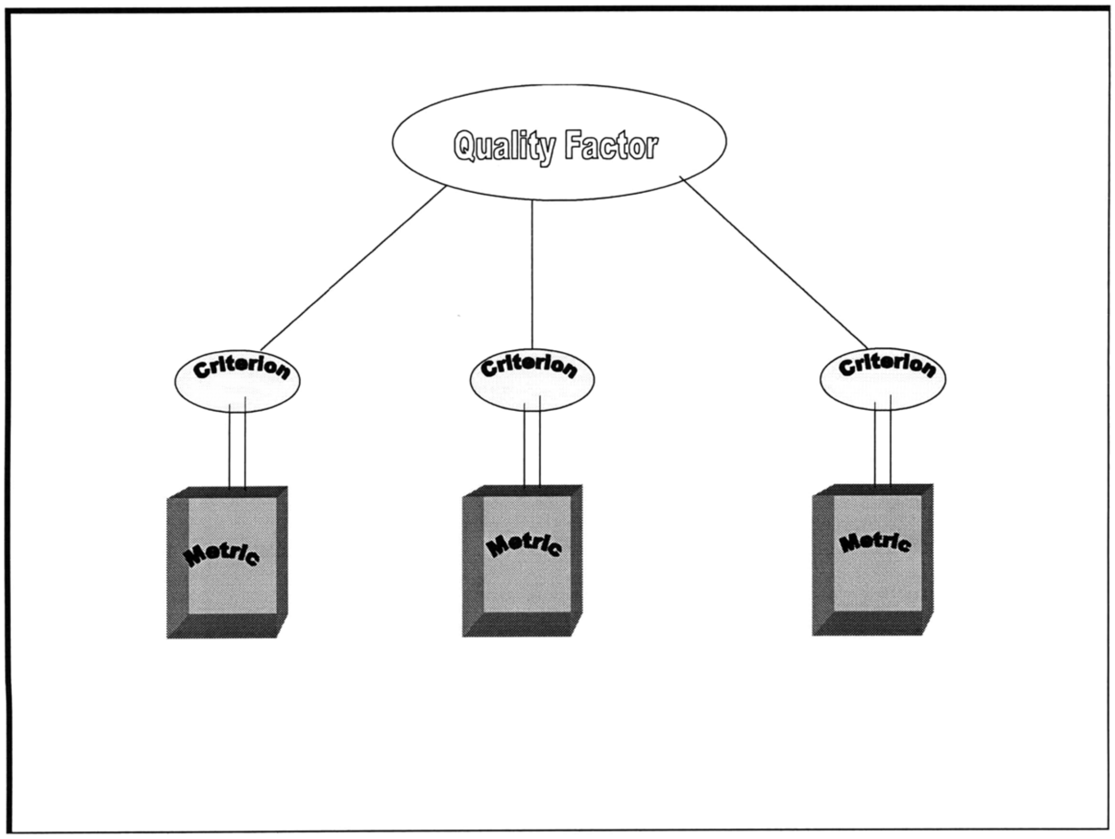


Figure 2-3: Quality Factor Breakdown

2.2.1.1 Software Quality Factors

There are approximately 12-13 accepted Quality Factors in Software Quality Assurance Circles (Evans and Marciniak, 1987). Both Quality Factors and Criteria Tables are developed from material in Evans and Marciniak. Quality Factors are listed in Table 2-1.

Table 2-1: Factors of Software Quality

Quality Factor	User Concern	Definition
Correctness	How well does it satisfy user's needs?	Extent to which program satisfies specifications and user needs.
Efficiency	How well does it utilize resources?	Extent to which computing resources, code and communication time are utilized.
Expandability	How easy is it to expand the capabilities of the system?	Required effort to increase the program functionality either through enhancement of existing functionality or addition of new functionality and data.
Flexibility	How easy is it to change the software system?	Effort required to change the software objective, functionality or other of an existing program.
Integrity	How secure is the software?	Extent to which unauthorized access to software is controlled and controllable.
Interoperability	Is the software versatile and easily interfaced?	Ease of interfacing it with other software systems and programs.
Maintainability	Is the software easy to fix?	Effort required to find and correct an error in an operational program.
Portability	How easy is it to move?	Effort required to transfer the software system from one hardware and/or software environment to another.
Reliability	How dependable is it in producing expected performance?	Extent to which the program performs or is expected to perform its intended function within a specified time period at a given precision.
Reusability	How easy is it to apply the software to another application closely related but other than its originally intended one?	Extent to which the system can be applied to another application that is related in scope to the one the system usually performs.
Testability	How easy is it to prove the system?	Effort required to test a program to ensure performance of its intended function.
Usability	How easy is it to use?	Effort required to get over the learning curve (learn, operate, prepare input, execute actions, and interpret program output).

Quality Factor	User Concern	Definition
Verifiability	How easy is it to confirm system performance?	Effort required to verify specified software operation and performance.

2.2.1.1.1 Criteria

Criteria, from one perspective, are characteristics that define Quality Factors. From an opposite perspective, Quality Factors may be divided into independent characteristics, or Criteria. Table 2-2 lists Quality Factors with associated Criteria.

Table 2-2: Quality Factors and Criteria

Quality Factors:	Criteria:
Correctness	Completeness, Consistency, Traceability
Efficiency	Effectiveness-Processing, Effectiveness-Storage
Expandability	Extensibility, Modularity, Self-descriptiveness, Simplicity; Generality
Flexibility	Modularity, Simplicity, Self-descriptiveness, Generality
Integrity	Access Control, Access audit
Interoperability	Modularity, Commonality; Independence, System compatibility
Maintainability	Modularity, Simplicity, Self-descriptiveness, Document accessibility, Consistency
Portability	Independence, Modularity, Self-descriptiveness
Reliability	Accuracy, Anomaly management, Simplicity
Reusability	Simplicity, Independence, Document Accessibility, Application independence, Self-descriptiveness, System clarity, Modularity, Generality
Verifiability	Modularity, Simplicity, Self-Descriptiveness, Document accessibility, Test
Usability	Operability, Training

2.2.1.1.2 Tradeoffs

Quality Factors can be clearly interdependent, have no relation or the relationship may be unclear. In cases of interdependence the impact between factors may be positive or negative. A matrix may be developed to obtain this relations with human judgment being used to decide on the importance of relative Quality Factors in cases of negative impact. For example, if processing of classified information is of extreme importance and integrity and efficiency impact each other negatively then a human decision may be made to trade-off efficiency for integrity in the software system.

2.2.2 SQA METRICS AND AUDITS

The most visible way to ensure quality in a system is to perform testing activities in an attempt to correct and fix bugs. However, this activity is limited to the final phases of the software project. In contrast SQA, in the form of life-cycle reviews and audit checklists, run the entire length of the product life-cycle, from conception to completion. SQA is therefore more comprehensive for bug prevention then testing.

2.2.2.1 Metrics: The need for a standard terminology

SQA is an integral aspect of the software development cycle. One of the basic problems in assessing SQA impact is that SQA is not as visible as code. This is true since SQA's fundamental units are quality functions and assurance processes, both intangibles. Therefore, standardized terminology in SQA is critical as meanings are not as self evident relative to tangible aspects of software development lifecycle. Metrics are unit measures of quantities and qualities that are used in SQA. Even these metrics need to be precisely defined in order to have the parties involved talking about the same units. Barry W. Boehm and co-authors in their book *Characteristics of Software Quality* define "metrics" as "a measure of the extent or degree to which a product (here we are concentrating on code) possesses and exhibits a certain (quality) characteristic" (Sinclair, Vincent and Waters, 1988).

2.2.2.1.1 Metricizing vs. Metrics: The Concepts

Quality Factors, as stated, are built from Criteria, and these in turn are built from Sub-criteria. Sub-criteria are built from Attributes or Itemized Requirements from which review or audit questions may be asked. These questions are formulated such that either a yes or a no is the answer. The summation of the yes and no values give the values of the Sub-criteria, Criteria or Quality Factor as necessary. Metricizing according to Sinclair and co-authors is the act of defining these Criteria and Sub-criteria of given Quality Factors (i.e. decomposition and weight assignment) and should only be used in this manner. Metrics are then applicable, according to the previous definition, to the process of assigning weights to questions and totaling the scores of the review or audit (Sinclair, Vincent and Waters, 1988).

2.2.2.1.2 Metricizing vs. Metrics: A Process Example

Metricizing:

One Criterion of the quality Factor "Efficiency" is "Execution Efficiency." One Subcriterion of "Execution Efficiency" is "Data Efficiency." An Attribute or Itemized Requirement of "Data Efficiency" is that "data is to be grouped for efficient processing." A review or audit question for the "Data Efficiency" Subcriterion would then be, "Is Data grouped for efficient processing?"

Metrics

If the answer to the question is "yes" a "1" would be placed in the score column of the review or audit sheet; if the answer is "no" a "0" would be placed in the score column. For a non-applicable question the score-holder would have an "X" drawn through it. The total score for all the Criteria/Subcriteria divided by the number of applicable questions will result in the Total Metric Score for the Quality Factor being judged.

Figure 2-4: Metricizing and Metrics: (Sinclair, Vincent and Waters,1988).

The above example clarifies what metricizing and metrics are, their relationship and differences (Sinclair, Vincent and Waters, 1988).

2.2.2.2 Reviews and Audits

The heart of the SQA effort is the reviews and audits. This is the actual implementation to ensure completeness, to the required degree, of the Quality Factors of choice. The nature of reviews and audits is that they are iterative and repetitive. They are iterative when a phase has a low numerical score on required Quality Factors. Then corrections are made and the phase redone with these corrections. They are repetitive in that certain types of reviews and audits may be applied to different phases of the project.

2.2.3 SQA PROGRAM AND SQA PLAN

The software quality program (SQP) consists of all SQA related activities before, during and after the software development program (SDP). This involves the establishment and implementation of requirements, standard practices and procedures to software product, process and quality. The Software quality plan is the section of the SQP that occurs during the SDP.

2.2.4 QUALITY ASSURANCE FUNCTIONS

Quality Assurance is closely related to the functions of Software Configuration Management (SCM), Validation and Verification (V&V), Testing and Evaluation (T&E) and Maintenance (ME). The relationship is a matter of how they are viewed within the quality department of the software development effort. For example, SCM, V&V, T&E and ME may be seen as integral parts of the overall SQA function or they may be viewed as being on the periphery of each other's actions. In reality, neither SQA is complete without the functions of SCM, V&V, T&E and ME nor are these functions complete without the SQA function. An appropriate way to view QA and the relations of V&V,

T&E, SCM and QC is illustrated in figure 2-4.

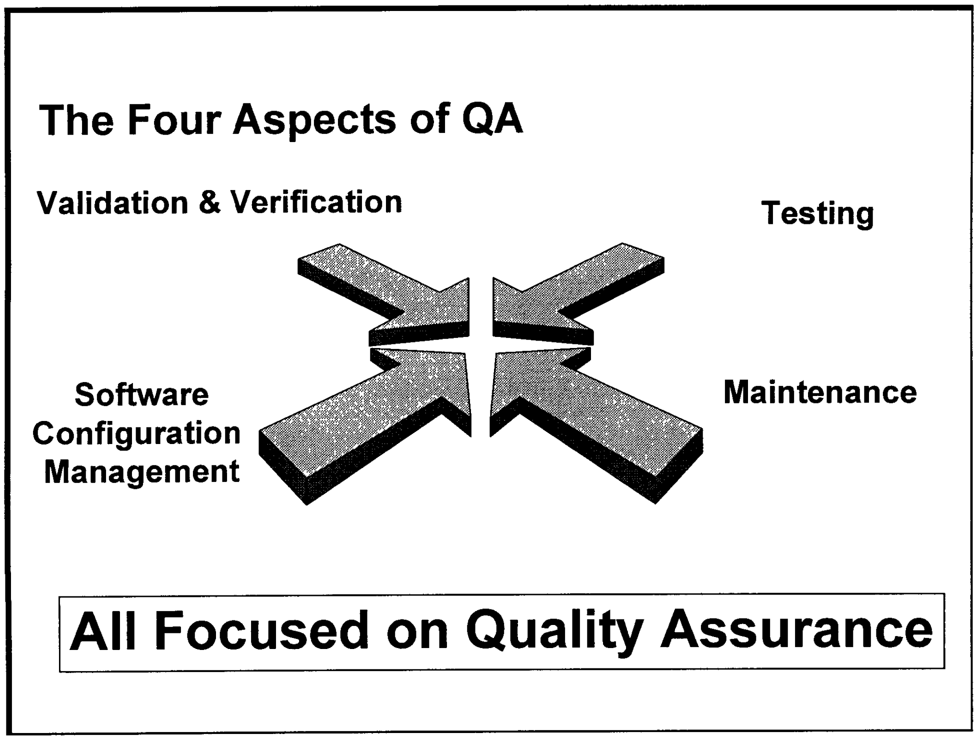

The Four Aspects of QA
Validation & Verification
Testing
Software Configuration Management
Maintenance
All Focused on Quality Assurance

Figure 2-5: The four aspects of Quality Assurance

2.2.4.1 Validation and Verification

Validation is defined as the process involved in evaluation of a specific phase in search of correctness and consistency according to those required by the products and standards that are provided as input. Verification is the process involved in the evaluation of software to ensure conformity with specified requirements. Validation and Verification may either be independent of Software Quality Assurance or dependent. Outside personnel are usually hired to provide this independence. Typical activities for Validation & Verification include monitoring user requests and documentation, ensuring completeness of material submitted, and managing discrepancies between produced documentation and required product and process documentation.

2.2.4.2 Testing and Evaluation

The overall goal of testing is to provide confidence in the correctness of a program. The ideal objective of testing is to provide perfection in resulting output when given all forms of input, perfection being defined as conformance to standards and user requirements. However, such results would require the testing of all inputs which is usually impossible as gains reduce drastically with cost for every additional input tested after a certain point. Good testing establishes and tests up to this point.

2.2.4.3 Software Configuration Management

Software Configuration Management involves the identification, control and assessment of the status of software and software documentation. This involves controlling changes to configuration items and related documentation, recording and reporting the information needed to manage configuration items effectively, and recording and reporting on the status of proposed changes and implementation of approved changes.

2.2.4.4 Maintenance

Maintenance has a different meaning for software products as compared to physical products since software has the peculiarity of maintaining its state to that existing at delivery. Maintenance involves the reworking of the software system to changes in the surroundings. This may include operating systems, expanded user requirements and the need to run comparable applications to those intended for the system initially.

CHAPTER 3
SOFTWARE ENGINEERING STANDARDS AND SOFTWARE ENGINEERING ENVIRONMENTS

As information technology progresses, the amount of software produced has increased and is still increasing raising the importance of the role of quality management of software products. Establishment of such a quality management system is justified by the need to provide guidance for software quality assurance (American National Standard, 1995). These standards are established to ensure the desirable objectives of a software process, namely, optimality and scalability (Jalote, 1997). To achieve these objectives, a process should have the following properties: predictability, support testability and maintainability, enable early defect removal and defect prevention, and allow process improvement (Jalote, 1997).

3.1 MOTIVATION

The requirements for a generic quality system in a two-party contractual situation have already been published. These requirements are "ANSI/ISO/ASQC Q9001, Quality Systems - Model for Quality Assurance in Design, Development, Production, Installation, and Servicing". These requirements are for two-parties agreeing under contract to have one party develop software for the other for some given compensation. But the processes, nature, development and maintenance of software differ from those of most industrial goods. Differences include documentation where in traditional industrial products the documentation accompanied the product development while in software, documentation is the product.

Therefore, in such a fast-track technology, it becomes necessary to provide extra guidance for quality systems that take into account these differences involving software products as well as the present level of information technology (American National Standard, 1995). These guidelines are then used within the two main quality assurance approaches (product- and process-based) illustrated further on, along with derived models such as the waterfall or spiral model, illustrated in Chapter 4, to deliver a quality system.

3.2 ISO 9000-3 : SOFTWARE ENGINEERING STANDARDS

Some of the properties of software govern that some activities relate to particular phases of the development process while other properties require activities to apply throughout the process hence ANSI/ISO/ASQC Q9000-3-1991 has been formulated and developed to reflect these differences of phase-specific activities and project-wide activities (American National Standard, 1995).

However an additional point to note is that when two parties agree to contracts on software product development, the contracts may have different forms and these guidelines may not be applicable even if "tailored". Therefore, determination of the adequacy of ANSI/ISO/ASQC Q9000-3-1991 application to the contract is important. Accordingly, ANSI/ISO/ASQC Q9000-3-1991 deals primarily with situations where specific software is developed as part of a contract according to purchaser's specifications. However, the concepts described may be equally of value in other situations (American National Standard, 1995).

3.2 SCOPE OF ISO9000-3

ANSI/ISO/ASQC Q9000-3-1991 (henceforth the 9000 series is referred to as ISO Q900x-x with the final extension assigned accordingly) develops guidelines that facilitate ISO Q9001 application to organizations developing, supplying and maintaining software. The purpose is to provide guidance when a contract between two parties requires proof of the supplier's ability to develop, supply, and maintain software products.

ISO Q9000-3 guidelines are intended to provide guidelines to meet a purchaser's requirements. These guidelines are suitable for use in agreements between two parties where design effort is necessary or performance terms are used to outline product requirements (American National Standard, 1995). In addition, the guidelines are suitable if the supplier has enough of a quality history to ensure or at least qualify that his product satisfies the purchaser (American National Standard, 1995).

ISO Q9000-3 uses the following core definitions in developing its guidelines amongst other functions.

Table 3-1: ISO Definitions.

Software: the abstract constructs that are embodied in a system and consist of programs, rules, help, algorithms and associated documentation.
Software product: the set of items delivered to the user/purchaser and consisting of the programs, rules of use and documentation.
Software item: A perceivable part of the software product at the end of the development or at a transitional point.
Development: The actions that are to be executed in the creation of the software product.
Phase: A work segment that is defined.
Verification (for software): The process involved in evaluation of the products of a specific phase in search of correctness and consistency according to those required by the products and standards that are provided as input.
Validation (for software): The process involved in evaluation of software to ensure conformity with specified requirements.

The ISO Q9000-3 guidelines outline a framework for a quality system consisting of management responsibility and the supplier's quality system. Management responsibility is threefold; supplier's side management responsibility, purchaser's side management responsibility and joint reviews. The supplier side management responsibilities include quality policy, organization (involving responsibility and authority, verification resources and personnel, and management representative) and management review. The purchaser's side management responsibilities include, but are not limited to, defining their needs to the supplier, answering questions from the supplier, approving the suppliers proposals, concluding agreements with the supplier, ensuring the purchaser's organization observes the agreements made with the supplier, defining acceptance criteria and procedures, and dealing with the purchaser-supplied software items that are found unsuitable for use.

Both the supplier and purchaser in a continuation of quality assurance are brought together during the joint reviews where the following tasks are performed as appropriate: conformance of the software to the purchaser's agreed-to requirements specification, verification results, and acceptance test results. The results of such reviews are then agreed upon and documented.

According to ISO Q9000-3 guidelines the supplier's quality system should document the system. Furthermore, according to these guidelines the whole quality system, all throughout the development of the software, needs to be integrated as a process hence making sure that quality is built as system development progresses. The emphasis should be on preventing problems and not on curing problems. The responsibility of ensuring effective implementation of the documented quality system belongs to the supplier (American National Standard, 1995).

3.3 PRODUCT CENTERED SOFTWARE ENGINEERING ENVIRONMENT

Product centered software engineering environment achieve quality by evaluation of products, or the final product achieved at the end of each phase. A primitive method, it is usually self-evident to software development teams that the next step necessary to take to ensure a quality product is optimization for quality of intermediate and final software products by looking at the processes involved in realizing these products. The fundamental relationship between the process model, process and the final product that results is illustrated in figure 3-1 (Garg and Jazayeri, 1996).

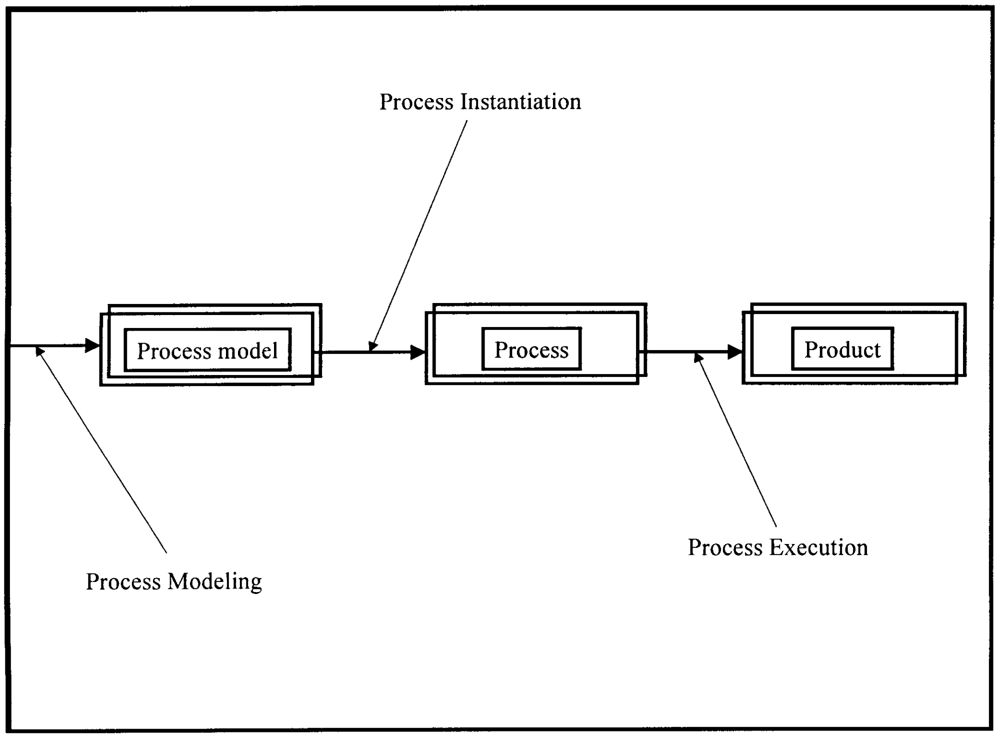


Figure 3-1: Process Model, Process, Product and Relating Actions

3.4 PROCESS CENTERED SOFTWARE ENGINEERING ENVIRONMENT

3.4.1 Overview

A process is a "partially ordered set of activities to achieve a goal" (Garg and Jazayeri, 1996). A software development process is a set of activities with the goal of producing a software product. A process model is a representation of a process. Key components of a process model are activities to be performed, agents performing the activities, products produced and the resources needed to execute the activity. Process-centered engineering looks at the process used in creation of the software product. The result is a focus on the people involved and their interaction. The core of process-centered engineering is the delimiting and modeling of these processes followed by mapping them onto the software development environment (Garg and Jazayeri, 1996).

The main difference between product- and process-centered engineering environments (PCSEE) is that PCSEE support the managerial and quality assurance functions in addition to the development function that is supported by both types of software engineering. The principal problem with a process centered approach is that, unlike a product which is visible and easier to agree upon, different processes may be used to achieve the same value with the value of each process being difficult to agree upon. The principal advantage is that PCEE's define the processes across functional boundaries based on the product, allowing the software environment to support QA naturally in assessing and delimiting a process.

3.4.2 Processes and Process Models

While a process model may be used to understand and communicate the process, it may also be used to analyze, improve and manage the process itself, being discharged if found lacking in any of these three operations. Formal, complete models are easier and more realistic for implementation in addition to allowing easier automation. Formality and completeness indicate that models have levels of abstraction. These levels may be at a high level such as the prototyping model illustrated in Chapter 4, though detailed models are more common and useful for development.

Because software development is an intellectual and creative activity, the software process cannot be static. The software process not only needs to be adapted to its environment but also change with environmental changes. In addition, some PCSEE's support dynamic modification of process definition as the process occurs. "The study of the software process as a dynamic entity, with its associated activities of modeling, definition, analysis, simulation and so on, has identified a new function in a software development organization called 'process engineering.' An extension of the traditional function of quality assurance, process engineering treats software processes in a systematic way with a well-defined life cycle. An important function of PCSEE is the support of process engineering" (Garg and Jazayeri, 1996). A process-centered system whose architecture is decentralized and geographically distributed is OZ created in 1992 (Garg and Jazayeri, 1996).

CHAPTER 4
SOFTWARE DEVELOPMENT MODELS AND SOFTWARE QUALITY ASSURANCE

The effectiveness of a given Software Quality Assurance plan is principally dependent on its suitability for the software development process and the particular model in use.

4.1 SOFTWARE DEVELOPMENT MODELS

Some typical models and their application from a software quality assurance perspective include the waterfall model, prototyping, iterative enhancement and the spiral model. In addition, a model with a focus on software quality assurance and based on process-centered software engineering is illustrated.

4.1.1 A PROCESS STEP SPECIFICATION

Process step specification addresses the process model property of step sequences with respect to their initiation and termination, amongst other issues related to the step property (Jalote, 1997). Since defect detection is one aim of a process, verification and validation (V&V) is necessary at the end of each step. Also termination of each step in a work product necessitates a small number of steps to reduce the cost of V&V. As noted, specifications are determined by when a step should be initiated and when it should be terminated, depending on project implementation. All these issues are fundamental to designing the common development process models described below.

4.1.2 WATERFALL MODEL

The waterfall model treats the software development lifecycle as a linear process with each phase following into the next from a time perspective. The waterfall model is illustrative of the types of activities that occur in software development. It is an appropriate model for illustrating the basic software engineering phases and work products but is deficient in application. The principal problem with the waterfall model is the conflict between the need to adequately analyze work products before reviews and the pressure to continue the overall software development effort without a loss of wholeness (Evans and Marciniak, 1987). Below is a schematic of the waterfall model. The decreasing space between each phase is representative of the time compression that occurs towards the end of a project.

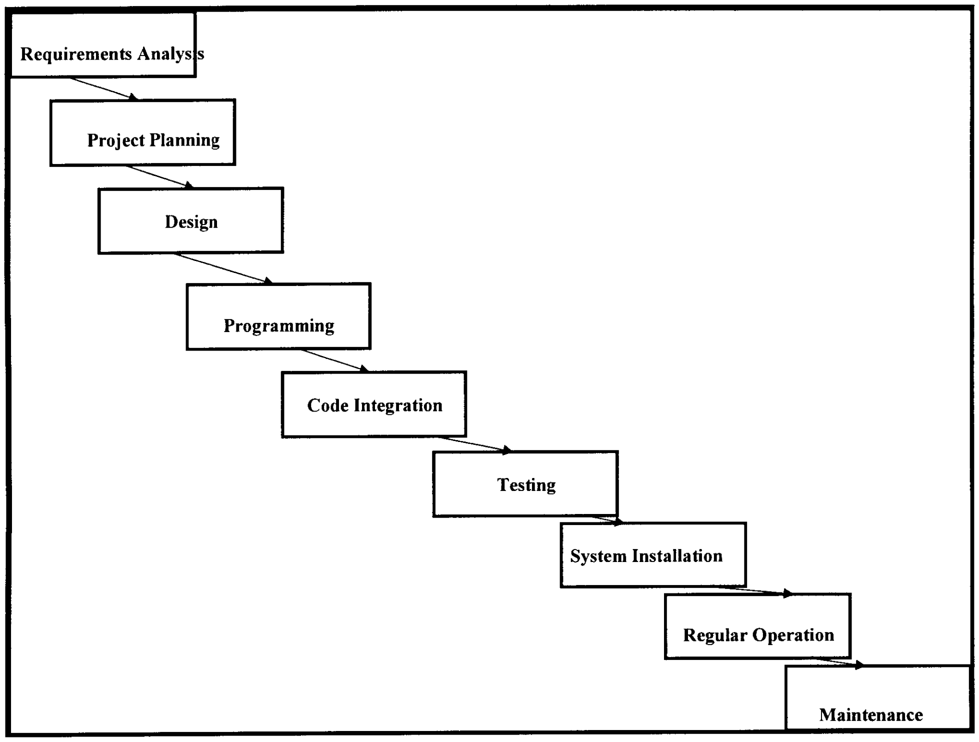


Figure 4-1: The Waterfall Model

Before major reviews occur, the documentation requiring reviewing may be bulky and complex due to the intertwining relationship between different forms of documentation such as requirement analysis, design, code and testing. What occurs is that data is submitted for review and the project proceeds. Therefore, since the results of a review ends up being submitted after the project has proceeded, changing the status, then the results of the formal review are no longer valid.

Since the software project is extremely dynamic, changes cannot be factored in the review process because of schedule pressures or project limitations. It is difficult to accurately assess project progress and software quality. What usually happens is that when technical data documentation, planning documentation and other documentation is submitted for review there is not enough time to sufficiently evaluate all documentation. Instead what occurs is sampling of the documentation with resulting approval to documents that should not be approved, critical relationships are not captured, risk is ignored and suggestions given are not optimal. The aim becomes keeping the documentation on schedule instead of assessing the integrity of the project.

4.1.3 PROTOTYPING

A prototyping development model facilitates rapid development of a system that reflects the projected operational capabilities of the system before the software product is developed. The prototype system serves as a test platform on which system and user interface concepts can be tested prior to development.

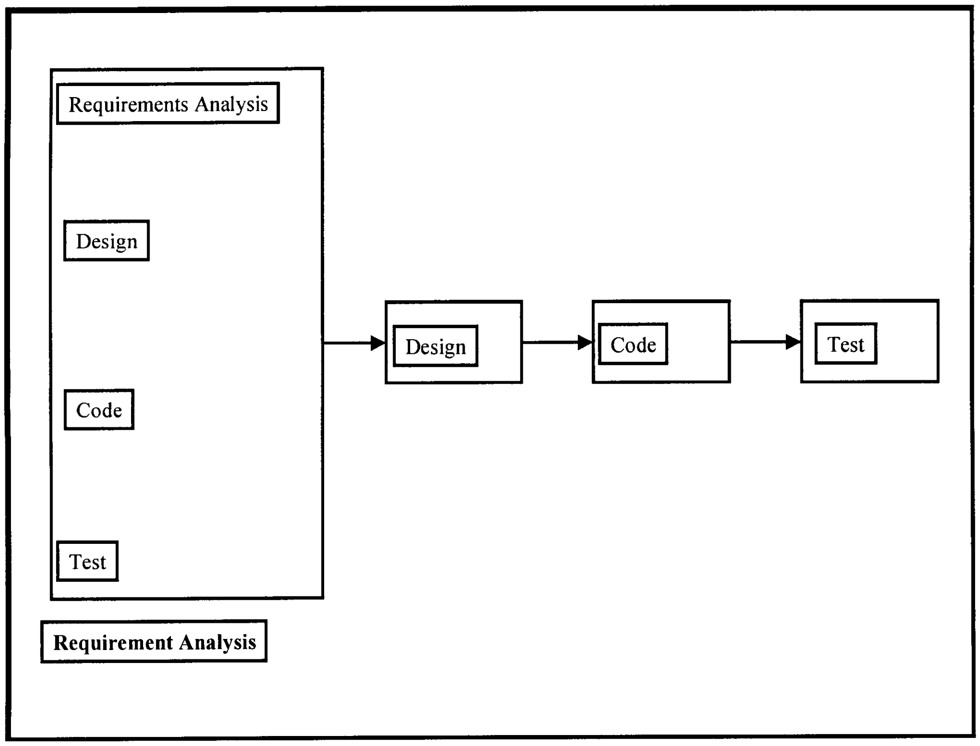


Figure 4-2: The Prototyping Model

Product assurance within a rapid prototyping development environment would involve activities that are less internal (project members) and more external (client interaction) in attempting to realize the client requirements aspect of the software product quality. In addition, satisfaction of standards and procedures, such as ISO guidelines in use, are easier to determine due to the visibility achieved by developing an early prototype.

4.1.4 ITERATIVE ENHANCEMENT

The iterative enhancement model involves software development in increments where a certain amount of functionality is added to the system per iteration until the complete system is implemented. It combines the linearity of the waterfall model that is necessary for project completion with the prototype model benefits of having a visible product that can be tested.

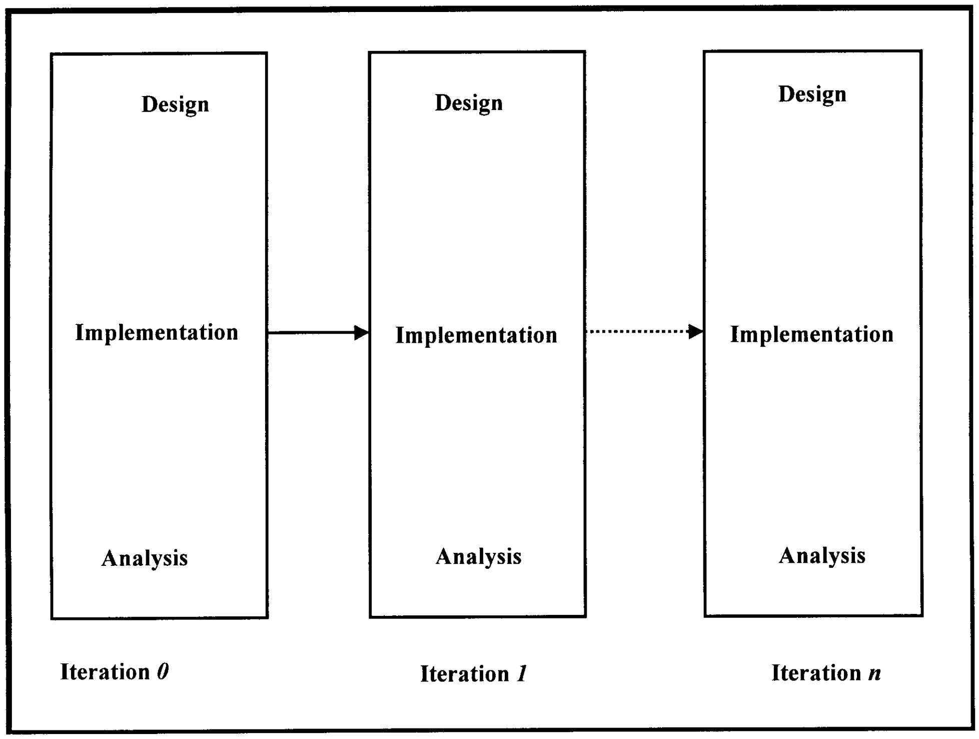


Figure 4-3: Iterative Enhancement Model

A simple implementation is initially done for a sub-set of the problem space. This sub-set contains a key aspect of the requirements easy to implement, involving a build up to the final system. Each of these iterations involves addition of key tasks to the incremental build system. These key tasks are contained in a list and realized in order of implementation difficulty. Each task consists of removing the next task from the list, designing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. Quality assurance as an iterative function fits well into this model.

4.1.5 THE SPIRAL MODEL

Project activities that use this model can be arranged in a spiral manner will many cycles. Each cycle in the spiral model starts with objective identification, alternatives and constraints. Evaluation of these items follow with a focus on risk/benefit analysis. Then strategies are developed, leading to development of the software and the start of planning the next cycle of the spiral.

4.2 MODEL APPLICATION OF PROCESS CENTERED SOFTWARE ENGINEERING

Figure 4-4 models the components of a typical process-centered software engineering environment. The principal components of this architecture are a data repository to store data, code, process and product documentation; software tools that help with the processes; some sort of communication system; and process engines that are support by the communication infrastructure. This process architecture, especially the data repository and communications subsystem, were realized in the DISEL endeavor. The data repository was used for the process of document creation, storage and manipulation such as schedules, agendas, minutes and the communications sub-system included e-mail for the process of communication.

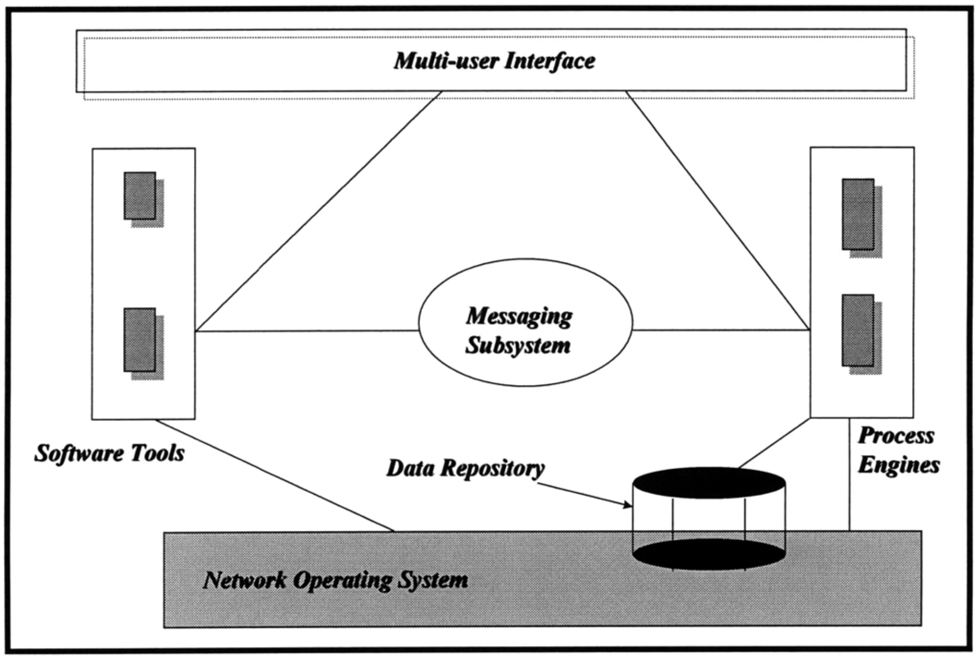


Figure 4-4: Process-centered Software Engineering Environment Schematic.

CHAPTER 5
1.120 DISEL PROJECT: QUALITY ASSURANCE PLANS AND THEIR IMPLEMENTATION

The diagram and definitions in figure 5-1 illustrates the software development life-cycle, the quality assurance functions of Software Configuration Management, Validation and Verification and Testing and Evaluation, and the scope of SQA (Sinclair, Vincent and Waters, 1988).

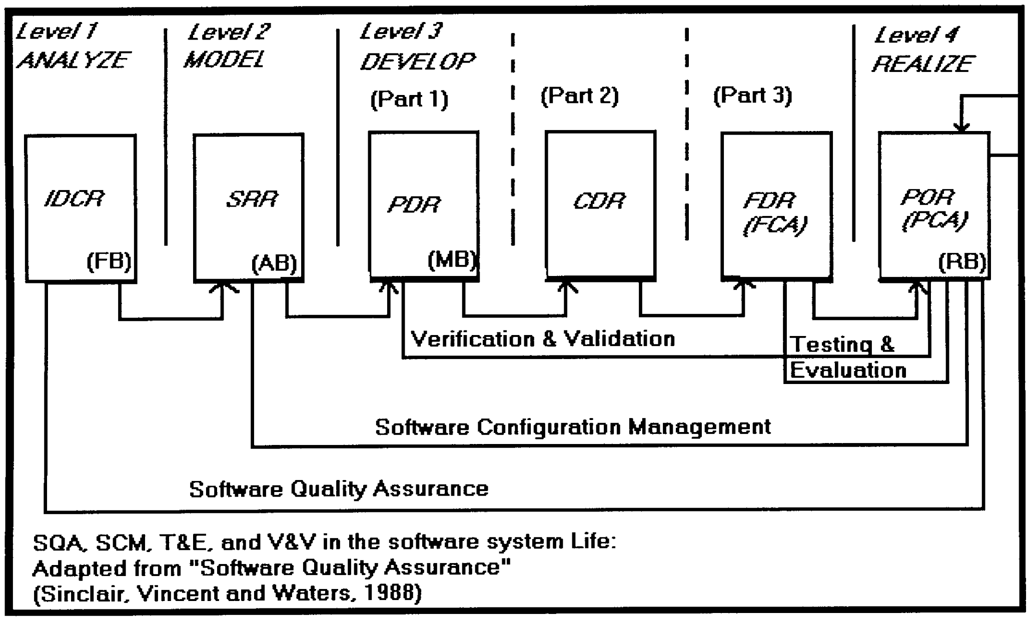

Level 1
ANALYZE
Level 2
MODEL
Level 3
DEVELOP
(Part 1)
(Part 2)
(Part 3)
Level 4
REALIZE
IDCR
(FB)
SRR
(AB)
PDR
(MB)
CDR
FDR
(FCA)
POR
(PCA)
(RB)
Verification & Validation
Testing &
Evaluation
Software Configuration Management
Software Quality Assurance
SQA, SCM, T&E, and V&V in the software system Life:
Adapted from "Software Quality Assurance"
(Sinclair, Vincent and Waters, 1988)

Figure 5-1: SQA, SCM, T&E, and V&V in the Software System Lifecycle

Table 5-1: Legend for Figure 5-1: SQA, SCM, T&E, and V&V in the Software System Lifecycle

Legend:
IDCR: Initial Design Concept Review (Requirements Analysis Phase)
SRR: Software Requirements Review (Requirements Specification Phase).
PDR: Preliminary Design Review
CDR: Critical Design Review
FDR: Final Development Review
FCA: Functional Configuration Audit
POR: Post-Operation Review
PCA: Physical Configuration Audit
FB: Functional Baseline
AB: Allocated Baseline
DB: Design Baseline
PB: Product Baseline
OB: Operational Baseline

Quality assurance in software creation includes the planned and systematic actions developed to establish that a given software product or system developed meets the user's requirements. Hence since meeting user requirements can be better achieved by optimizing these planned actions, then quality assurance involves looking at the process in addition to monitoring the product quality. A third aspect of Quality Assurance that has been neglected until recently and may be the most important aspect in realization of a collaborative development project is the people involved and the team dynamics that occur.

The purpose of the DISEL effort in broad terms was to re-create the "campus experience" in distance learning situations and allow individuals to express themselves without the constraints of the machine environment. The objectives were to enable Casual Contact and improve Social Interaction through collaborative awareness, personal expression and social feedback (http://kiliwa.cicese.mx/~disel/Documents/Documentation.html). The result was the Cliq! system which delivered on each of these three objectives. The DISEL effort was conducted within the 1.120 Information Technology project. The development model that was implicitly applied was the Waterfall model.

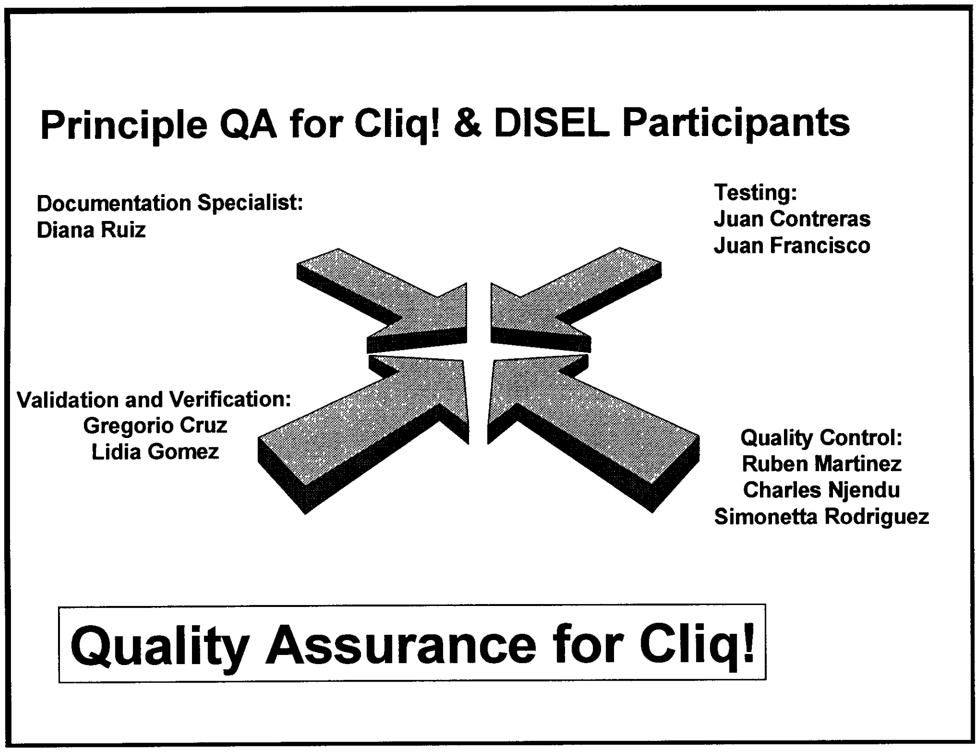


Figure 5-2: Quality Assurance for Cliq!

The quality assurance sub-teams were quality control, software configuration management, validation and verification, testing, documentation and maintenance. The quality assurance relationship has been reproduced above as it was structured within the DISEL project, with the four principle active roles. Documentation played the key role in ensuring compliance with any of the aspects of SQA. Compared to the generic relationship established in Chapter 2, quality control in the DISEL replaced positions with maintenance in activities performed towards SQA. Whatever occurred within the scope of maintenance was implicitly merged with testing. Software Configuration Management within definitions also occurred.

5.1 QUALITY ASSURANCE PLANS DESIGNED IN THE 1.120 PROJECT

The following is an overview of the quality assurance plans developed by the team members of the DISEL endeavor. These plans consisted of the quality control plan, the software configuration management plan, the testing plan, the validation and verification plan, the maintenance plan, and the documentation specialist plan.

5.1.1 QUALITY CONTROL PLAN

The purpose of the quality control (QC) plan was to improve software quality by monitoring both the software product and the development process that produced Cliq!. The software product included the code, design documentation, test plans and user manual. In addition, the plan was to ensure compliance with established standards and procedures for both the product and process that had been tried and proven. Finally, the quality control plan was to ensure that any inadequacies in the product, the process or the standards were brought to the attention of the team to ensure that they are corrected.

The following were the benefits that the DISEL effort pursued in creating a quality plan that captured these purposes. An appropriate development methodology in achieving client requirements would be used. As the project progressed, project members would have standards and procedures to direct their work. Independent reviews and audits necessary to ensure a quality system would be conducted. Documentation would be produced to support maintenance and enhancement of the system and process. Most importantly, this and other documentation being produced during and not after the system development. In addition, if changes were necessary, there would be mechanisms to ensure a smooth transition and project progression. Also, since risk within the process was not even, nor was it expected to be, testing would be developed with an emphasis on the high-risk product areas. Finally, each software task such as the development, and the finer sub-tasks, would be completed to requirement satisfaction before the next tasks begun, with deviations from agreed upon standards and procedures being exposed as soon as possible and external professionals being able to audit the project as necessary.

The following was the schedule of the actions planned, documentation involved and producers of documentation for the stated quality control objectives in realization of the Cliq! system.

Table 5-2: Quality Control Plan

Date:	Action Planned:	Documentation Involved:	Producer of the Document:
Sept: 04 – 30th	Definition of the Quality Control Plan	Quality Control Plan	Quality Control Engineers
Sept 18th	Participation as witnesses in the client interview that is used to obtain requirements.	None.	Client
Oct 2nd	Presentation of the Quality Control Plan within the laboratory.	Quality Control Plan	QC engineers
Oct 9th	QC participates in the review of the Captured Requirements using the walkthrough technique within the laboratory class	Requirements Specification	Analysts
Oct 14th	Requirements Specification Audit	Requirements Specification	Analysts
Oct 14th	Project Plan Audit	Project Plan	Project Managers
Oct 16th	Requirement Analysis Audit Presentation in the laboratory class	Requirement Specification Review	QC Engineers
Oct 20th	Documentation Specialist work plan audit	Documentation Specialist work plan	Documentation Specialist
Oct 28th	Requirement Analysis audit	Requirement Analysis	Analysts
Oct 30th	Presentation of the Requirement Analysis audit at the laboratory	Requirement Analysis Audit	QC Engineers
Nov 3rd	Requirement Analysis review as preparation for the technical review scheduled for November 6th	Requirement Analysis	Analysts
Nov 6th	Participation as reviewers in the group review of the requirement analysis within the laboratory class	Requirement Analysis	Analysts

Date:	Action Planned:	Documentation Involved:	Producer of the Document:
Nov 13th	Participation as reviewers in a review of the requirements analysis at the laboratory class	Requirements Analysis	Analysts
Nov 17th	Audit to the Software Configuration Management Plan	Software Configuration Management Plan	Software Configuration Manager
Nov 18th	Requirements Analysis audit	Requirement Analysis	Analysts
Nov 20th	Presentation of audit performed to the requirement analysis at the laboratory	Requirement Analysis audit	QC Engineers
Nov 20th	Presentation of audit performed to the software configuration management work plan in the laboratory	Software Configuration Management work plan audit	QC Engineers
Nov 28th	Test Plan audit	Test Engineers plan	Test Engineers
Nov 29th	Requirement Analysis audit	Requirement Analysis	Analysts
Dec 2nd	Presentation of the Requirement Analysis audit at the laboratory class	Requirement Analysis audit	QC Engineers
Dec 2nd	Presentation of the Test Engineers plan audit at the laboratory class	Test Engineers plan audit	QC Engineers
Jan 4th	Maintenance Engineer work plan audit	Maintenance Engineer work plan	Maintenance Engineer
Jan 9th	Preliminary Design Audit	Preliminary Design	Designers
Jan 15th	Participation as reviewers in the Preliminary Design review at the laboratory class. A walkthrough review.	Preliminary Design and Preliminary Design review	Designers and QC Engineers
Jan 29th	Preliminary Design audit	Preliminary Design	Designers
Feb 5th	Presentation of the review results made to the preliminary design at the	Preliminary Design audit	QC Engineers

Date:	Action Planned:	Documentation Involved:	Producer of the Document:
	laboratory class		
Feb 8th	Detailed Design preparation review	Detailed Design	Designers
Feb 12th	Participation as reviewers in the Detailed Design review at the laboratory class. This review will be in the form of inspection	Detailed Design and Detailed Design preparation review	Designers and QC Engineers
Feb 19th	Participation as witnesses in the code technical review at the laboratory class. This review will be in the form of a peer review	Source code	Programmers
Mar 3rd	Audit to the changes recorded by the Software Configuration Manager	Changes record	Software Configuration Manager
Mar 12th	Participation as witness in the code technical review at the laboratory class. A peer review	Source code	Programmers
Apr 7th	Audit to the tests applied to the code modules. The Test Engineer must supply a document with the results.	Test results	Test Engineers
Apr 9th	Presentation of the audits made to the code modules tests at the laboratory class	Test results audit	QC Engineers
Apr 30th	Final audits to all deliverable documents: the final user manual, the software configuration document, and the system and acceptance test results	User manual, Software Configuration documentation and Acceptance Test results	Documentation Specialist, Software Configuration Manager and Test Engineers
May 7th	Presentation of final audits results and the trend analysis at the laboratory class	Final audits and Trend Analysis	QC Engineers

5.1.2 SOFTWARE CONFIGURATION MANAGEMENT PLAN

Software configuration management is traditionally applied to the development of hardware systems, or to the development of hardware elements of hardware-software systems tailored to a system, or portion of a system, predominantly comprising software. Software configuration management is the discipline of applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of software configuration items (SCI's) (Buckley, 1992).

In developing Cliq! software configuration management (SCM) was expected to include the following tasks. Software configuration management items were to be audited to verify conformance to specifications, interface control documents, and other contract requirements. In addition, SCM was to control changes to configuration items and their related documentation, recording and reporting information needed to manage configuration items effectively; including the status of proposed changes and the implementation status of approved changes (Buckley, 1992).

The software configuration management plan was developed to identify and define the organization, activities, overall tasks, principles, and objectives of SCM. The policy statement used in creation of the plan included keeping the number of configuration changes to a minimum, and any suggested improvements not required to meet requirements were to be approved and authorized only if they could be justified on a cost-savings and/or effectiveness ground.

The purpose of the software configuration plan was to identify and describe the overall policies and methods for SCM to be used during the software life cycle. The full purpose, scope and the glossary of terms for SCM are included in Appendix A.

SCM activities were to include Software Configuration Identification (which involved the identification of Configuration Items and the naming of Software Configuration Items), Software Version Control, Software Configuration Control, Software Configuration Status

Accounting, and Software Configuration audits and reviews.

Identification of Software Configuration Items (SCI's) was to be used in each of the documents, or set of documents, with a formally designated, fixed specific time in the software lifecycle. These documents and their associated times were to form baselines as the project progressed. A series of different baselines to be used were established permitting an ordered flow of development work.

Table 5-3: Development Baselines

Requirements baseline	Specification baseline	Design baseline
Unit baseline	Integration baseline	Operational baseline

Marcela Rodriguez gives a concise description of these roles and their interaction in the Software Configuration Management Plan she developed for the Cliq! development effort. In it she states that "The *Requirements baseline* is established when the requirements are completed and initially approved. The *Specification baseline* includes the program external specifications together with a cross-reference to the requirements and operational concept. The *Design baselines* are established when the design is initially completed and inspected. As each unit is completed, inspected, and tested a *Unit baseline* is established. After initial implementation and unit test, the programs are placed in the *Integration baseline.* The *Operational baseline* is established at the time of system shipment." (Urreas, 1997).

5.1.3 TESTING PLAN

The development of any software system, including Cliq!, involves a series of production activities with the possibilities of human error just like any other sequence of production. The complexity, changeability and invisibility of software magnify the number of possible errors and their type. Testing is extremely difficult as the essence of software is conceptual, abstract, highly precise and richly detailed (Brooks, 1995). Errors tend to show up from the first moment of the process (because from this stage on the objectives can be specified in erroneous or imperfect form). These errors typically magnify in the subsequent stages of design and development. In summary, it is impossible for humans to work and communicate in perfect form. This imperfection requires the development of software accompanied by an activity that attempts to guarantee the system quality, the activity of testing (Benjamin and Garcilazo, 1997). Testing, put simply, is looking at the code elegance, robustness and consistence.

The testing plan purpose was providing confidence, robustness and consistency in Cliq!. It is impossible to test for all possible scenarios even if known. Therefore the scope of the test plan was intended to ensure that Cliq! satisfied the requirements given by the client, captured in the design documentation and detailed in the product description, standards and procedures of the quality control plan in appendix A. The main objective of the plan was to implement a test that exposed different error classes, in minimal time and manpower without the loss of efficiency, and in a preventive and not curative manner (Castillo and Ortiz, 1998).

The separate components used in the development of the testing methodology are listed in table 5-4.

Table 5-4: Components of the Testing Methodology

The program to be tested.
The environment in which the testing occurred.
The methods used to obtain test cases.
The methods used for performing test cases and evaluating results.
The methods for assessing program quality based on the results.

Test team activities included participation in the revision of the analysis document to get familiar with user requirements and design methodology. The purpose of the participation was to enable the test team to design tests that captured the functionality, performance and a good application of the methodology (Castillo and Ortiz, 1998).

Testing was divided into 3 levels: unit, integration and systems testing (Castillo and Ortiz, 1998). The unit level was easiest to define, structure and achieve a complete test criteria for. Unit testing is also known as structural testing.

However for a software product, the sum is rarely the whole of its parts meaning that unit tests cannot provide a guide to the overall system behavior. Therefore after unit testing was performed integration testing occurred which involved working at different levels of abstraction leading to the final system testing. The main problem with the intermediate system testing was that unlike a unit (such as the User Interface) whose behavior is usually specified and the whole Cliq! system whose behavior must always be specified and was specified in the requirements and design phases, integration behavior was unknown. This made it impossible to use functional testing at the integration level.

Since unit testing could not give the behavior of the system as a whole and integration testing was structural as well, system testing of Cliq! was the final step that helped eliminate the variant behaviors that came from software not being the sum of its parts (Castillo and Ortiz, 1998). The test documentation fields that resulted are listed in table 5-5.

Table 5-5: Test Documentation Fields

Test identifier	Version of the test	Name of the test
Level of the test	Modules to be tested	Test Objectives

Test date	Required time	Inputs
Expected outputs	Results	Comments
Conclusions		

This information allowed the test engineers to obtain a history of all tests applied to the system as well as a database of errors found in the design and programming stages (Castillo and Ortiz, 1998).

In addition a bug-tracking system or database was developed and included the fields listed in table 5-6.

Table 5-6: Bug-Tracking System Information Fields

Date	Stage [Design or Programming]	Type of error
Syntax	Incongruent with respect to previous stage	Class
Relation	Function	Attribute
Other (Specify)	Level of error (High, Medium and Low)	Section of the design document or code where the error was found (row, page or diagram).

The documents that the test plan generated in achieving its purpose were:

- A test plan

- The results of tests applied to modules and methods
- The results of integration tests
- The results of usability tests
- The results of regression testing

5.1.4 VALIDATION AND VERIFICATION PLAN

Validation, as defined in Chapter 2, refers to the process of evaluating software at the end of its development to ensure that it is free from failures and complies with its requirements. A failure is defined as incorrect product behavior. Verification refers to the process of determining whether or not the products of a given phase of a software development process fulfill the requirements established during the previous phase (Cruz and Gomez, 1997). These definitions cover the traditional look at the production of a software system: the product and the process.

Within the DISEL effort software verification and validation helped determine that the software requirements were implemented correctly and completely and were traceable to the Cliq! system requirements. V&V evaluated how well the software was meeting its technical requirements and its safety, security, and reliability objectives relative to the system. It also ensured that software requirements were not in conflict with any standards or requirements applicable to other system components. Software V&V was typically supposed to analyze, review, demonstrate or test all software development outputs.

The responsibilities of V&V were to ensure that Cliq! was free from failures and meet its user's expectations. There were several theoretical and practical limitations that applied to Cliq! or any software system, that made these responsibilities impossible to obtain. These include the impracticality of testing all data, the impracticality of testing all paths and the lack of a rigorous mathematical proof of correctness that could be applied to software engineering (Cruz and Gomez, 1997). In delivery of the Cliq! system, V&V set out to analyze and test the software with the focus being delivery of functionality at the level of personal expressions and social feedback, and to assess how well the software realized the issues of reliability, security, and safety.

The Validation and Verification (V&V) plan developed for the Cliq! system had to satisfy these client requirements at the functional level and in its performance. The particular objectives of the plan created to ensure this included:

- *Correctness:* Involved making sure that the final product is free or errors.
- *Consistency:* Involved making sure that the product is consistent with itself and with the other products. As an example, the definition of a function key should be the same one for all interfaces (F1 = help).
- *Necessity:* The system was to have all that is necessary in order to operate in accordance with the requirements of the client. That is, more capacities to the system then the ones needed should not be added.
- *Sufficiency:* Involved making sure that the system is complete.
- *Performance:* Involved making sure that the system satisfied client requirements.

The key activities through which the validation and verification added value in the DISEL endeavor were:

- Creation of a Software Verification and Validation Plan
- Creation of a System Acceptance Plan
- Creation of the final Software Verification and Validation Report
- Creation of the Agenda for the Laboratories where Cliq! was created, and
- Work as Moderators of the Activities performed in the Laboratory Sessions.

In short, validation and verification in the development of Cliq! asked whether the functionality of Cliq! was meet and attempted to answer this question.

5.1.5 MAINTENANCE PLAN

The role of maintenance, as stated earlier on, never really occurred in the development of Cliq! and was integrated within the testing function. However, the objectives of the maintenance plan, as developed, were to ensure that the development team was updated on errors detected and ensure system flexibility. The mission, objectives and work plan are detailed in appendix A.

5.1.6 DOCUMENTATION SPECIALIST PLAN

Documentation is one of the most important aspects of quality assurance since it is the most visible and durable result of the software production. Documentation is essential both to the creation, use and maintenance of the system. Within the development of the DISEL software development process a great amount of documentation was generated including requirements analysis documentation, design documents and source code. This documentation was stored in order to have a development process history and in a manner that was user-friendly. The main objective of such documentation was to be a communication, reference and idea generation medium between the several members of the DISEL development team, especially in the distributed endeavor. In addition, during the project, the documentation served as a means of avoiding the distortion of ideas, helped with project control, stored the rationale of the decisions and made visible the capacities and limitations of the system (Ruiz, 1997). Documentation in the DISEL effort was collected in a data documentation and code repository online at http://kiliwa.cicese.mx/~disel and was classified within two categories reflective of process and product importance.

One classification was process documentation. Examples included project schedules, agendas, and minutes. These documents recorded the development process and maintenance of the system. In addition, the documentation made the development process "visible" and maintained information about project scheduling, prediction and control of the process (such as plans, calendars, estimates). Also process documentation included reports about the resources used during the development, standards to be implemented in the several phases, compilation of ideas and strategies to be implemented for the project members, rationale of the design decisions and details of the daily communication between managers and development team.

The other category was product documentation. Examples included requirements specification, design documentation and the code. These documents described and detailed the product developed from both the technical (system documentation) perspective and the system user (user documentation) point of view. A good example of common user documentation was the user manual. System documentation included all the documents from the requirement specification stage to the final acceptance test plan. These documents were to be used during the maintenance phase and would need to be updated every time that changes were carried out on the system. The most visible documentation, user documentation, typically contained an overview of the services that Cliq! delivered, how the system was to be used, a reference manual that enabled the handling of errors and facilities and a section that described the installation process of the system as well as an administration manual.

As mentioned earlier on, the quality of the generated documentation was of great importance since the utility of the system degrades without adequate information on how to use it or how to understand, reconfigure and redefine its development.

In order to achieve this quality the development of the documents followed standards. These standards governed the process, ensuring the satisfactory information exchange of electronic copies of the documents and fundamentally determined the nature of the documentation. These standards needed to be as specific as possible yet generic enough to be used in the elaboration and useful creation of several types of documents. Last but not least, the document standards specified the appearance of all the documents. These standards served the purpose of maintaining the consistency of the documents and allowed easy relationship constructs and use of analogy to reduce information overload. Documentation included the identification, structure and presentation of its content.

CHAPTER 6
EVALUATION

In development of the Cliq! system, complexities existed mainly due to the DISEL effort having been a first attempt that consequently magnified productivity detriments such as cultural differences, poor communication protocols that resulted in the SQA function being discharged or implemented haphazardly. This mainly occurred because the benefits of the SQA functions and a well-designed SQA framework were the least visible. Hence SQA was the first to be discharged under the pressure of meeting project deadlines that occurred with project progress in the DISEL effort. In addition, it was poorly defined.

6.1 REVIEW OF DISEL PROJECT

The MIT collaborative software engineering experience was unique as a learning experience because it brought to the forefront all the factors necessary to stress the importance of process, product and people interaction quality. A good number of the team members had minimal software development experience on the scale required by the project and hence spent a good amount of the time on the learning curve. However, this is the expected composition in a learning environment and for a non-industrial team. Another complexity was the dispersed nature of the team members. Team dynamics developed along with the project. In addition to cultural differences there were language differences (English and Spanish). The main issues involved were the cultural differences as well as the structure that was already in place at the start of the project. These cultural differences included a dependence on a set structure on one side (requiring no deviation from a topic or agenda and passive consensus) versus an aggressive, individualistic approach on the other side.

In addition, the physical separation magnified difficulties. If the team was in one physical location, a shared culture might have developed; in fact, it would have had to. The distance allowed and fostered the ability to disengage oneself and passively sink in the background, something that would have been impossible in a physically co-located environment. An important but contestable point to note (since this effort had to be distributed) is that the best possible scenario might have realized these two cultures working together on impartial ground. That is, if the MIT counterparts were in CICESE they would have had to fully adapt or move back to MIT and likewise with the CICESE students in the MIT environment. A non-MIT or non-CICESE dominant physical environment would have been ideal to develop this shared culture that was mission-critical to the whole project endeavor.

In the working difficulties that developed, a question to ask is, is it possible that a separation of culture and communication is necessary to identify the problem? If it were only MIT students or CICESE students involved in a distributed endeavor, would most of these problems persist, hence indicating more of a communication problem then a cultural difference problem?

Other more substantial issues that affected the productivity of the group involved spatial issues of the local environment on either side. Using the MIT side as an example, it would have made immense difference if the team members meeting had individual keyboards to communicate fostering participation during the lectures and lab sessions. The particular setting of the design studio with a conference-type table in the center with group members clustered around it created more of a clustering effect and more chances for passive group isolation (an incentive to associate as one team instead of two) as well as active group isolation (the opportunity to develop a we versa them mentality). In addition, the room set-up had and required too many points of foci. The cameras were at a given angle, the screen at another and the computers and mike coming from yet another angle. The cameras provide a fixed static look to the MIT students of their CICESE counterparts governing how the group interacted and particularly replied. The room was futuristic in design with lacking functionality.

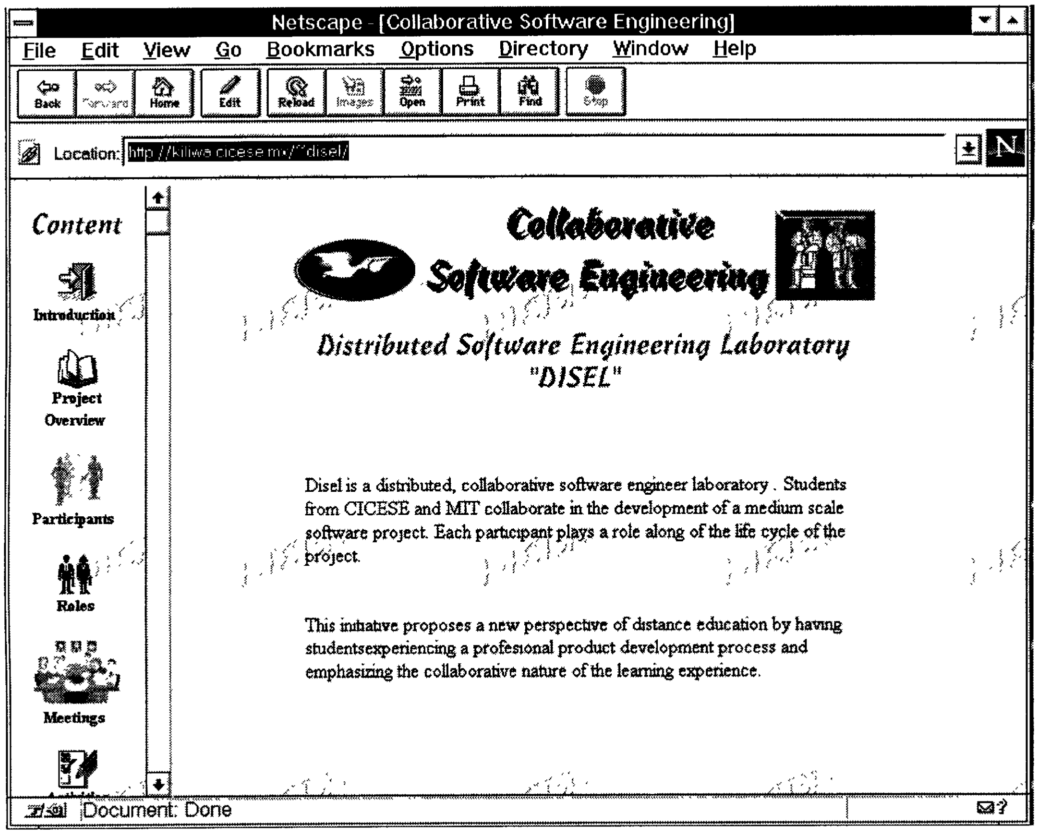
6.2 REVIEW OF INITIAL SQA PLANS AND IMPLEMENTATION TO THE DISEL PROJECT

The main result of these SQA plans and their relation to the DISEL effort was that, besides the documentation plan, they were never implemented comprehensively or even in a manner required to add substantial value to the project. As the project progressed these plans were abandoned and ad hoc methods were applied as found or thought necessary. For example, as the testing phase was entered an entirely different test plan was used instead of the one initially documented. In a similar vein, alternate quality control activities such as testing the user interface and network modules from an outside user's perspective developed with the project.

This evolution of the SQA function with time is good, however the plans failed as they were generic, massive documents that had "nuggets of value" hidden in "masses of writing". As such they may have been effective for a large development team, but were time-consuming to extract value for a small team like the DISEL one. Most of the SQA value added was from these ad hoc methods instead of the pre-prepared plans. The discrepancy in the SQA value added and how much of it came from the plans designed at the start could have been largely avoided by the following method. A method that involved designing these plans carefully and early to fit the expected model of the project in terms of client requirements, the distributed nature of the endeavor, the language and cultural differences amongst other factors.

Another shortcoming of the SQA plans was that they were not as tightly coupled to each other and the project schedule as possible. In addition, they were linear in manner with no iteration planned for either within or between the phases (such as preliminary design and specific design, or design and coding). A waterfall model was implicitly assumed. In addition, provision of incentive schemes to develop, implement and maintain SQA plans were non-existent or not explicitly defined.

An excellent implementation of the SQA function was evident in the data repository developed for the DISEL effort. This data repository was located at http://kiliwa.cicese.mx/~disel/ with the front end illustrated in figure 6-1.


Netscape - [Collaborative Software Engineering]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location: http://kiliwa.cicese.mx/~disel/
Content
Introduction
Project Overview
Participants
Roles
Meetings
Document: Done

Collaborative Software Engineering

Distributed Software Engineering Laboratory "DISEL"

Disel is a distributed, collaborative software engineer laboratory . Students from CICESE and MIT collaborate in the development of a medium scale software project. Each participant plays a role along of the life cycle of the project.

This initiative proposes a new perspective of distance education by having studentsexperiencing a profesional product development process and emphasizing the collaborative nature of the learning experience.

Figure 6-1: DISEL Repository Main Web Page

The data repository was also a direct communication media in the form of an e-mail archive that tracked communication between project participants according to subject title, author or date. A good structure allowed the on-going documentation to fit in well. In addition, timely updates of crucial importance were communicated to team members via e-mail and an established protocol that divided messages in levels of importance. However, room for improvement is possible in the organizational form of this repository.

In addition, a project plan that supported early prototyping, spending less time on analysis and more time on coding and testing, along with SQA plans that encompassed this method, would have been efficient. It would have kept team morale up, allowed sufficient time for bugs that surfaced to be resolved, and for the client to confirm that their requirements were being meet and incorporated, seeing and adapting to the effect of changes as the project proceeded. Rapid prototyping provides a tangible object with properties that can then be assessed for conformance early on in comparison to the relative abstractions of the requirements analysis and design phases. Therefore, less time should have been spent on analysis, more on coding and testing with reiterations to previous phases as necessary. This does not imply that requirement analysis is unimportant. In fact, this simply stresses the importance of the requirement analysis role since by implementing rapid prototyping the project becomes a more concrete attempt earlier on to capture the client's requirements.

In summary, the SQA plans needed to be developed in synch, account for iteration, involve rapid prototyping and fit the distributed collaborative nature of the project as best possible with room for change as the project progressed.

CHAPTER 7
CONCLUSION

7.1 RESEARCH RESULTS

Based mainly on the evaluation and partial suggestions given in Chapter 6, the following Software Quality Assurance framework has been developed mainly through implementation of the lessons learnt on the first DISEL effort and lessons learnt by Brooks in designing the IBM System/360. Most of the lessons are still meaningful to the software engineering discipline as a whole. These lessons have been used in creation of an SQA framework within the scope of a distributed collaborative software-engineering environment, the environment within which Cliq! was developed.

7.2 A FRAMEWORK FOR DEVELOPING A QUALITY ASSURANCE METHODOLOGY

Using the Capability Maturity Model as a framework for description of effective software processes and a pathway to improve an organization's software process maturity, the initial DISEL effort was at level 1 (Curran and Sanders, 1994). The next effort, without a well-defined SQA methodology, will be at level 1 as well. Level 1 as defined in Curran and Sanders is the "initial" level where "The software process is characterized as ad hoc. Few processes are defined and success depends mainly on individual effort". Key process areas that may exist include configuration management, quality assurance, sub-contract management, project tracking and oversight, project planning, and requirements management. The second level to aim for is repeatability where "basic project management processes are established to track costs, schedule and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications." (Curran and Sanders, 1994). Key processes include peer reviews, inter-group coordination, product engineering, integrated software management, training programmes, organization process definition, and organization process focus.

The purpose of this SQA framework is to help the next endeavor for distributed collaborative software development within DISEL develop SQA plans and methodologies that will help move towards fully achieving level 1 maturity and, if possible, move on to level 2.

Starting at the top-most level that is project management, estimating should not be built around cost-accounting, as such an estimation substitutes effort with progress. While it is true that cost varies with the product of man-months, progress does not. Within project scheduling itself, the use of Brook's rule of thumb to establish the lengths of the different phases will be an appropriate working estimate to start scheduling by. This rule states that a third of the schedule should be for design, a sixth for coding, a quarter for component testing, and a quarter for system testing (Brooks, 1995). Requirements should be included in the one-third of the time that is allocated to design. In addition, a schedule will need to be defined, understood by team members and closely followed. A critical-path schedule will be invaluable as an indicator of schedule slippage and associated cost. If slippage occurs and on disclosure of the project status to the project manager, acceptance and dealing with the slippage without panic will encourage future ease of full status disclosure (honest reporting). With particular application to the next DISEL endeavor and its possibility of being an extension of the present one, Brook's cautionary second-system effect which states that the second is the most dangerous system a person ever designs as the tendency is to over design it, needs to be kept in mind.

With respect to project management and documentation, the excellent effort in maintaining a data repository should be continued. However, establishing a code repository will be invaluable in allowing the programmers to keep track of code versions. This code repository should be an aspect of the data repository or an entity in itself. Appropriate access levels should be established as necessary. The data repository is to be a structure imposed on the documents that a distributed project will produce and hence its design can be established early on. This design will need to be carefully done with all documents being a part of this structure. Proper design will improve final products such as the user and systems manuals. The World Wide Web was and is an excellent medium for accessing and modifying the repository, allowing all distributed software development team members easy access to documentation from just about anywhere, a desirable feature for distributed collaborative development.

Due to the amount of paper that will be generated and, in a distributed collaborative effort, the kilobytes of electronic mail, a small number of documents will become the critical pivots around which every project's management will revolve. These will be the manager's main personal tools. For the distributed software project, the documents will be the project's objectives, user manual, internal documentation, communications protocol, schedule, budget, organization chart, and floor space allocation. Maintaining each critical document will provide the project manager with status surveillance and a warning mechanism. Other benefits will include the documents serving as a checklist and a database and, since the project manager's chief daily task will be communication and not decision-making, the documents will communicate the plans and decisions to the whole team, easing the manager's burden, providing precision and avoiding repetition. In addition, ordinary English documentation will be necessary throughout coding and other documents produced as memory will fail the programmer-user-author. In documentation the programmer-user-author will need to be concise, standing back and seeing the big picture then extracting the necessary information.

Communication being key, the organizational structure of communication will need to be a network, and not a tree, so all kinds of special organization mechanisms will have to be devised to overcome the communication deficiencies of the tree-structured organization. Scheduling disasters, functional problems, and systems bugs mainly arise because of miscommunication and assumptions. Another cause is the perfection required in programming unlike other disciplines where estimates work. In programming, team members translate ideas into programs, but human ideas are imperfect and programs need perfection to run. Project members in the distributed effort will need to communicate with each other as often as possible and with as many means as available. These include organizational mechanisms such as regular project meetings with technical briefings, the shared data repository, code repository, chat sessions and Net Meeting.

A system design fact that will allow the implementation of iterative, prototype-supporting SQA plans is that much of software architecture, implementation and realization can proceed in parallel (for example, in the Cliq! system development the network layer implementation could have proceeded in parallel with the User Interface design). In addition, stressing SQA plans that require knowledge of the full system will be impossible (and is impossible in all but the smallest systems), therefore, SQA plans developed will need to demand and make use of interfaces instead of system internals. Hence, parts should be encapsulated.

The SQA plans will also need to account for and help establish the choice of programming language, the programming environment (for example, whether the programmer will be responsible for multiple duties), programmer's experience, editor enhancements, sub-second response-time decrease, additional automation of the software development process, on-line programming, languages, and program editors. These are all factors involved in code productivity. As a start, Emacs was found to be the editor of choice in the DISEL project and was perceived as one of the principle sources of value added to the system. It will be crucial to incorporate documentation in the source program as much as possible instead of keeping it as a separate document in order to keep the documentation maintained. Three notions will be key to minimizing the documentation burden (Brooks, 1995). These will be the use of parts of the distributed system that have to be there anyway, such as names and declarations; to carry as much of the documentation as possible; the use of space and format to show subordination and the use of nesting to improve readability; and the insertion of necessary ordinary language documentation into the program as paragraphs of comment, especially as module headers.

Test cases framework will need to include valid, functional input data, some borderline input data, and some test cases for clearly invalid input data. The following is an example using an input field for dates and requiring input to be in the form of mm/dd/yy. Valid, functional data is 06/29/74, borderline input data is 02/29/96 where the borderline data is that the 29th in February only occurs in leap years and invalid input data would be something like 00/0T/0H.

An appropriate overall project methodology that should be incorporated into the SQA plans is rapid prototyping with iterations i.e. iterative enhancements. This is because for most projects (as happened in the development of Cliq!) the first system built is barely usable with it being too slow, too big, too hard to use, or all three. Hence the distributed project should plan to throw one away and probably will anyway. Rapid prototyping will allow the project to accommodate this problem, solve it (since the first few iterations behave as a first system that is consequently be thrown away) as well as gain from the accompanying concretization of clients requirements that result in building a prototype. The tractability and the invisibility of the software product will expose the developers to perpetual changes in requirements. Quantifying these changes into well-defined numbered versions will then be necessary. In addition, the total lifetime cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it therefore this needs to be accounted for in the SQA plans if the distributed system developed is to have any long term usage (Brooks, 1995). The system will entropy with additional changes as, though modules added to a system may increase linearly with versions, the number of modules affected increases exponentially (Brooks, 1995).

With respect to the SQA function directly, the following is the framework for development of plans.

- The review techniques will have to reveal the true status to all distributed team members. For this purpose a milestone schedule and completion document will be the key.

- Independence for the SQA personnel will have to be achieved. This independence mainly involves independence from project control, financially or other. As an example, the SQA sub-team could report directly to the client, establishing the advantages obtained by having independent SQA reviewers – an in-house out-sourcing effort.

- The SQA personnel will need to keep in mind that partitioning a task among multiple people occasions extra communication effort and therefore should keep the SQA team composition and size consistent as much as possible throughout the project (Brooks, 1995).

- The SQA Quality Factor of principle importance will be (conceptual) integrity from which the other important ones such as system reliability and usability for distributed collaboratively developed software systems will follow. To achieve conceptual integrity, single control of the concepts will be necessary. Such a conceptually integrated system is faster to build and test.

7.3 RECOMMENDATIONS FOR FUTURE IMPROVEMENT

As noted this is a framework within which precise and elaborate SQA methodologies may be crafted and implemented for future endeavors involving collaborative software development. In addition, separate frameworks may be devised and implemented based on the concepts, functions and models outlined earlier on. Improvement in terms of establishment of guidelines, standards and procedures specific to distributed collaborative software development may be derived as well from these in addition to the nature identified in Chapter 1.

REFERENCES

American National Standard. *American National Standard: Quality Management and Quality Assurance Standards - Guidelines for the Application of ANSI/ISO/ASQC Q9001 to the Development, Supply, and Maintenance of Software.* American Society for Quality Control Standards, 1995.

Benjamin, Kareem and Garcilazo, Juan. DISEL. September, 1997 (http://kiliwa.cicese.mx/˜disel/Documents/)

Boehm, Barry W., *Characteristics of Software Quality.* North-Holland Pub. Co., 1978.

Brooks, Frederick P., Jr. *The Mythical Man-Month.* Addison Wesley, 1995.

Brown, Bradley J. *Assurance of Software Quality.* SEI Curriculum Module SEI-CM-7-1.1 (Preliminary), Carnegie Mellon University, Software Engineering Institute, July 1987.

Buckley, Fletcher J. *Implementing Configuration Management.* IEEE Computer Society Press, 1992.

Castillo, Juan Jose Contreras. DISEL. *Maintenance Contract.* November 12, 1997 (http://kiliwa.cicese.mx/˜disel/Documents/MaintenanceContract.html#WorkPlan)

Castillo, Juan Jose Contreras and Ortiz, Juan Francisco Garcilazo. DISEL. *Plan of Test Team.* February 2, 1998

(http://kiliwa.cicese.mx/˜disel/Documents/Plans/Test/Plan.htm)

Chavez, Humberto and Rodriguez, Simonetta A. DISEL. *Project Requirements Analysis.*

December 10, 1997

(http://kiliwa.cicese.mx/˜disel/Documents/Documentation.html)

Cruz, Gregorio and Gomez, Lidia. DISEL. *Validation and Verification Plan*. December, 1997

(http://kiliwa.cicese.mx/˜disel/Documents/Plans/VV/VVDiselPlan.html)

Curran, Eugene and Sanders, Joc. *Software Quality: A Framework for Success in Software Development and Support*. Addison-Wesley., 1994.

Evans, Micheal W. and Marciniak, John. *Software Quality Assurance and Management*. John Wiley and Sons, Inc., 1987.

Flecher, Tom and Hunt, Jim. *Software Engineering and CASE: Bridging the Culture Gap*. McGraw-Hill, Inc., 1993.

Garg, Pankaj K., and Jazayeri, Mehdi. *Process-Centered Software Engineering Environments*. IEEE Computer Society Press, 1996.

Jalote, Pankaj. *An Integrated Approach to Software Engineering*. Springer-Verlag New York, Inc., 1997.

McManus, James I. and Schulmeyer, Gordon G. Eds.. *Handbook of Software Quality Assurance*. Van Nostrand Reinhold Company Inc., 1987.

Microsoft. *Microsoft Net Meeting*. 1997

Njendu, Charles and Martinez, Ruben. DISEL. *Quality Control Plan*. November 11, 1997

(http://kiliwa.cicese.mx/˜disel/Documents/Plans/QC/QcPlan.htm)

Ruiz, Diana. DISEL. *Documentation Specialist Work Plan*. October 23, 1997

(http://kiliwa.cicese.mx/˜disel/Documents/DSWorkPlan.html#Document Standard)

SEI, *Collaborative Skills in Software Engineering.* March, 1998

(http://www.sei.cmu.edu/technology/collab.skills.in.se.html)

Silicon Graphics. *InPerson..* 1997

Sinclair, John, Vincent, James and Waters, Albert. *Software Configuration Assurance: Practice and Implementation.* Prentice-Hall, Inc., 1988.

Urrea, Marcela Deyanira Rodriguez. DISEL. *Software Configuration Management Plan.*

October 30, 1997

(http://kiliwa.cicese.mx/~disel/Documents/Plans/SCM/ScmPlan.html)

APPENDIX A: DISEL QUALITY ASSURANCE PLANS

MIT/CICESE

QUALITY CONTROL PLAN

Charles Njendu, Ruben Martinez, November 11, 1997

INDEX:

QUALITY PLAN.

Quality Plan Goals.

Quality Plan Benefits.

Quality actions planned.

<u>PRODUCT INTRODUCTION:</u>

PRODUCT DESCRIPTION:

The task is to develop a system that allows geographically distributed users to: (1) exchange and publish documents, (2) handle and track a common and an individual agenda, (3) access the administrative system of the corporation, (4) hold conferences with other users, and (5) perform co-development.

PRODUCT REQUIREMENTS:

There are some requirements that the system must satisfy:
The World Wide Web must be used as the interaction method among the users.
The system must include handling of graphics, audio, and video.
The system must track organizational documents.
The system must have an effective and adequate man-machine interface.
The system must integrate information, multimedia, and communication technologies.
The system must be flexible so can be adapted to different organizational models.
The system must have the ability to integrate and communicate the different locations as if they were only one location, in other words, there must be audio and video contact between the communicator and audience from one location to the other locations and vice versa.

PLANNED MARKET:

The main users will be people from universities and from corporations who have the need to exchange information as described in the Product Description.

COMPETITIVE/PREDECESSOR PRODUCT:

Microsoft NetMeeting.

OPERATING ENVIRONMENT:

The system must be platform- and computer-independent.

THE ASSURANCE PROCESS:

Appendix A-QC Table 1 : Identification of the Product Documents to be Reviewed.

IDENTIFICATION OF THE PRODUCT DOCUMENTS TO BE REVIEWED:
Requirements Specification.
Project Plan.
Software Configuration Manager Plan.
Documentation Specialist Plan.
Preliminary Design Document.
Preliminary Operator Manual.
Detailed Design Document.
Test Plan Document.
Maintenance Engineer Work Plan.
Design Test Results Document.
Code Modules Test Result Document.
System Integration Test Result Document.
Changes made to the design.
Changes made to the code modules.
Changes recorded by the Software Configuration Manager.
System Acceptance Test Results Document.
Final Operator Manual.

IDENTIFICATION OF THE PRODUCT DOCUMENTS TO BE REVIEWED:
User Manual.

REVIEW TECHNIQUES APPLIED

Audits:

The audit is a formal review performed to evaluate conformance to standards and plans, and to ensure change integrity. The leader of the review is responsible for verifying changes as part of review report. These techniques can be applied to Initial Requirements and to Development Plans. The quality control engineer's audits will be similar to reviews done in an inspection preparation.

Walkthroughs:

The walkthrough is used to detect defects, examine alternatives, and be used as a forum of learning. The producer makes all change decisions, and change verification is left to other project controls. The walkthrough technique can be applied to the detailed requirements document, the system design, and to the preliminary and final operator manual.

Inspections:

A group of people meets to detect and identify errors in a software product. The focus is on finding errors, not correcting them. During the meeting, the reviewers present a list of errors detected in a preview review and any errors are found during the inspection. The uncovered errors must be removed and the moderator verifies the rework. This technique can be applied to the detailed design document and to the source code.

Peer Reviews:

The objective is to detect errors in the source code. An expert, who performs a mental execution of the program to pick up errors before execution or compilation, does the task.

The issues that must be reviewed are the following: correctness, misuse of variables, omitted functions, poor programming practices and redundancy. This technique is applied only to the source code.

QUALITY PLAN:

QUALITY PLAN GOALS:

To improve software quality by monitoring both the software and the development process that produces it.

To ensure full compliance with the established standards and procedures for the software and the software process.

To ensure that any inadequacies in the product, the process or the standards are brought to management attention, so they can be fixed.

QUALITY PLAN BENEFITS:

An appropriate development methodology is used.

The project use standards and procedures in its work.

Independent reviews and audits are conducted.

Documentation is produced to support maintenance and enhancements.

Documentation is produced during and not after the development.

Mechanisms are used to control changes.

Testing emphasizes all the high-risk product areas.

Each software task is satisfactorily completed before the next one is begun.

Deviations from standards and procedures are exposed as soon as possible.

External professionals can audit the project.

QUALITY ACTIONS PLANNED

Appendix A- QC Table 2 : Quality Control Plan

Date	Action Planned	Document involved	Producer of the document
Sep 4th-30th	Definition of the Quality Control Plan	Quality Control Plan	Quality Control Engineers
Sep 18th	Participation as witness in the client interview used to obtain the requirements	None	Client
Oct 2nd	Presentation of Quality Control Plan at the laboratory class	Quality Control Plan	QC Engineers
Oct 9th	Participation in the review of the requirements using the walkthrough technique at the laboratory class.	Requirements Specification	Analysts
Oct 14th	Requirement specification audit.	Requirements Specification	Analysts
Oct 14th	Project Plan audit.	Project Plan	Project Managers
Oct 16th	Presentation of the requirement analysis audit at the laboratory class	Requirement Specification Review	QC Engineers
Oct 20th	Documentation Specialist work plan audit	Documentation Specialist work plan	Documentation Specialist
Oct 28th	Requirement analysis audit.	Requirement Analysis	Analysts
Oct 30th	Presentation of the requirement analysis audit at the laboratory class	Requirement analysis audit	QC Engineers
Nov 3rd	Requirement analysis review as preparation for the technical review to be performed on Nov 6th.	Requirement analysis	Analysts
Nov 6th	Participation as reviewers in the review of the requirements analysis in the laboratory class	Requirement analysis	Analysts
Nov 13th	Participation as reviewers in the review of the requirements analysis in the laboratory class	Requirement analysis	Analysts
Nov 17th	Audit to the Software Configuration Plan	Software Configuration Plan	Software Configuration Manager

Date	Action Planned	Document involved	Producer of the document
Nov 18th	Requirement analysis audit.	Requirement analysis	Analysts
Nov 20th	Presentation of audit performed to the requirements analysis in the laboratory	Requirement analysis audit	QC engineers
Nov 20th	Presentation of audit performed to the software configuration work plan at the laboratory.	Software configuration work plan audit	QC engineers
Nov 28th	Test plan audit	Test plan	Test engineer
Nov 29th	Requirement analysis audit.	Requirement analysis	Analysts
Dec 2nd	Presentation of the requirement analysis audit in the laboratory class.	Requirement analysis audit	QC Engineers
Dec 2nd	Presentation of the test plan audit in the laboratory class.	Test plan audit	QC engineers
Jan 4th	Maintenance Engineer work plan audit.	Maintenance Engineer work plan	Maintenance Engineer
Jan 9th	Preliminary design audit.	Preliminary design	Designers
Jan 15th	Participation as reviewers in the preliminary design review in the laboratory class. This review will be a walkthrough.	Preliminary design and Preliminary design review	Designers and QC Engineers
Jan 29th	Preliminary design audit.	Preliminary design	Designers
Feb 5th	Presentation of the review results made to the preliminary design in the laboratory class.	Preliminary design audit	QC Engineers
Feb 8th	Detailed design preparation review	Detailed design	Designers
Feb 12th	Participation as reviewers in the detailed design review in the laboratory class. This review will be an inspection.	Detailed design and Detailed design preparation review	Designers and QC Engineers
Feb 19th	Participation as witnesses in the code technical review at the laboratory class. This review will be a peer review.	Source code	Programmers
Mar 3rd	Audit to the changes recorded by software configuration manager.	Changes record	Software Configuration Manager

Date	Action Planned	Document involved	Producer of the document
Mar 12th	Participation as witnesses in the code technical review at the laboratory class. This review will be a peer review.	Source code	Programmers
Apr 7th	Audit to the tests applied to the code modules. The test engineer must supply results documentation.	Test results	Test Engineer
Apr 9th	Presentation of the audits made to the code module tests in the laboratory class.	Test results audit	QC Engineers
Apr 30th	Final audits to all deliverable documents: the final user manual, the software configuration document, the system and acceptance test results document.	User manual, Software Configuration, System and Acceptance test results.	Documentation Specialist, Software Configuration Manager and Test Engineer
May 7th	Presentation of final audit results and the trend analysis at the laboratory class.	Final audits and Trend Analysis	QC Engineers

MIT/CICESE

SOFTWARE CONFIGURATION MANAGEMENT PLAN

Marcela Deyanira Rodriguez Urrea, DISEL, October 30, 1997

INDEX

INTRODUCTION

This plan applies to the DISEL project in identification and definition of the organization(s), activities, overall tasks, principles, and objectives of Software Configuration Management. This document is based on ANSI/IEEE Std 828-1990. The Software Configuration Management Plan (SCMP) policy statement is as follows:

- To keep the number of configuration changes to a minimum, and in connection to this it needs to be kept in mind that technical or engineering feasibility are not in themselves sufficient grounds for change authorization.
- Improvements not required to meet requirements will be approved and authorized only if they can be justified on cost/effectiveness grounds.

The SCMP is a living document and as a result additions, deletions, and modifications will occur as it is utilized. It will be updated, as work progresses and the necessity arises, and additional configuration activities are defined.

The primary objectives of this document are:

- To provide a coherent view in pursing a compatible method and procedure for configuration management of the systems, it's subsystems and SCI's.
- To establish a change status reporting method for the emerging Software Configuration Items (SCI's).

- To provide a reference for the common terminology and vocabulary for configuration management.
- To provide change status visibility of Software Configuration Management.

Purpose

The purpose of this document is to identify and describe the overall policies and methods for Software Configuration Management (SCM) to be used during the system life cycle of the system/subsystem and SCI's for the project. These policies and methods will be updated progressively as the work proceeds and the necessity arises. This SCMP (Software Configuration Management Plan) will establish and provide the basis for a uniform and concise Software Configuration Management practice for the system/subsystem and SCI's during the total system life cycle of the project. The primary intention of this SCMP is to provide information on the SCM policy and methods to be adopted and implemented for the project.

Scope

This Software Configuration Management Plan (SCMP) establishes the overall plan of the Software Configuration Management requirements for the system, subsystems and Software Configuration Items (SCI's) during total system life cycle of the project. This SCMP will support the four main activities of SCM: *Software Configuration Identification, Software Version Control, Software Configuration Control, Software Configuration Status Accounting and Software Configuration Auditing* [Rigby].

A software system proceeds through a sequence of stages called a life cycle. This sequence begins with the formulation of the system concept. The system concept is brought to maturity through various development stages during which it is transformed into tested and refined software. With the completion of these development stages, the system will have matured to the point of deployment. After deployment, the system enters its final, but most significant stage: the operational stage. The life cycle of a software system terminates when the operational stage terminates. There are six stages in the system life cycle. Each stage culminates in a baseline, a point at which management has the opportunity to view the system in detail in order to examine its integrity, that is, its adherence to and satisfaction of the operational requirements [Bryan 80].

The *Requirements baseline* is established when the requirements are completed and initially approved. The *Specification baseline* includes the program external specifications together with a cross-reference to the requirements and operational concept. The *Design baselines* are established when the design is initially completed and inspected. As each unit is completed, inspected, and tested a *Unit baseline* is established. After initial implementation and unit test, the programs are placed in the *Integration baseline*. The *Operational baseline* is established at the time of system shipment.

Once the initial product level has stabilized, a first baseline is established. With each successive set of enhancements, a new baseline is established in step with development. Each baseline is retained in a permanent database, together with all the changes that produced it. The baseline is thus the official repository for the product, and contains the most current version. Only tested code and approved changes are put in the fully protected baseline. It is the official source that all the programmers use to ensure their work is consistent with that of everyone else [Humphrey 90].

Glossary of Terms

Audit. An independent examination of a work product or process or a set of work products or processes with the goal of assessing compliance with specifications, standards, contractual agreements, or other criteria.

Authentication. The procedure (essentially approval) used by the approval authority in verifying that specification content is acceptable. Authentication does not imply acceptance or responsibility that the specified item will perform successfully.

Baseline. A Baseline is a Configuration Identification formally designated and applicable at a specific point in an item's life cycle. Baselines, plus approved changes from those baselines, constitute the current configuration identification. A Configuration identification document, or a set of such documents, are formally designated by the acquirer (Customer) at a specific time during a Software Configuration Item (SCI) life cycle.

Components. Components are the named "pieces" of design and/or actual entities (subsystem, SCI's, software units) of the system/subsystem. In system/subsystem architectures, components consist of subsystems (or other variations), SCI's, and manual operation.

Configuration. The functional and/or physical characteristics of hardware/software as set forth in technical documentation and achieved in a product.

Configuration Control Board. A board composed of technical and administrative representatives who approve or disapprove proposed engineering changes to an approved baseline.

Configuration Documentation. Configuration documentation is the sum of all the documents that define the physical and functional characteristics of a system, subsystem, or SCI such as specifications, design documents, engineering drawings, and source code listings.

Engineering Change Order. The formal documentation that is prepared for specification change request in accordance with the SCMP Change Procedure.

Engineering Change Proposal. The formal documentation that is prepared for proposed change in accordance with the SCMP Change Procedure.

Interface Control. Interface control comprises the delineation of the procedures and documentation, both administrative and technical, necessary by contract, for identification of functional and physical characteristics between two or more configuration items which are provided by different contractors/acquiring agencies, and the resolution of the problems.

Item. A non-specific term used to denote any product, including systems, subsystems, assemblies, sub-assemblies, units, sets, accessories, computer programs, computer software or parts.

Life Cycle. A generic term covering all phases of acquisition, operation, and logistics support of an item, beginning with concept definition and continuing through disposal of the item.

Software Configuration Control. The systematic evaluation, co-ordination, approval or disapproval and dissemination of proposed changes and implementation of all approved changes in the configuration of any item after formal establishment of its configuration baseline.

Software Configuration Identification. The current approved or conditionally approved technical documentation for a configuration item as set forth in specifications, drawings, and associated lists, and documents referenced therein.

Software Configuration Item (SCI). A configuration item is an aggregation of hardware or software that satisfies an end user function and is designated by the acquirer for separate configuration management.

Software Configuration Management (SCM). A discipline applying technical and administrative direction and surveillance to:

- Identify and document the functional and physical characteristics of SCIs
- Audit the SCIs to verify conformance to specifications, interface control documents and other contract requirements
- Control changes to SCIs and their related documentation, and
- Record and report information needed to manage SCIs effectively, including the status of proposed changes and the implementation status of approved changes.

Software Configuration Management Plan. The configuration management plan defines the implementation (including policies and methods) of software configuration. The application of Software Configuration Management on a particular program/project is the Software Configuration Management Plan (SCMP).

Software Configuration Status Accounting. The recording and reporting of information needed to manage configuration effectively, including:

- A listing of the approved configuration identification
- The status of proposed changes, deviations, and waivers to the configuration
- The implementation status of approved changes, and
- The configuration of all units of the SCI in the operational inventory.

Software Unit. An element in the design of an SCI such as a major subdivision of an SCI, a component of that subdivision, a class, object, module, function, routine, or database. Software units may occur at different levels of a hierarchy and may consist of other software units. Software units in the design may or may not have a one-to-one relationship with the code and data entities (routines, procedures, databases, data files, etc.) that implement them or with the computer files containing those entities.

Status Accounting. The process of documenting the correct, approved status of the system, including a historical record of the development of SCIs and all approved changes.

Technical (Formal) Reviews. A series of system engineering activities by which the technical progress on a project is assessed relative to its technical or contractual requirements. The formal reviews are conducted at logical transition points in the development effort to identify and correct problems resulting from work completed before the problem can disrupt or delay the technical progress. The reviews provide a method for the contractor to determine that the development of a CI and its identification has met contract requirements before proceeding to the next activity.

Version. An identified and documented software body. Modifications to a version of software (resulting in a new version) requires configuration management actions either by the supplier, acquirer, his agent, or both [Rigby].

SCM ACTIVITIES

Software Configuration Identification

Identifying Configuration Items

The first step in managing a collection of items is to uniquely identify each one. In the configuration management sense, a baseline is a document, or a set of documents, formally designated and fixed at a specific time during a SCI's life cycle.

In practice, a series of different baselines are established to permit an ordered flow of development work [Rigby]. These baselines are:

- Requirements baseline.
- Specification baseline.
- Design baseline.
- Unit baselines.
- Integration baseline.
- Operational baseline.

Once an initial product level has been stabilized, a first baseline is established. With each successive set of enhancements, a new baseline is established in step with development. Each baseline is retained in a permanent database, together with all the changes that produced it. The baseline is thus the official repository for the product, and it contains the most current version. Only tested code and approved changes are put in the fully protected baseline. It is the official source that all the programmers use to ensure their work is consistent with that of everyone else [Humphrey 90].

Naming Software Configuration Items

To control and manage software configuration items, each must be named uniquely and then organized. Each item has a name and a label that identifies it uniquely. The name is given by the creator of the item (document, diagram, plan etc.). with the label being given by the Software Configuration Manager. This label consists of three parts, the first part consists of a role identifier as depicted in table 1 and the second part as a character indicating the type of the document as depicted in table 2. The third part of the label is an identification number that will be provided by the Software Configuration Manager according to the type and sequence of document. A list of all the items generated during the life cycle of the software will be published in this page. The second and third parts are to be separated by a hyphen.

Appendix A-SCM Table 1 : Role identifier

Role	Identifier characters
Project Manager	PM
Analyst	An
Designer	Ds
Programmer	Pr
Test Engineer	TE
Quality Control Engineer	QQ
Validation and Verification Engineer	VV
Documentation Specialist	Ds
Maintenance Engineer	ME

Role	Identifier characters
Software Configuration Manager	SC

Appendix A - SCM Table 2 : Characters indicating the type of the document.

Document	Identifier character
Work Plan and others plans	P
Diagrams	D
Specifications	S
Requirements	R
Meeting Minutes	M
Code	C
Any other Form	F

SOFTWARE VERSION CONTROL

Since any of the SCI's can change at anytime within the software life cycle, the SCI's will have a label indicating their actual version. This label will consist of three numbers separated by a period, and they will change when the reviewers (CCB) consider it necessary. The criterion that they will have to use is the following:

Appendix A- SCM Table 3 : Criteria.

Number	The change was	Changes made
1	Not meaningful	some corrections and minimum changes
2	Meaningful	Important updates and improvements
3	Very meaningful	Great modifications

SOFTWARE CONFIGURATION CONTROL

The roll of software configuration control is to provide the administrative mechanism for precipitating, preparing, evaluating, and approving or disapproving all change proposal processing. Software configuration control focuses on managing changes to SCIs (existing or to be determined) in all of their forms.

The engineering change proposal (ECP), a principal control document, contains information such as a description of the proposed change, identification of the originating organization, rationale for the change, identification of affected baselines, SCIs (if appropriate) and others. ECPs are reviewed and coordinated by a configuration control board (CCB), which is typically a voting body representing organizational units that have a vested interest in proposed changes. The CCB consist of one person from each of the following roles: Project Manager, Designer, Programmer, V&V Engineer and Software Configuration Manager.

ECP's can be made in response to a variety of events. These include:

- Software deficiencies
- New operational requirements
- Economic savings
- Schedule accommodation

Change control combines human procedures and automated tools to provide a mechanism for the control of change. The procedure to propose a change is the following:

An ECP is submitted to the SCM and it is evaluated by the CC, the group that will make a final decision on the status and priority of the change. The results of the evaluation (approved or disapproved) are presented as a change report in the same ECP form. If the change is approved an engineering change order (an approved ECP) will be sent to the responsible party to implement the change. But if the change was disapproved, the person(s) who requested the change will be notified. The object to be changed is "checked out" of the project database, the change is made, it is reported to SCM, and the appropriate SQA activities are applied. The object is then "checked in" to the database and the appropriate version control mechanisms will be used to create the next version of the software. Once the change is implemented, SCM will notify the person(s) who requested the change [Pressman 92].

Change incorporation is not an SCM function, but monitoring the change implementation resulting in change incorporation is such a function. The analysis that may be required to prepare an ECP is also outside the SCM responsibilities. The ECPs not approved by the CCB are not simply discarded but are archived for future reference [Bryan 80].

SOFTWARE CONFIGURATION STATUS ACCOUNTING

Software configuration status accounting is the administrative tracking and reporting of all software items formally identified and controlled. Software configuration status accounting is thus the means by which the activity associated with the other four SCM functions is recorded; it therefore provides the means by which the history of the software system life cycle can be traced.

A Status Accounting Report (SAR) will be made regularly by the SCM to keep management and practitioners informed of important changes. A report will be made when one or more of the following events occur [Pressman 92].

An SCI is assigned new or updated identification

A change is approved by the CCB

A configuration audit is conducted by SQA

The SARs generated will be placed in an on-line database, so that software developers or maintenance can access change information.

SOFTWARE CONFIGURATION AUDITS AND REVIEWS

Identification, version control, and change control help the software developer to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms can track a change only until an engineering change order is generated. How can we ensure that the change has been properly implemented? The answer is twofold: formal technical reviews and the software configuration audit.

Software configuration auditing provides the mechanism for determining the degree to which the current state of the software system mirrors the software system in baseline and requirements documentation. It also provides the mechanism for formally establishing a baseline.

The formal technical review focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine its consistency with other SCIs, omissions, or potential side effects. A forma technical review should be conducted for all but the most trivial changes.

A software configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during the review [Pressman 92].

Bibliography

[Bryan80] Bryan, William, *Tutorial: Software Configuration Management*, IEEE Computer Society Press 1980.

[Buckley92] Buckley, Fletcher J., *Implementing Configuration Management*, IEEE Computer Society Press, 1992.

[Humphrey90] Watts S., *Managing the Software Process*, Ed. Addison-Wesley 1990.

[Pressman92] Pressman, Roger S., Software Engineering, Practitioner's Approach, Ed. Mc. Graw Hill 1992. Hoek, A., The Configuration Management Yellow Pages.

[Loria1]http://www.cs.colorado.edu/users/andre/configuration_management.html Loria, M., *CM.*

[Loria2]http://www.loria.fr/~molli/cm-index.html *Key Practices of the Capability Maturity Model Version1.1.*

[Rigby]http://www.rbse.jsc.nasa.gov:80/process_maturity/CMM/TR25/tr25_12f.html Rigby, K., *Configuration Management Plan.*

[Standard] http://www.airtime.co.uk/users/wysywig//cmp.htm IEEE Std 828-1990, IEEE Standard for Software Configuration Management Plans.

MIT/CICESE

PLAN OF TEST TEAM

Juan Francisco Garcilazo Ortiz. Juan Jose Contreras Castillo, DISEL, February 2, 1998

INTRODUCTION:

The testing plan was developed with the purpose of providing confidence in the correctness of a program. With testing, the only way to guarantee a program's correctness is to execute it on all possible inputs, which is usually impossible. The following Test Work Plan is intended to guarantee that the final product satisfies the requirements given by the client in addition to assessing program quality by executing the program in a test environment with test data. Quality doesn't only mean correctness, but also concepts like reliability, performance, robustness, usability and so on.
The main objective of this plan was to design a test that systematically brings to light different error classes, in minimal time and manpower without loss of efficiency as a preventive measure and not a curative one.

COMPONENTS OF TESTING METHODOLOGY

Different testing concerns can be related to each component. The testing methodology is composed of the following components.
the program to be tested and the testing environment,
methods for obtaining test cases,
methods for executing test cases and for evaluating the results, and
methods for assessing the quality of the program based on the results.

PROGRAM DEVELOPMENT

The development of a program involves many concerns, some of which do affect testing. As an example, object-oriented development will lead to different kinds of errors, and hence different testing techniques, compared to more traditional development processes.

TESTING ENVIRONMENT

The testing environment affects how easy it is to control the execution and observe faults. Concerns include testability, design for testability, and managing test scaffolding (stubs, drivers, instrumentation code). Different testing takes place in the developer's environment than in the user's environment.

OBTAINING TEST CASES

Methods to obtain test cases are the main concern in testing. Some concerns in this category include:
how to select "good" test cases, (where good may mean good at catching errors or representative of user input);
how to compute the inputs to exercise a particular program element;
how to create test cases that are modular, easy to understand and easy to use.

TEST EXECUTION

The execution of test cases is a very tedious job and can usually be easily automated.

EVALUATION OF TEST RESULTS

The evaluation of test results may mean determining whether the program output is correct with respect to the specification. Evaluation is even more difficult for usability testing: we need to determine if the program was "easy to use" for the test case.

ASSESSMENT OF QUALITY

The whole goal of testing is the assessment of the quality of the program. Assessing quality is more advanced for reliability testing than correctness testing. There have been very few methods developed to assess the correctness of a program, and these generally apply only to very small programs.

TEST TEAM ACTIVITIES

Knowing the language, platform and environment in which the software will be developed is very important. As mentioned above this influences the ease of obtaining and developing the test cases.

With the language, platform and environment, the test cases will be designed using diverse parts including the design document, analysis document and requirement analysis document. As a test team it is necessary to participate in the revision of the analysis document to get familiar with the user requirements and the design methodology

After that, the design tests will be realized in such a way that the functionality, performance and good application of the methodology will be assured.

Usually testing is divided into three levels: unit, integration, and system.

At the unit level, we test each unit individually, using abstraction to enable us to ignore the rest of the system.

At the integration level, we must abstract away the internal working of each unit, and focus on the interactions between units.

Finally, at the system level, we must abstract away even the high-level structure and concentrate solely on the external behavior.

REASONS FOR TESTING AT DIFFERENT LEVELS

The testing of a program happens at different levels of abstraction in order to reduce complexity and consequently increase our ability to control and observe test results, thereby allowing us to use more realistic and precise test criteria.

UNIT TESTING

At the unit level, it is easier to achieve complete coverage criteria. The test is usually structural and code-based, but a functional technique based only on the specification of the unit is sometimes used. However, when all units have been tested individually, the overall system has undergone a structural test.

Unit test alone cannot guarantee that the program produces the desired global behavior. The first reason is that only a part of the behavior of a unit is explicitly specified. Thus, unit testing can fail to notice extra behavior that is produced as a side effect of computing the desired behavior, and missing behavior that is expected implicitly by another unit. The second reason is that even if all units are correctly implemented, the global behavior may be incorrectly implemented because the specifications for the units are wrong or they are incorrectly combined.

INTEGRATION TESTING

Integration testing focuses on finding errors that occur due to interaction of units. Since the structure of interacting units is so complex, it is important to take advantage of the fact that each unit has already been tested, and abstracts away its structure. In addition, it is important to work at different levels of abstraction. Integration should start with small groups of units. When these are tested, the structure of the groups is abstracted away, and more units are integrated into the groups. The process proceeds until all units are integrated into one large group.

Another feature of integration testing is that there is no specification for the behavior of a group of units, whereas behavior for a unit is usually specified, and behavior for the whole system must always be specified. The required behavior of the group is implicitly defined by the structure of the group. Thus, it is impossible to use functional testing for the integration level. It is also more difficult to decide if the program behaves correctly or not.

SYSTEM TESTING

Since integration testing is structural, it has the disadvantage of not being able to eliminate variant behaviors; this is done in system testing. System testing may also benefit from knowledge of structure in order to reduce the number of functional tests.

TEST DOCUMENTATION

For the report elaboration of the realized tests we designed a format that should have the following information.

Test identifier.

Version of the test.

Name of the test.

Level of the test.

Modules to be test.

Test Objective

Test date

Required time

Inputs

Expected outputs

Results

Comments

Conclusions

With the information obtained we could have a history of all the tests applied to the system, as well as a database of errors found in the design and the programming stages. The following is the information to be included in this database .

BUGS DATABASE

Date.

Stage {Design or Programming}.

Type of error

Syntax

Incongruent with respect to the last stage

Class

Relation

Function

Attribute

Other (Specify)

Level of error (High, Medium, and Low).

Section of the design document or code where the error was found (Row, Page, and Diagram).

Comments.

PROBLEMS THAT MAY ARISE

Some of the problems that could prejudice the good performance of the tests are:

The design does not correspond to the programmer's code.

There are delays in the previous stages to tests.

The code does not present a uniform programming style.

The code is not documented well enough for third parties to interact with it.

There are no final versions of design or code.

Appendix A - TE Table 1: List of Activities

Activity	Date
Participation in the revision to the Requirements Analysi Documents	10/30/97-12/2/97
Work Plan Preparation	11/12/97-12/2/97
Application of Test to Design	1/9/98 - 1/22/98
Execution of the low level test	2/13/98-2/20/98
Execution of the middle level test	2/20/98-2/26/98
Execution of the High level Test	4/23/98-4/30/98
Creation and Implementation of Test Documentation	11/01/97-4/30/98

DOCUMENTS GENERATED

Test Plan

Results of test applied to modules and methods

Results of integration test

Result of usability test

Result of regression test*

If is necessary

MIT/CICESE

VALIDATION AND VERIFICATION PLAN

Gregorio Cruz and Lidia Gomez, December 1997

INTRODUCTION

Software verification examines the products of each development activity. The examination is to determine if the outputs meet the requirements established at the beginning of the activity. Validation ensures that the software is a correct implementation of the system requirements for which the software is responsible. In addition, validation ensures that this check is conducted concurrently with, and at the end of, all software development activities. Software V&V tasks analyze, review, demonstrate or test all software development outputs.

A review process can be defined as a critical evaluation of an object. Walkthroughs, inspections and audits can be viewed as forms of review processes. This document presents the first version of the V&V plan for the project. This plan depends on the project's plan since V&V has to adjust their schedule to the one proposed by the project manager.

OBJECTIVES OF THE V&V WORK PLAN

The main objective of this plan is to make sure that the final product will satisfy all the requirements established by the client, both at a functional and performance level.

PARTICULAR OBJECTIVES

- *Correctness.* Making sure that the final product is free of errors
- *Consistency.* Making sure that the product is consistent with itself and with the other products. For example, the definition of a function key should be the same for all interfaces (e.g. F1= help).

Necessity. The system must have all that is necessary in order to operate in accordance with the requirements of the client. That is, we should not add more capacities to the system then the ones needed.

Sufficiency. Making sure that the system is complete (finished).

Performance. Making sure that the system satisfies all the requirements established by the client. In the case where they are not specified, the system must match the capabilities of other similar products that exist in the market.

REVIEW LOGISTICS

AGENDA

The review agenda is provided in advance as a planning device for the project team in preparing its presentation and as a scheduling notice for the panel members. In order to assure an efficient review, the agenda needs to be adhered to by the presenters and the panel members.

Preparation by the panel members is often a key factor in meeting the agenda schedule. The agenda will be available each Tuesday, that way all the members of the group will be able to access it and know in advance the activities that will take place.

Part of the responsibility of every review attendee is to return from meals and breaks on time☺.

CONDUCTING THE REVIEW

The moderator is responsible for ensuring that the reviews add value to the engineering process. The moderator will serve as a facilitator to elicit participation by the panel members and encourage open discussion of issues by project team members. At the same time, the moderator needs to exercise control of the review activity to ensure that discussions are pertinent to the review.

REVIEW METHODOLOGIES

There are many ways to perform technical reviews. Most of these approaches involve a group meeting to assess a work product; however, variations of reviews exist that do not require a review group meeting. V&V will focus on reviews that involve a group meeting.

WALKTHROUGHS

Walkthroughs are presentation reviews in which a review participant, (usually the developer of the software being reviewed) gives a description of the software and the rest of the review group provides their feedback throughout the presentation.

The walkthroughs can be done at any phase of the project. The recommended group size should be 4 to 6 people going through the following process:

An agenda will have been prepared that will have product definition and revisions needed, participants selection, role assignments and obligations, type of revision, place, date, duration of the revision, specific points that will be discuss and the time assigned to each activity.

All the members from the selected reviewers group will be notified about the meeting where they are to review the selected product. Each notified person must confirm their participation.

Each reviewer will be notified about the product material to be evaluated. This is the author's responsibility.

In addition, each reviewer will have the responsibility of reading and revising the material before the meeting with the goal of make the meetings more productive. It will be optional for the reviewer to make annotations about the details (errors, defects and observations) found during his/her individual revision which he/she might consider necessary to discuss during the revision meeting. During the revision session, the author will present the product material to be revised.

The people making the revisions must evaluate the material being presented. The revision will focus primarily on the detection of errors as well as explore possible problem areas. Once the presentation of the product is finalized, a time will be assigned to each reviewer so that he can present any corrections or questions according to his criteria and experience.

The author in this case will be the person responsible for the revised material and answer, clarify, and explain all the questions that the revision team may ask. The secretary will note all of the points discussed in the meeting, and will be recorded in the minutes of the meeting.

During each session, a moderator will be present and will be responsible for keeping the agenda on track. The time recommended for this type of revision is 2 hours but in DISEL an hour and 20 minutes will be used. This will be barely enough time to revise the material therefore V&V will have to cover the principal aspects of the material and ensure that all the participants are focused on the revision.

INSPECTIONS

Inspections should be presented in more formal approaches that can be viewed as work product reviews. Inspections require a high degree of preparation by the review participants, but the benefits include a more systematic review of the software and a more controlled and less stressful meeting. Software formal inspections are in-process technical reviews of a product of the software life cycle conducted for the purpose of finding and eliminating defects. The main difference between walkthroughs and inspections is that an inspection process involves the collection of data that can be used as feedback on the quality of the development and review process.

The formal inspections can be applied to total or partial products in the software process, including requirements, specifications, designs and code. Usually, formal inspections are done during the first phases of the project, their goal being to make sure that defects found are fixed during the early phases of the project, when they are easier to detect and their correction is less expensive. The process that V&V will follow for the formal inspections contain 5 phases, which are detailed as follow:

SCHEDULE

The schedule is the moderator's responsibility. The schedule is the time used for:

- Determine the starting criteria for the product to be inspected
- Select the team of inspectors and assign them roles
- Schedule the meeting agenda, and
- Prepare and distribute the formats and material for the inspection.

PREPARATION

During this phase, the inspectors will individually prepare for the inspection. They will revise the product in detail with the purpose being to get familiar with the material and find potential defects. Lists that contain different types of typical defects that can be found in the inspection of the material help the inspectors do the revisions. For example, if it is a code inspection, some typical defects would be syntax errors, the need of initializing variables, the different types of variables and so on. The inspectors should, during preparation, register in a special format the defects that were found and the time used.

Each inspector needs to send the format that contains all the defects found before the session to the moderator. The moderator will then revise these formats and determine how well the team is prepared. In addition, the moderator will check to determine which problems found will require extra attention during the inspections.

INSPECTION MEETING

The objective is to detect errors in a partial or total product and verify it.

The activities in this phase are:

The moderator will monitor the meeting, going over the agenda for the inspection meeting at the beginning. In addition, he will ask if the reviewers are prepared for the inspection. If they are not prepared, the revision will not occur. The moderator will then check to see if the producer is fully prepared. For example, whether the reviewers' error logs have been analyzed, duplicates consolidated, and discussion notes prepared.

The producer will then review each major error either to clarify why it is an error and understand what the reviewer(s) meant, or verify that it is not an error. Pertinent data on each error will be recorded. This will include error location, a brief description, the error category, and the cause of the error if readily apparent. After discussing all major errors, the product will be briefly reviewed to identify any other areas of confusion or concern. Data errors identified in this process will also be recorded. Throughout the meeting, the moderator will keep the discussion focused on identifying and explaining the errors.

At the conclusion of the meeting, the moderator will check that all participants have voiced their concerns and questions, that all action items are clearly defined and understood, and that the responsibilities and planned resolution dates are established. All the necessary forms will be reviewed to ensure that all required data was obtained. The basic forms are the error logs, the preparation form, the inspection report, and the inspection summary. Based on the inspection results, and after asking the reviewer for their views, the moderator will then decide whether a re-inspection is required.

Once the meeting has ended, a brief summary about the defects found will be given and it will then be decided if there will be a new product inspection. In addition, the moderator and author will determine correction times and the date(s) for the moderator to check if changes have been done. Finally, all that occurred in the meeting will be registered. The secretary will be charged with this activity and it is recommended that the meeting time be a maximum of 2 hours.

CORRECTIONS

During this phase, the authors will fix the defects found during the inspection and that were found in the list of defects.

FOLLOW UP

The moderator and author will determine if the defect correction in the product has occurred. If all the defects are corrected, then the product will satisfy all the requirements specified and the moderator will log the final report of the inspection. If not, the correction phase must be repeated until the requirements are satisfied.

Peer Reviews

The purpose of peer reviews is to remove defects from the software work product early and efficiently. An important corollary purpose is to develop a better understanding of the software work products and of defects that might be prevented. Peer review activities will be planned, review plans will be documented and peer reviews will be performed according to a documented procedure. Then defects in the software work products will be identified and removed. In addition, adequate resources and funding will be provided to perform peer reviews on each software work product to be reviewed. Reviewers who participate in peer reviews will receive required training on the objectives, principles, and methods of peer reviews.

AUDITS

Audits are an external review process and serve to ensure that the software is properly validated and that the process produces intended results.

When a technical revision is to be done, the revising team or inspectors will consist of a moderator, author and secretary amongst others. These reviewers will be charged with finding defects in the product missed or overlooked during the internal preparation and inspection meetings. The Moderator will guide the inspection team and will be responsible with ensuring that a good inspection is performed. The moderator will be active during the whole section. The author is the creator of the product under revision and will present the material for revision. The secretary will be responsible for registering the information generated during the meeting, the defects found and the agreements reached.

The observing team in the revision will take note of all that is happening during the audit. That is, they will only be observers and will not be able to make any opinions about the revision.

It is recommended that the revising team be chosen according to the products that will be revised. For example, if the requirement specification document is to be analyzed, then it is efficient to invite the designers, V&V engineers and Quality Control engineers, to revise the product.

V&V DURING THE SOFTWARE LIFE CYCLE

V&V engineers must have continuous collaboration with the testing and quality control engineers. The following describes how V&V will apply the V&V process at each phase of the development of the Cliq! system.

Software Concept and Initiation Phase

During this phase, the software concept is developed, the feasibility of the software system is evaluated, and the acquisition strategy is developed. The most important document to verify then will be the portion of the system requirement document that applies to software.

Software Requirements Phase

During this phase, the software concept and allocated system requirements are analyzed and documented as detailed software requirements. The requirements document should be revised for completeness and accuracy, for traceability back to previous level documents, and to assure that a sufficient base is provided for the software design. In addition, V&V will perform the following activities:

- Evaluation of the defined concept to determine whether it satisfies user needs and project objectives in terms of system performance requirements, feasibility (such as compatibility of hardware capabilities), completeness, and accuracy.
- Evaluation of the software requirements for accuracy, completeness, consistency, correctness, testability, and understandability.
- Assessment of how well the software requirements accomplish the system and software objectives.

Software Architectural (Preliminary) Design Phase

During this phase, the overall design for the software is developed, implementing all of the requirements in software components. V&V and other DISEL project members will review the design to ensure satisfaction of and trace-ability to the requirements, correctness, clarity, code-ability, testability, and consistency.

Software Detailed Design Phase

During this phase, the architectural design will be specified to the unit level. Constraints and system resource limits will be re-estimated and analyzed, and staffing and test resources validated. The detailed design will follow the base-lined higher level design exactly, and will be inspected for the same characteristics. In addition, the design will be inspected for satisfaction of software quality engineering standards such as information hiding and the use of simple structures.

Software Implementation Phase

During this phase, the software will be coded and unit tested. All documentation will be produced in quasi-final form, including internal code documentation. Code verification will check for technical accuracy and completeness of the code, verifying that it implements the planned design, and ensuring that good coding practices and standards are used. Documents will be inspected for accuracy, completeness, and traceability to higher level (previous phase) documents. The revision team will be selected from participants in the detailed design, code, test, verification and validation, or from within software quality assurance.

Software Integration and Test Phase

During this phase the software units will be integrated into a completed system; non-conformance will be detected, documented, and corrected; and Cliq! will be demonstrated that it has meet client requirements. The integration and test plan will be executed, the software documentation updated and completed, and the products finalized for delivery. The final version of the Acceptance Test Plan will be verified to detect defects in the definition of test cases and to verify that each test case verifies the requirements with which it is associated.

In this phase, the objectives of the software integration test will be established as well as the strategies to be employed, the coverage requirements, reporting and analysis, and elimination of anomalies. In addition, interface testing of reused software from other system software may be planned and tracing of test design, cases, procedures, and execution results to software requirements.

The inter-unit communication links and test aggregate functions formed by groups of units will be checked, confirmation that anomalies during test are software anomalies, and not problems detected for other reasons. Finally, functional, structural, performance, statistical, and coverage testing of successfully integrated units after each iteration of software integration and successful testing of interfaces and interactions.

Software Acceptance and Delivery Phase

The formal acceptance procedure will be carried out during the acceptance and delivery phase. The objectives of the software system test, the strategies to be employed, the coverage requirements, reporting and analysis, and close-out of anomalies will be established. System and software requirements to test software design, cases, procedures, and execution results will be traced.

In addition, testing the operation of the software as an entity; confirming that anomalies during test are software anomalies, ensuring that any changes to software have been made; and conduct a re-test as necessary needs to be done. Finally, it will be determined that all software outputs needed to operate the system are present and checks done to ensure that the software installed in the system is the software that underwent software V&V.

Software Maintenance and Operations Phase

In this final phase, Cliq! will be used to achieve the objectives for which it was developed and acquired by the client. Corrections and modifications will be made to the software to sustain its operational capabilities and, as necessary, upgrade it to support new uses. In this phase, software changes may range in scope from simple corrective action to major modifications that require a full life cycle process. V&V will be scheduled in response to the degree of new development activity involved.

Appendix A - V&V Table 1: Scheduled V&V Activities.

DATE	ACTIVITY
October 9, 1997.	A walkthrough will be given on the client requirements.
October 16 & 23, 1997.	Audits to the analysis of requirements will be done.
October 28, 1997.	Audits done on the requirement analysis will be presented by the Quality Control and V&V Engineers.
November 6, 1997.	Project specifications applied to the inspection technique will be presented and revised.
November 13, 1997.	Project specifications review.
November 20, 1997.	Testing plan revision.
December 2, 1997.	Specification audits by Quality control and V&V.

DATE	ACTIVITY
December 4-9, 1997	Elaborate of plan for product acceptance requirements.
January 15, 1997	Walkthrough of the preliminary design of the product.
February 5, 1997	Presentation of tests and audits made to the design.
February 12, 1997	Inspection of a detailed revision of the design.
February 19, 1997	Peer review of code presentation (modules).
February 26, 1997	Presentation of tests made to the code modules.
March 12, 1997	peer review of code presentation (modules).
March 19, 1997	Presentation of the tests and audits made to the code modules.
April 9, 1997	Presentation of the tests and audits made to the code modules.
April 16, 1997.	An audit will be done to the integration of the system.
April 23, 1997	An acceptance test will take place.

DATE	ACTIVITY
May 7, 1997	Results of audits to the system will be presented.

MIT/CICESE

MAINTENANCE CONTRACT

Juan José Contreras Castillo, DISEL, November 12, 1997

INDEX

MISSION

To adapt the system to changes in its external environment, making enhancements requested by users, and reengineering an application for future use.

OBJECTIVES

- Assure that the development team is informed of the errors found in the program.
- Try to maintain the software modernized by means of changes carried out in it.
- Modify the software in order to adapt new functions or modify some already existent.

WORK PLAN

The maintenance activities are carried throughout the development of the project. There are activities geared to each phase of the project, from preventive to corrective maintenance. The maintenance organization consist of members from each role, therefore this means that maintenance will consist of 10 persons including the maintenance engineer.

Appendix A - ME Table 1: Scheduled Activities.

Activity	Time
To read about the role.	1 week
Establishment and coordination of the organization of maintenance within the project.	2 weeks
Prescription of procedures of evaluation and information.	2 weeks
Development of tools to care about all the maintenance petitions.	2 weeks
Definition of a standardized sequence of events for each petition of maintenance.	2 weeks
Establishment of a system of registration and information of the activities of maintenance and the maintenance organization.	1 week
Definition of approaches of revision and evaluation.	1 week

MIT/CICESE

DOCUMENTATION SPECIALIST WORK PLAN

Diana Ruiz, DISEL, October 23, 1997

INDEX

DOCUMENT STANDARD

Document standards act as a basis for document quality assurance. Documents produced according to appropriate standards have a consistent appearance, structure and quality. The documentation standard includes the process standards, product standards and interchange standards.

PROCESS STANDARDS

Process standards define the approach to be taken in producing documents. There are a large variety of editors that allow creation of document in HTML format. Editors including Microsoft Word and Netscape Gold Navigator Editor are WYSIWYG Editors that have a lot of functions for HTML documentation.

As a part of the process standard it is necessary to use a spelling and grammar checker tool. This will avoid a lot of errors and generate a better document. It is recommended the team developing the document reviews performs these checks. This process ensures that all the team members agree to the document content. In addition, it is easier to detect errors when more than one person reviews the document.

PRODUCT STANDARD

Products standard apply to all documents in the software development life-cycle. Documents should have a consistent appearance with documents of the same class having a consistent structure. The documentation format defines its presentation.

The documentation structure standard defines the distribution of the elements of the different documents that will be developed. These standards purpose is maintaining the consistency of the documents and are based on HTML format.

INTERCHANGE STANDARDS.

Each one of the documents created during the development of the project will be in HTML format. This will simplify their publication on the World Wide Web (WWW).

INFORMATION REPOSITORY

Before designing a repository of information it is necessary to determine the necessary characteristics. For the DISEL project it is desirable that the repository of information be accessible from several places. This characteristic is necessary because the team is distributed. Therefore, it was determined to create and use a Home Page for the project as a starting interface to consult the documents. A database will be developed for that purpose and will include query functions as necessary into the DISEL repository.

The database will contain references to all the documents that are generated during the process, including agendas, minutes, agreements, requirements, analysis, design, code, tests, audits, work plans, schedules and reports. The database will include only a reference to the name of the document and important data about it including date of generation and authors.

The documents will be stored in a file system. This file system will contain all the files for DISEL.

The only information the database will not contain are the e-mails sent between the participants, since these will be stored in a hyper-mail archive.

DATABASE.

The database used will be ORACLE and this will be accessed through JAVA routines, using query forms elaborated in JAVA (Applets), HTML forms and Servlets (JAVA functions).

The following information will be stored in the database:

Project:

- Id.
- Name.
- Description.

Meeting:

- Portfolio of the meeting.
- Project.
- Date.
- Type of meetings.
- Participants.
- Role of each participant in the meeting.

Participants:

- Names
- Role.
- Password.
- E-mail address.
- Place (MIT or CICESE).
- Project.

Documents:

- Date of publication.
- Version (managed by the Software Configuration Manager).

- Physical name of the document (file in disk).
- Author(s).
- Type of document.

Reports:

- Participant.
- Receive date.
- Date of assignment.
- Advance.
- Number of report.

Audits:

- Date.
- Document audited.
- Information.

FILE SYSTEM STRUCTURE.

The file system will consist of a directory structure. Some of the directories will contain sub-directories. The root level will be a public_html/ directory. The structure will be similar to the structure of the Home page for consistency and ease of use. The root directory will contain the folders of the sections and files.

- *Activities.* This directory will contain the information of the Activities section. It will contain the following directories and files:
  - **Activities.html.** Frame definition file of the section.
  - **ActivitiesIndex.html.** Index of the section of activities.
  - **Contract.htm.** Team contract file.
  - **Homework.html.** Index of the Homework's.
  - **Homeworks.** Directory that contains assignment given.
  - **ProgressReport.html.** Form of progress report.
- *Documents.* This folder will contain the files of the documents developed like work plans, requirements, design, etc.

  - **DocIndex.html.** Index of the section of Documents.
  - **Documentation.html.** Main file (query form) of the last documents version.
  - **Documents.html.** Frame definition file of the section.
  - **Plans.** Contains the work plans of the roles.
  - **ProjectSchedule.** Contains the different versions of the project schedule.
  - **Requirements.** Contains the different versions of the requirement document.
- *Images.* This folder contains all the images used in the home page except the logos of the roles.
- *Instructor.* This folder contains the DISEL Role Selection Form.
- *Lectures.* This folder contains the slides of the presentations (each group of slides are in different folders).
- *Meetings.* This folder contains :
  - **Agendas.** This folder contains the agendas.
  - **Agendas.html.** Index of agendas.
  - **Agreements.html.** Index of agreements documents.
  - **Meetings.html.** Frame definition file of the section.
  - **MeetingsIndex.html.** Index of the section.
  - **Minutes.** Folder that contains the minutes.
  - **Minutes.html.** Index of minutes.
- *Overview.* This folder contains the files of the vision, mission and objective of the project and the frame definition file.
- *Participants.* This folder contains the index of the participants and the frame definition file of the section.
- *Roles.* This folder contains the folders of the description of each role, the index file of the section and the frame definition file of the section. The role description folders are:
  - **Analyst.** Contains the files of the role description of the Analyst, their logo and title (GIF images)
  - **Design.** Contains the files of the role description of the Designer, their logo and title (GIF images)

- **DocSpec.** Contains the files of the role description of the Documentation Specialist, their logo and title (GIF images)
- **Maintenance.** Contains the files of the role description of the Maintenance Engineer, their logo and title (GIF images)
- **Programming.** Contains the files of the role description of the Programmer, their logo and title (GIF images)
- **ProjMan.** Contains the files of the role description of the Project Manager, their logo and title (GIF images)
- **QControl.** Contains the files of the role description of the Quality Control Engineer, their logo and title (GIF images)
- **SCM.** Contains the files of the role description of the Software Configuration Manager, their logo and title (GIF images)
- **Test.** Contains the files of the role description of the Test Engineer, their logo and title (GIF images)
- **VandV.** Contains the files of the role description of the Validation & Verification Engineer, their logo and title (GIF images).

Each one of these folders contains the following files:

- *XAct.html.* Activities of the role.
- *XBib.html.* Bibliography
- *XFrame.html.* Frame definition of the role
- *XIntro.html.* Introduction of the role.
- *XLogo.gif.* Logo of the role (image)
- *XMeth.html.* Methodology of the role.
- *XObj.html.* Objective of the role.
- *XProf.html.* Profile of the role.
- *XRelat.html.* Relation with other roles.
- *XSup.html.* Supporting tools of the role.
- *XTitle.html.* Title file of the role.
- *XTitle.gif.* Title Image of the role.

- *XWp.html.* General Work plan of the role.

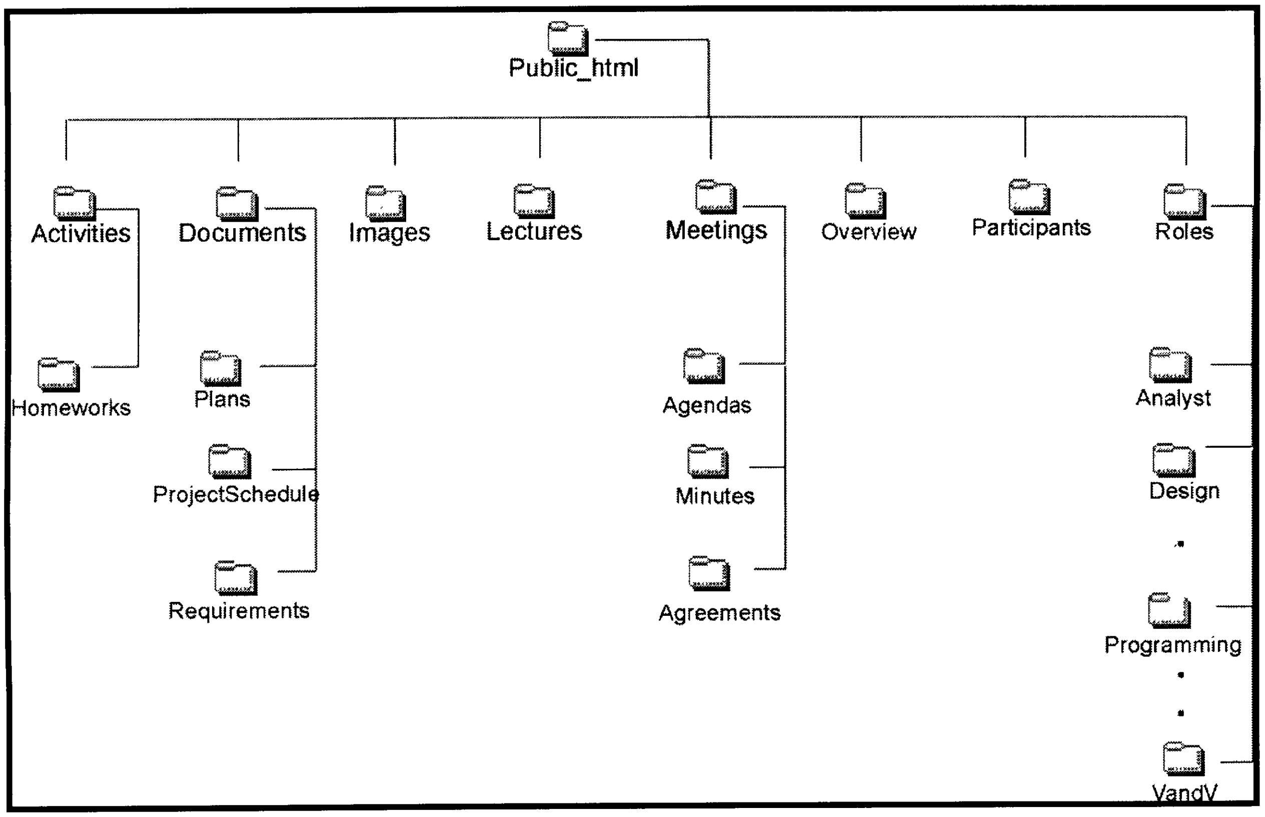
Where the *X* is denoted as necessary by:

- *An*- Analyst
- *D*- Designer.
- *Ds*- Documentation Specialist
- *M* - Maintenance Engineer.
- *P*- Programmer.
- *Pm* - Project Manager.
- *Qc* - Quality Control Engineer.
- *Scm* - Software Configuration Manager.
- *Test* - Test Engineer.
- *Vv* - Validation & Verification Engineer.

Files:

- *Contents.html.* Main index of the Home Page.
- *Introduction.html.* Introduction to the Home Page
- Lectures.html. Index of lectures.
- *index.html.* Frame definition file of the main page.

The following figure shows the file structure of DISEL.



Appendix A-Figure 1: DISEL file Structure.

DOCUMENT STORAGE.

The storage routine for the participants will be the one used to generate the project advance reports. The activities reports forms are included in the Home Page of the project (section activity-project progress). A reference to the report will be included in the database, once the participant fills and submits the form. In addition, a new document in HTML format is generated. In order to submit the progress report it will be necessary to provide a password, to deter use of the form by non-project members. The user name part of the e-mail address will be used as the password.

All the other documents will be stored by the documentation specialist. The purpose will be maintenance of a information and structure control of what is in the database and file system.

The documents generated by each one of the participants in the project will be sent to the documentation specialist who will include them in the database where they may be consulted by project team members. Document submission will be through e-mail with the documents included as attachments and indicating anything to be taken into consideration in the main body of the e-mail.

Once the document(s) are received, document references will be stored in the database. The document(s) will be verified so that it fulfills the structure and specified format. In case the document doesn't fulfill the format, it will be returned to the authors with a list of the corrections that need to be done.

QUERY FUNCTIONS.

The following functions are included for information query:

- *Documents search.* This will allow searching the documents by date, author, type of document (analysis, plan of work, etc.). These functions will be included in frozen documents and last version documents in the Documents section.
- *Audits search.* These will be included in documents-audits and will allow searching by date, the team that carried out the audit, document audited and/or the meeting in which the audit was carried out.
- *Meetings search.* These functions will be included in the sections of meetings-agendas, meetings-minutes and meetings-agreements. These functions will allow a search of these documents by date and by objective of the meeting.
- *Reports search.* These functions will allow searching the reports by activities, by date and by author.

These functions will be accessible to all the participants or to people outside the project.

MINUTE AND DOCUMENT OF MEETING'S AGREEMENTS.

MINUTES.

The minutes will be taken during the meetings of the project. These will highlight the issues of importance, questions and answers that occur during the meeting. These will be used to keep track of the meetings.

The minutes will then published so that the participants in the project can review them, make their comments, suggest changes, or otherwise approve them as they are.

At the beginning of the following meeting the proposed changes or comments will be reviewed. After that the minutes will be considered approved and designated as a final version. This final version will be included in the database and in the file system.

DOCUMENT OF AGREEMENTS.

During the meetings a series of agreements will usually be generated and require to be recorded. At the end of each meeting the documentation specialist will read this document so the entire team can review the agreements.

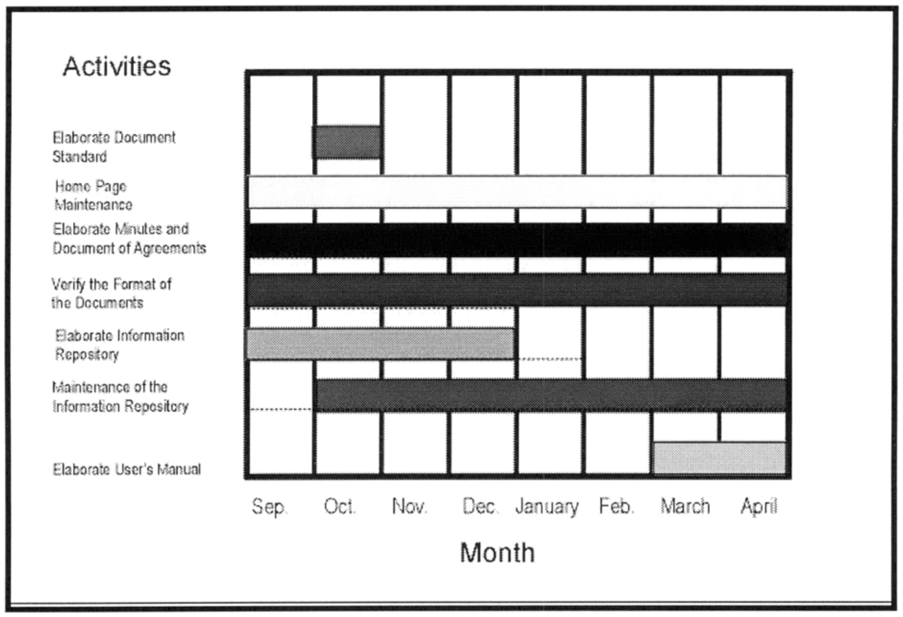
During the following meeting these agreements will be previewed, with the purpose of verifying if the proposed activities were carried out.
The final document of agreements will be published and stored in the database and in the file system.

USER'S MANUAL.

The user's manual is the document that will be used as a guide by the users of the system. This document will be developed almost upon project completion.

DOCUMENTATION SPECIALIST SCHEDULE.

The schedule of the documentation specialist is listed as follows.



Appendix A-Figure 2. Documentation Specialist Schedule.