A Framework for Software Reuse in a Distributed Collaborative Environment

by

Nhi Tan

Bachelor of Science in Environmental Engineering Science
Massachusetts Institute of Technology, 1999

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

May 2000
[June 2000]

Author..
Department of Civil and Environmental Engineering
May 5, 2000

Certified by..............................
Feniosky Pena-Mora
Associate Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by..
Daniele Veneziano
Chairman, Departmental Committee in Graduate Studies

A Framework for Software Reuse in a Distributed Collaborative Environment

by

Nhi Tan

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

Abstract

The current software development infrastructure does not support effective collaboration. The virtual team concept has not been fully utilized because dispersed teams experience barriers from limited technology. Each new cycle frequently begin with limited experience and knowledge from previous cycles, incurring high monetary and time costs. In addition, the development infrastructure isolates phases from each other, restraining inter-team contact between functional groups. To avoid the costs of limited collaboration and fully realize the benefits of increased cooperation in an engineering project, past knowledge and experience must be reapplied.

This thesis examines reusability in the software development process and the products generated in one development cycle. Reuse in supporting both temporally separated and geographically distributed collaboration in a development environment is discussed. ieCollab, a specific case distributed multi-year collaboration, ieCollab, strongly supports integrating reuse processes in the development cycle. This thesis clearly shows software reuse is collaboration.

Thesis Supervisor: Feniosky Peña-Mora
Associate Professor, Department of Civil and Environmental Engineering

ACKNOWLEDGEMENTS

I would like to thank Professor Feniosky Peña-Mora and the entire ieCollab 1999-2000 team who made this unique learning experience possible.

I would like to share my gratitude for all my friends who've made life interesting and worthwhile.

I would also like to dedicate this thesis to my parents and my sisters, Li, Ky, and Nina for their love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A large-scale engineering project's success often depends on the effectiveness of communication and cooperation among its collaborators. Collaborators may be split into teams and each team assigned to complete a different task. Teams may also be geographically distributed, especially in the increasingly global environment. Finally, compounding organizational and physical barriers, is an extended project lifetime. Projects can last several years and cycle through several iterations. Each cycle may begin with different collaborators with no memory of previous project versions. In order to maintain a consistent level of quality throughout the project's lifetime, the project team must leverage common resources between groups. The project team may achieve a feasible end-product delivered on-time and under-budget by sharing knowledge and experience.

Software development is a large-scale collaborative engineering project that depends on the cooperation of its developers. Projects are broken into phases, and different teams will be assigned to complete different phases. Members, both within one team or between teams, are increasingly dispersed. And projects cycle through several versions before a final product is released. Teams need to work together effectively to ensure successful software development.

The current software development infrastructure does not support effective collaboration. Phases are isolated from each other and inter-team contact between phases is limited. The virtual team concept has not been fully utilized because dispersed teams experience barriers from limited technology. Each new cycle frequently begin with limited experience and knowledge from previous cycles, incurring high monetary and time costs. To avoid the costs of limited collaboration and fully realize the benefits of increased cooperation in an engineering project, past knowledge and experience must be reapplied.

1.1 Purpose

This thesis examines reusability in the software development process and the products generated in a cycle. The focus will be on reuse in supporting collaboration, both temporally separated and geographically distributed, in a software development environment. Following will be a case study of software reuse in ieCollab, a development project in a multi-year, distributed environment. This thesis will clearly show reuse is necessary to collaborative engineering.

1.2 Motivation

The motivation behind this thesis is to suggest improvements in product and process for a smoother transition between dispersed workgroups and chronologically separate development teams. Valuable knowledge and experience is often lost between cycles and phases as well as within the project team. Transitions can be streamlined by integrating reuse engineering in the software development cycle. Implementing a strong reuse framework can also be used to effectively facilitate knowledge sharing between dispersed groups.

1.3 Reuse Engineering

The Institute of Electrical and Electronics Engineering defines reuse as "the use of an asset in the solution of different problems" (1999). Software reuse is solving an existing problem with previous software knowledge (Biggerstaff and Perlis 1989). Reuse engineering describes a complete software development cycle that produces new software by reusing already existing components (Gall and Klosch 1992). Software reuse may include, among others, document, planning, test cases and contract reuse. Reuse engineering issues can be divided into organizational, module, user query, domain, and methodological subtopics (Gall and Klosch 1992). Figure 1-1 below delineates issues in reuse engineering based on these categories. The first three topics define the code. Organizational issues include the classification and attribution of reusable components. SCL represents the available software components libraries. Module refers to issues in producing reusable components and using existing modules in development. User query topics cover the retrieval of software components. Methodological issues examine complications and solutions in integrating reuse to the development cycle. Finally, domain aspects describe the end-product.

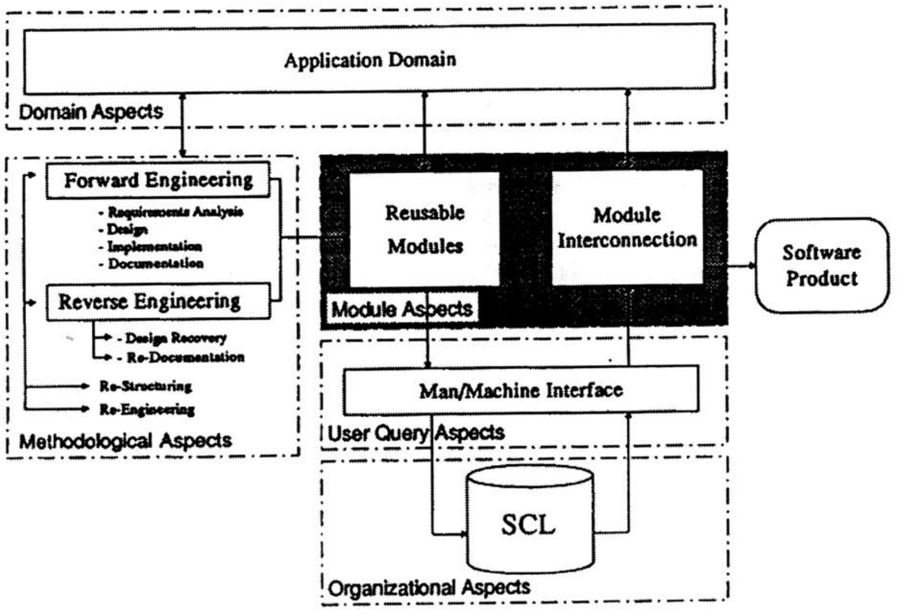


Figure 1-1. Reuse Engineering Domains (Gall and Klosch 1992)

This thesis takes an expansive view of reuse engineering. A limited reuse program, such as restricting reuse to code and design, has shown only a small return on investment (Biggerstaff and Perlis 1989).

1.3.1 Definition

Reuse is applying an existing asset to solve different problems (IEEE 1999). Software reusability is the degree to which an asset can be used in more than one software system, or in building other assets (IEEE 1999). In a reusable system, reusability is the characteristics of an asset that make it easy to use in different contexts, software systems, or in building different assets. Systematic reuse is the practice of reuse according to well-defined, repeatable process (IEEE 1999).

An asset is any item, such as a design or test plan, that has been designed to be used in multiple contexts, such software products, multiple implementations of a software product or multiple software projects (IEEE 1999). An asset is either fine grain or large grain. Fine grain components are code modules or instantiated classes. Large grain components may be as large as third-party vendor provided products. In reuse engineering, these assets form a composite end-product.

Degrees of reuse of an asset may also vary between organizations. Black box reuse reapplies existing components unmodified. Gray box reuse makes slight modifications to a component before using the component. White box reuse performs major modifications to a component before modification.

Definitions of terms are provided below for organizational clarity and reader understandability. All definitions are in the context of this thesis and may be used differently outside this space.

- Domain – The problem space.
- Project Team – All members of the current development team.

- Functional Team – Member of a particular development phase.
- Work Products – Assets generated from software development cycle that may include design diagrams and test cases. Document assets such as functional team plans are not included.
- Functional Products – Assets generated to support the software development cycle that includes teams plans.

1.4 ieCollab

The **I**ntelligent **E**ngineering **Collab**oration project (ieCollab) is based on a distributed course taught at the Massachusetts Institute of Technology (MIT) in Massachusetts, Centro de Investigación Científica y de Educación Superior de Ensenada (CICESE) in Mexico, and Pontificia Universidad Catolica (PUC) in Chile. The course, Distributed Development of Collaborative Engineering Support System, covers the software development cycle, collaboration and organizational strategies, entreprenuership, collective memory and technology. Lectures are presented from professors in each of the three locations in rotation utilizing the latest technology in distributed communication. Assignments are completed in groups composed of students from all three universities, which models distributed collaboration.

Lectures and distributed team assignments were designed to familiarize the participants with distributed engineering and the problems encountered in this form of collaboration. Based on this experience, students then begin to solve the problems faced in distributed collaboration in the project. ieCollab is the final product of this research project.

ieCollab is an application that enables collaboration between geographically dispersed parties in a virtual environment. The project builds on research performed in previous years and incorporates earlier solutions in the design. As a result, architecture and technology vary between versions as well as between geographic locations. This difference may be the result of development habits, inherent design structure, or coding practices. These same fundamental differences impede the use of previous versions of software. As a result, each new cycle or distributed development of the same application requires completely new code, structure, and design. Development process recommendations are not passed to the new cycle. Consequently, software development incurs immense cost and time delay. These consequences are avoided by applying reuse techniques throughout the application development and between life cycles.

1.5 Overview

This thesis assumes no prior knowledge of reuse theory and practice or ieCollab. The focus is on reuse in a collaborative environment. General reuse application is not discussed in detail.

Chapter 1 introduces the thesis and outlines the flow of the following chapters. This chapter will provides a brief overview of the issues covered.

Chapter 2 gives a brief overview of CAIRO and ieCollab, the team members involved, and the team structure. Chapter 2 serves to remind the reader the problem paradigm and background domain addressed by this thesis.

Chapter 3 presents in depth the case for reusability. Both the costs and benefits are considered. This chapter will show that although the initial cost of reuse is high, the return on investment and non-monetary benefits achieved clearly outweigh the cost.

Chapter 4 introduces basic principles and techniques of software reuse and the current technology. This chapter serves to provide enough background for the user to understand and apply reuse ideas presented in a collaborative environment.

Chapter 5 is a case study of the research behind this thesis. The case study presents in depth ieCollab and reusability as supporting a multi-year, distributed environment. The software methodology and the end-products are discussed. Based on the problems encountered by the project team, recommendations for reengineering both applications are presented. The objective of this chapter is to present a strong case for formalizing reuse activities in the project development cycle.

Chapter 6 concludes this thesis with recommendations for formalizing reuse in a collaborative engineering. This chapter presents a formal structure to reuse and technology supporting activities. The benefits of reuse in ieCollab are also discussed.

Chapter 2

Background

This thesis is based on the work completed in the **Di**stributed **S**oftware **E**ngineering **L**aboratory (DiSEL) and **I**ntelligent **E**ngineering **Collab**oration (ieCollab) projects. The project seeks to improve collaboration of geographically dispersed teams through Internet-based communication tools. The goals are to enhance the software development process to facilitate effective engineering regardless of location.

2.1 Project Motivation

The motivation behind this project is to extend the **C**ollaborative **A**gent **I**nteraction and Synch**ro**nization (CAIRO) software project, the original research of Karim Hussein, Sc.D. (Hussein, 1998). Dr. Hussein investigated the role of computers in supporting distributed design teams. His work was implemented through the 1997 and 1998 CAIRO teams and the 1999 ieCollab team, whose members are composed of students from the Massachusetts Institute of Technology (MIT) in Cambridge, Massachusetts, Centro de Investigacion Cientifica y de Educacion Superiro de Ensenada (CICESE) in Mexico, and the Pontificia Universidad Catolica (PUC) in Chile. The geographically dispersed nature of the team and the waterfall model of software development followed by each team are also a case study to support the work of Dr. Hussein. Students in CAIRO develop software applications for distributed collaboration in a distributed environment, effectively becoming customers of the application.

2.1.1 CAIRO

The CAIRO project was initiated at MIT in 1995. CAIRO is a platform-independent application enabling geographically distributed collaboration in a virtual environment. CAIRO incorporates communication technology and meeting control processes to simulate an actual meeting and collaborative environment. CAIRO specifically aims to include human interaction and simulate human expression and reaction in its design to increase the realism of the virtual setting (Hor 1999).

CAIRO was introduced to the DiSEL class as a software engineering project in 1997. The 1997 CAIRO team enhanced personal interaction through the addition of facial expression capability. The main focus of that group was computer aided distributed learning. The 1997 CAIRO team members were: Kareem Benjamin, Humberto Chavez, Juan Contreras, Gregorio Cruz, Juan Garcilazo, Lidia Gomez, Sergio Infante, Felix Loera, Ruben Martinez, Rene Navarro, Charles Njendu, Marcela Rodriguez, Simoneta Rodriguez, Diana Ruiz, Christine Su, Tim Wuu, and Bob Yang.

The 1998 CAIRO team enhanced the social interaction features introduced by the 1997 team. This team created a 3D work setting, with people and objects in 3D representation. The motivation behind the 1998 project was social interaction and casual contact in collaboration in a virtual setting. The 1998 CAIRO team members were: Ricardo Acosta, Juan Contreras, Kiran Choudary, Gregorio Cruz, Alberto Garcia, Octavio Garcia, Joon Hor, Cagaln Kuyumcu, Gregoire Landel, Rafael Llamas, Jaime Solari, Sanjeev Vadhavkar, Padmanabha Vedam.

The 1997 and 1998 CAIRO teams members were geographically dispersed and collaborated through available technology mediums. Students used Internet chat programs, email, and CAIRO itself as mediums to substitute for collocated interaction. Working groups within the team were structured to encourage collaboration by having members of a group from both MIT and CICESE. In addition, each team built on work of the previous year or of previous research.

2.1.2 ieCollab

The purpose of the 1999 ieCollab team was to apply the procedures developed for distributed learning inherited from previous teams to include even more geographically dispersed members. This year's team includes members from PUC in Chile. The 1999 ieCollab team members were: Erik Abbott, Manuel Alba, Joao Arantes, Hao Chen, Gyanesh Dwivedi, Wassim El-Solh, Octavio Garcia, Kaissar Gemayel, Cesar Guerra, Hermawan Kamili, SaeYoon Kim, Bharath Krishnan, Chang Kuang, Steven Kyauk, Li-Wei Lehman, Ivan Limansky, Teresa Liu, Kenward Ma, Roberto Macchorro, Anup Mantena, Justin Mills, Alberto Moran, Alan Ng, Blanca Roman, Sugata Sen, Jaime Solari, Nhi Tan, Eswar Vemullapali, Pubudu Wariyapola, Paul Wong

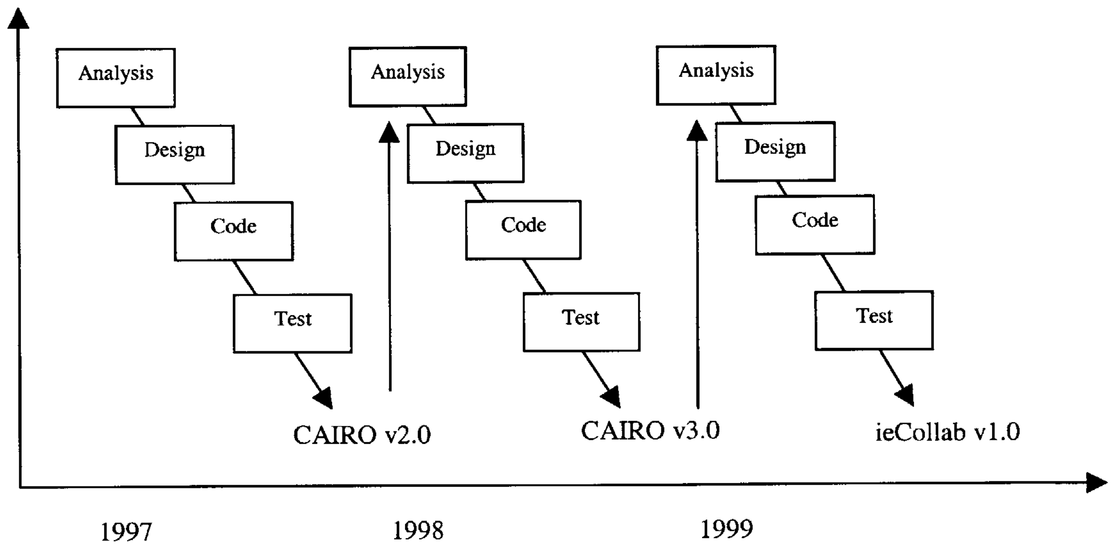


Figure 2-1. Incremental model of software development and deliverable by year.

This year's project saw a fundamental shift in the application paradigm. Whereas previous years developed a software product to solve distributed collaboration, this year's team members formulated the solution as a technical service. The project became a service problem. We attempted to incorporate CAIRO as an application service. The team's project goal is to develop a comprehensive suite of services to solve the problem of geographic barriers, and the cost effects associated with this issue, to enable distributed collaboration.

2.2 Team Structure

The team structure was organized according to the phases of a software development lifecycle. Members chose from a variety of roles, including: business manager, marketing manager, project manager, requirements analyst, design engineers, programmer, quality assurance engineers, configuration manager, testers, and knowledge manager. The project team strove to attain geographic both across and within functional teams. Each group was composed, as much as possible, of people from all schools. The 1998 CAIRO team was composed of three students from CICESE and ten students from MIT. The 1999 ieCollab team was composed of five students from PUC, six students from CICESE, and 23 students from MIT.

2.3 Summary

This project models a real-world software development lifecycle during a nine-month period. Geographically dispersed teams worked together to develop a solution that would facilitate distributed engineering and design. Each team's efforts built on a product from the previous development cycle. The multi-year development time and virtual environment of this project supports reuse engineering as a method to improve the development process and the final end-product. The next section will explore the benefits gained from integrating reuse engineering in collaborative software development.

Chapter 3

Software Reuse

The previous chapters introduced the motivation and background behind this thesis. Chapter 3 introduces the theory of software reuse and why reuse is necessary to a collaborative software development. This chapter first describes the current software production environment. It then presents a strong case for reusability in the development cycle, examining such factors as cost and technology. Finally, a general overview of reuse activities is presented before concluding with common barriers to software reuse.

3.1 Current Industry

Advancements in software technology in the last thirty years have been unable to close the widening gap between demands on the industry and the capability of the practice (Mili, Mili, and Mili 1995). Although years of research have refined the development process, current practices and activities have not yielded significant improvements in productivity or quality (Biggerstaff and Perelis 1989). In addition, improvements in supporting technology and tools have not been able to improve dramatically the ability to develop software (Biggerstaff and Perlis 1989). Finally, the human resource demand is increasing, yet the number of skilled professionals able to fill those roles remains unfulfilled. As problems become more complex and applications require more maintenance, current practice and activities will be unable to meet long-term demands for low-cost, high quality software (Mili, Mili, and Mili 1995).

3.1.2 Capability Maturity Model

Unmet demand is also compounded by the industry's immaturity in development as a whole. The maturity level of an organization is measured by how evolved the organization is. The Capability Maturity Model, shown in Figure 3-1, is a benchmark for assessing an organization's ability to formulate and apply a software development process.

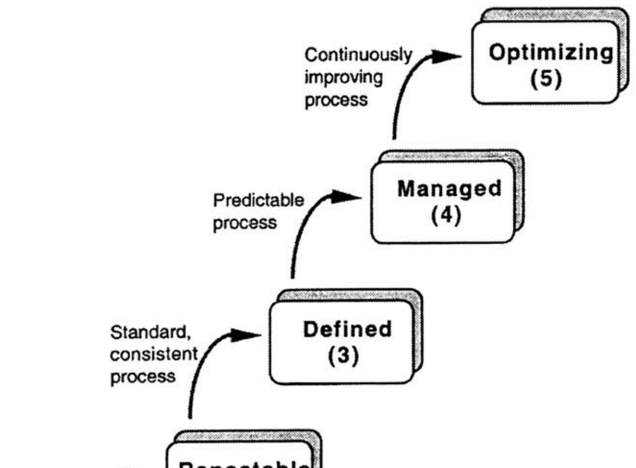


Figure 3-1. Capability Maturity Model (Paulk, Curtis, Chrissis, and Weber 1993)

3.1.2.1 Maturity Levels

Maturity levels indicate process capability and contain key process areas that define objectives of that level. Common features that address implementation or institutionalization organize process areas. Features contain keypractices that describe the development infrastructure or activities.

Five maturity levels, described in Table 3-1, are milestones in a software organization's evolution. The primary process change describes activities that should occur at each level.

Level 1, the initial level, organizations typically lack a structured development environment, reinforcing ineffective planning and reaction-driven processes. In a crisis, organizations abandon process structure and merely code and test. Only extremely effective managers and experienced programmers may be successful at this level. Activities are not stable or set at this level. Thus, an organization's performance is unpredictable at this level because processes are spontaneous and not stable.

The repeatable level, level 2, reuse effective process by implementing policies and procedures that institutionalize best management practices in software development. Effective processes are practiced, documented, enforced, trained, measured, and improved. At this level, organizations reuse effective processes, though in a slightly modified implementation than previous. Project schedules, costs, and productivity are tracked. Basic work products and requirements are formalized. Thus, level 2 organizations are capable of disciplined processes.

Level 3 is the Defined level. Development and management processes are defined and documented as feedback to the developers and managers for process improvement in the next cycle. In level 3, reused processes are tailored to the project with well-defined requirements. Readiness criteria, verification mechanisms, outputs, and completion criteria distinguish well-defined processes. Thus, level 3 organizations are able to predict capabilities and productivity.

At level 4, the managed level, organizations establish quantitative values for process objectives and products. Organization-wide database collects and analyzes software process data, which is used to determine variations and minimize risk in the next development cycle. Thus, management achieves control over development and output. Level 4 process capabilities are predictable because process achievements are quantified and quality is measured.

The optimizing level, level 5, is oriented towards process improvement. Process measurements in this level are used to identify defects and their causes and propose change. Level 5 analyzes costs and benefits of new technology and innovations are spread throughout the organization. Process capability of this level is continuously improving because the range of this level's capability is constantly seeking improvement, thus improving the process itself.

Table 3-1. Capability Maturity Model levels. (Paulk, Curtis, Chrissis, and Weber 1993)

Maturity Level	Primary Process Change
Initial	The software process is characterized as ad-hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.
Repeatable	Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
Defined	The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
Managed	Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
Optimizing	Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

The capability maturity model is used by organizations to predict process performance. Level. Figure 3-2 shows process capability as a prediction of performance. The ability of a project to meet its goals depends on the level of maturity of the organization. An organization that has achieved a higher maturity level is more effective in determining performance and minimizing risks. Level 5 organizations experience less variation in achieving cost, schedule, functionality, and quality targets. As maturity increases, organizations are better able to predict project results. In addition, variability around targeted results decreases with increasing maturity. Third, as software organizations mature, targeted results are more obtainable because projects are incrementally obtaining targeted results. Thus, the capability maturity model is an indicator of the development process and output quality.

The majority of software organizations have not reached the third level. None have achieved the fifth level (Kocur 1999). High-level software organizations have not evolved to efficiently create complex software solutions to existing challenges.

3.1.2.2 Reuse in the Capability Maturity Model

Reuse activities appear in all levels of the capability maturity model. Ad-hoc reuse of process and code occurs in Level 1. In a crisis, reuse is ignored due to the preparation involved in understanding another person's work. At level 2, development processes are repeated over several projects. Work products are reused informally but are not refined until level 3. At level 3, feedback improves available products or processes and modified for the next development cycle. Organizations may implement a formal structure for product reuse. At level 4, modified repeatable processes are standardized for use across organizations or other projects. A software development library is integrated in the reuse process to facilitate reuse among developers. At level 5, reuse engineering is integrated completely in the process cycle and new products are created and designed specifically for the next project.

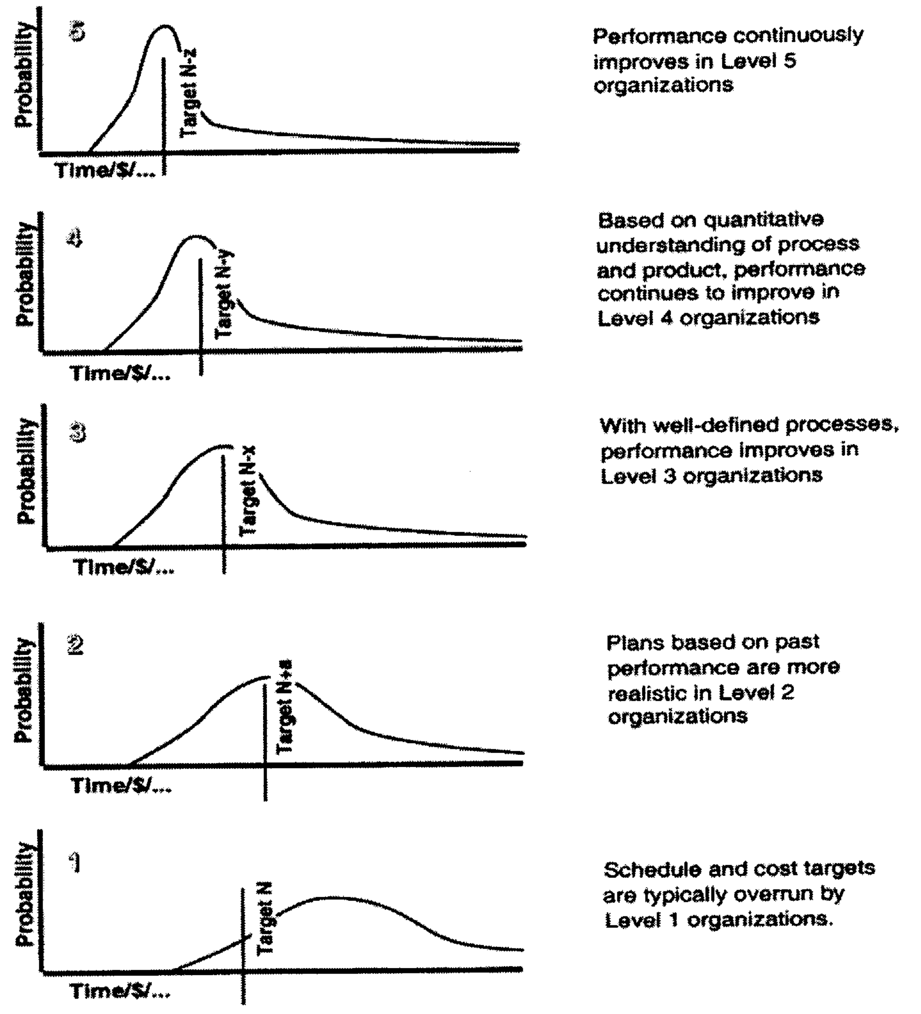


Figure 3-2. Predication of performance (Pressman 1999).

3.1.3 Support for Software Development

Organizations are hesitant to support technology development because of the high initial cost and the low return on investment. Few companies have the resources to finance advanced software development projects. Most projects are designed to provide one specific solution for one specific problem. In projects that are funded, companies must often wait a long time before realizing technology assets. As a result, software development is low priority to many companies and lack adequate support. Thus, current technology has provided only marginal improvements with diminishing returns (Biggerstaff and Perlis 1989).

3.1.2 Development Redundancy

Software productivity must increase by an order of magnitude to relieve pressure on the industry (Mili, Mili, and Mili 1995). The gain in productivity will only be realized when the industry discovers a method to write less code, not more code faster. Guetari in 1995 reported only fifteen percent of code written each year worldwide is original and 85 percent of new code already exists. Zand and Samadzadeh estimate that 40-60 percent of code is reusable from one application to another and 75 percent of program functions are common to more than one program. Only fifteen percent of code is unique to one specific use. Most coding projects are redundant to what has been created. Arbaoui and Oquendo estimate that the majority of software products could have been built with assets, had they been available (1997). Software reuse is the only practical approach to meet this challenge.

3.2 The Case for Reusabilty

Reuse engineering, as defined in chapter one, is a complete software cycle that integrates already created components into development and produces assets that can be reused in another development cycle. When implemented correctly and consistently practiced, software reuse incurs both immense costs and immense benefits.

3.2.1 Cost of Reuse

Incorporating reuse activities in the software development cycle is costly. Reuse engineering is capital-intensive from creating reusable assets, reusing assets, and defining and implementing a reuse process, activities which require additional development costs. Lim in 1994 estimates the cost for creating reusable assets is double the cost of developing conventional components. Tracz estimates the same cost to be a minimum of 60% of the cost for developing software solutions (1994), broken down by assets shown below in Table 3-2.

Table 3-2. Percent increase in costs due to making software reusable (Tracz 1994).

Additional reuse activity	Percent cost increase
Generalization	25%
Documentation	15%
Testing	10%
Library support and maintenance	10%
TOTAL	**60%**

The cost of using reusable assets is also great. Integrating reused components into new products is ten to 20 percent of the cost of creating an application without existing assets (Lim 1994). Favaro in 1990 offered a wider cost range in integration, providing an estimation of between ten and 63 percent. Process reuse incurs both time and monetary costs. Process reuse includes all activities supporting the development of reusable or reuse of software. Direct process costs is difficult to measure, but can be inferred based on resource efforts. The breakdown in life-cycle phases and engineering months increase from using and creating reusable assets is in Table 3-3, which compares two different estimates.

The total increase in engineering months is estimated to be 20% (Lim 1994). Margano and Lindsay found a total increase in engineering months to be 40% (1991). Both estimates determined initial analysis and design phases to incur the greatest time cost. One possible reason is that designing reusable software requires consideration of all contexts an asset will be used, and integrating reusable components requires intimate knowledge of the asset. Thus, preparation time before a components is used or created is vital yet expensive.

Table 3-3. Percent increase in engineering months due to reuse (Lim 1994).

Development process activity	Percent effort increase (Lim 1994)	Percent effort increase (Margano and Lindsey 1991)
Investigation	22%	13%
External design	20%	60%
Internal design	0%	0%
Code	17%	20%
Test	5%	5%
Repair	5%	0%
TOTAL	**20%**	**40%**

3.2.2 Benefits of Reuse

Although the cost to reuse maybe great, the benefits from implementing and practicing a strong reuse program are great. Reuse engineering returns direct and indirect monetary, development, organizational and social benefits.

Direct benefits are monetary return on investment in reuse program. However, the return on investment of integrating reuse engineering into an organization's lifecycle is not easily measured. The lifecycle of reusable and reused assets is difficult to determine. Assets may be used in several projects or maybe modified to create a new asset. Thus, expensive development costs for new projects are replaced by one expensive development cost and several inexpensive reuse activities (Zand and Samadzadeh 1994).

In addition, an asset's development costs can be reduced if new products and new code are produced based on the asset. Consequently, investment in these assets is a corporate investment that can be amortized over several projects (Frakes 1994).

Reuse engineering provides indirect cost benefits as well. Reuse engineering incurs shortened development time because components are recycled (Zand and Samadzadeh 1994). Components, already coded and tested, would thus greatly reduce the estimated 70% effort in these two phases. In creating reusable parts, the cost of prevention and debugging can be amortized over a greater number of uses (Lim 1994). The "ripple-effect" that occurs when code is changed normally incurs high costs from re-coding, re-testing, re-integrating, and re-debugging. A reusable component, high-level in nature, reduces life-cycle maintenance cost by reducing "ripple effect" caused by changing code. As a result, productivity improves inherently because less input is required for more output (Lim 1994). Products are on the market and available in a shorter time, thus providing a competitive advantage to the organization (Neighbors 1994). Non-specific assets are easily transferable to other industries or markets, increasing the organization's customer base (Neighbors 1994). Neighbors estimates the sum total savings in effort from reuse activities is 34% (1994).

Reuse engineering incurs non-monetary benefits as well. In addition to productivity gains, reuse also provides benefits related to product quality. Overall quality improves because quality-enhancing processes are recycled and systematized in the development process (Mili, Mili, and Mili 1995). System testing needs is reduced and system reliability increases with the number of assets reused. Reusable development is better predictor of effort and time expenditures during development because many of the assets already exist. A reuse-sensitive system thus manages inherent risk by uncovering unknown factors in earlier asset development.

Organizations mature with reuse activities because resources and engineer efforts are inherited and optimized. As shown in figure 3-2, mature organizations are better able to predict performance.

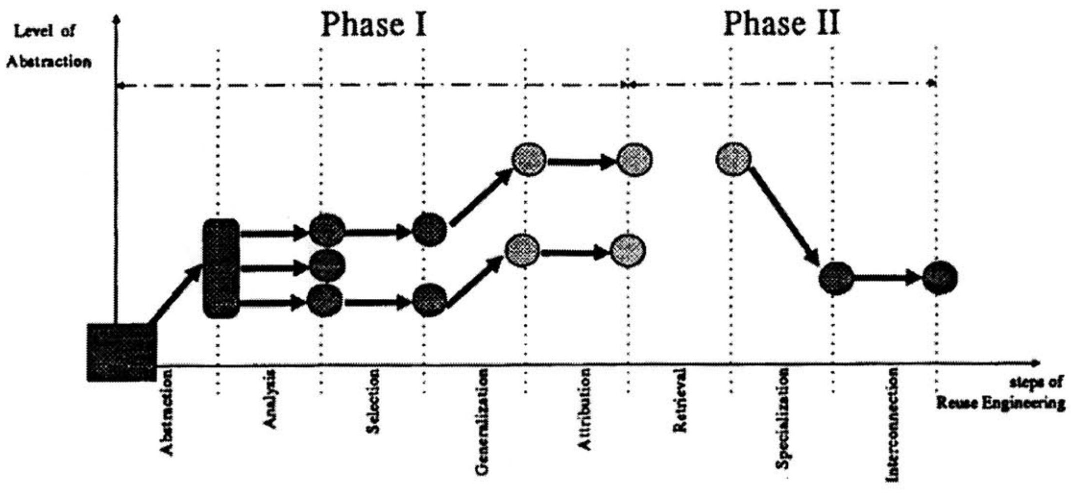


Figure 3-3. Abstraction in reuse engineering (Gall and Klosch 1992).

3.2.3 Technology

The greatest indirect cost benefit from reusing assets, however, may be from its ability to adapt to emerging technology. The rapid pace of changing technology constantly replaces old technology sometimes even before old technology is widely accepted. As a result, constant upgrades and modifications to past versions of an application incur great redevelopment cost. Creating reusable components extends the lifetime of each asset, and can be reused constantly because components are able to accommodate to changing technology. Software reuse also allows software interoperability between existing assets (IEEE Standards 1999).

3.3 Spectrum of Reusability

Reuse engineering includes all activities that apply prior software development knowledge. Past efforts mainly focused on creating a reuse library of code pieces. Development teams need to reuse more than code to gain benefits mentioned above (Frakes 1994). The greatest benefits will be gained not in salvaging existing code, but implementing a deliberate reuse program in the development process.

3.3.1 Reuse Theory

McIlroy introduced the idea of formal software reuse in 1968 as a method to recycle old software code. He advocated the development of a reusable source code industry and an automated method to manufacture software applications. Proponents recognized difficulties in software development within large groups. Early reuse engineers attempted to piece together old code to form a functional application with inconclusive results. The idea of reuse has evolved since then to include all knowledge contributing to and produced during software development.

Reuse is founded on the belief that the current software development process, as an engineering field, is not an optimal method to solve problems. Engineering disciplines are based on reapplying experience and knowledge from previous work to a new problem. For example, in civil engineering, bridge designers rarely create a unique bridge design in every project; often past bridge designs, proven to be reliable and stable, are recycled in subsequent projects. Reusing experience and previously acquired knowledge transforms software development into software engineering (Biggerstaff and Perlis 1989).

Reuse is a fundamental reorganization of typical team structure and software development environment. John and Spiros-Theodoros state the three basic premise of software reuse as:

- Most products and procedures in a software development cycle have the potential for reuse.
- Reuse is dependent on the specific structures, phases, and workflow of a lifecycle because of the added activities required in each phase. Complete reuse must be made adaptable to any type of lifecycle.
- Objects are reused in different ways depending on the problem. Maturity Levels are needed to define the correct application of an object.

3.3.2 Reuse Framework

A complete reuse framework encompasses the entire software development process, but modified either the inputs or the products with a reuse trait. Biggerstaff and Richter divided the software reuse program into the following phases: Abstraction, Selection, Specialization, and Integration (1989). Abstraction determines how assets are identified, classified, and interrelated. Selection outlines a method to search for a reusable component. Specialization includes activities to customize a part for specific instantiation. Integration includes steps to compose an application from components for specific use.

Reused application domain knowledge is represented as domain models. Domain model is a formal method to identify, store, and organize information created in or by a software development process for reuse engineering (Zand and Samadzadeh 1994). Mili, Mili, and Mili identify three main purposes of domain models.

- Identify the purpose of the domain
- Provide application-dependent categorizations
- Act as the starting point for systems analysis

Models should clarify entities, the entity operations common to the application domain, and relationships and constraints connecting entities. In addition, domain models should set properties of the entities that allow for easy search, such as key words or other property cues.

Arango in 1994 proposed that domain models be self-improvement processes. His domain model includes a feedback mechanism in which the reuse infrastructure encompasses the conventional software development process and is modified for the next cycle. Within the domain model are activities to define boundaries, or the extent of reusability, in creating reusable assets and procedures to identify opportunities to reuse existing components. Domain analysis is the part of the descriptive modeling process in Sodalia's Object-Oriented Domain Engineering methodology (SOODEM), shown in Figure 3-4.

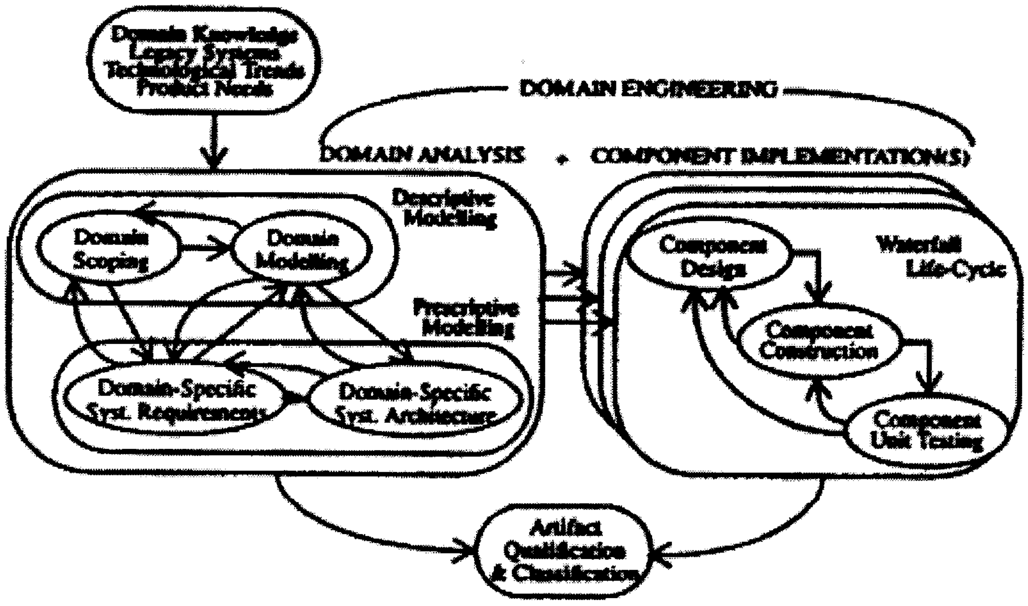


Figure 3-4. Domain engineering flow diagram (Doublait and Lissoni 1997).

Figure 3-4 shows the flow of assets through a reuse system. In the model, domain activities occur prior to asset creation and implementation. SOODEM accounts for both creation and instantiation of reusable assets. Creating reusable assets, either from reengineering an existing asset or without any prior resource, takes the following path:

1. Input any existing resources into the system.
2. Examine the problem domain for reuse opportunities.
3. Create the component to solve the problem domain.
4. Examine the solution for potential reuse assets.
5. Abstract and Select assets for reuse.
6. Archive assets into the library.

Integrating reusable assets follows a similar path.

1. Examine the problem domain for reuse opportunities
2. Select assets for reuse
3. Specialize the asset for customization in this problem domain.
4. Integrate asset into existing solution.

3.3.3 Defining Assets

Freeman classified reuse assets into five levels of software development knowledge (1987). Freeman's hierarchy corresponds to domain-influenced development. The last three levels, representing the design and coding phase, are functional architectures, logical structures, and code fragments. The first two levels, environmental knowledge and external knowledge, correspond to system specifications from user requirements. Environmental knowledge encompasses technology transfer knowledge, which describes the business context in which the software product will be used. External knowledge covers the development process, including the planning and management, and includes knowledge of the underlying models for the application domain.

Although no standards for knowledge resource categorization exist, most proposed models use three factors in characterizing assets:

- development stage at which the resource will be produced or used
- current level of abstraction

- nature of knowledge

Past research recognize three general classes of artifacts:

- reusable program patterns
- reusable processors
- reusable transformation systems (Mili, Mili, and Mili 1995).

Reusable program patterns are used to instantiate specific design cases and specific code pieces. Reusable processors are interpreters for high-level specifications. Reusable transformation systems embody developmental activities that transform and translate descriptions of assets to another language. Krueger in 1992 proposed a multi-level hierarchy of reuse knowledge. Level i reuse assets are abstractions of level i-1 assets (Figure 3-5). Hierarchical organization levels enables reuse activities as varying as code search and high level design languages.

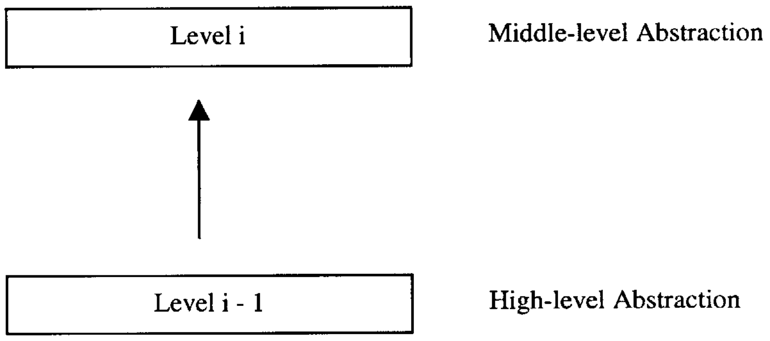
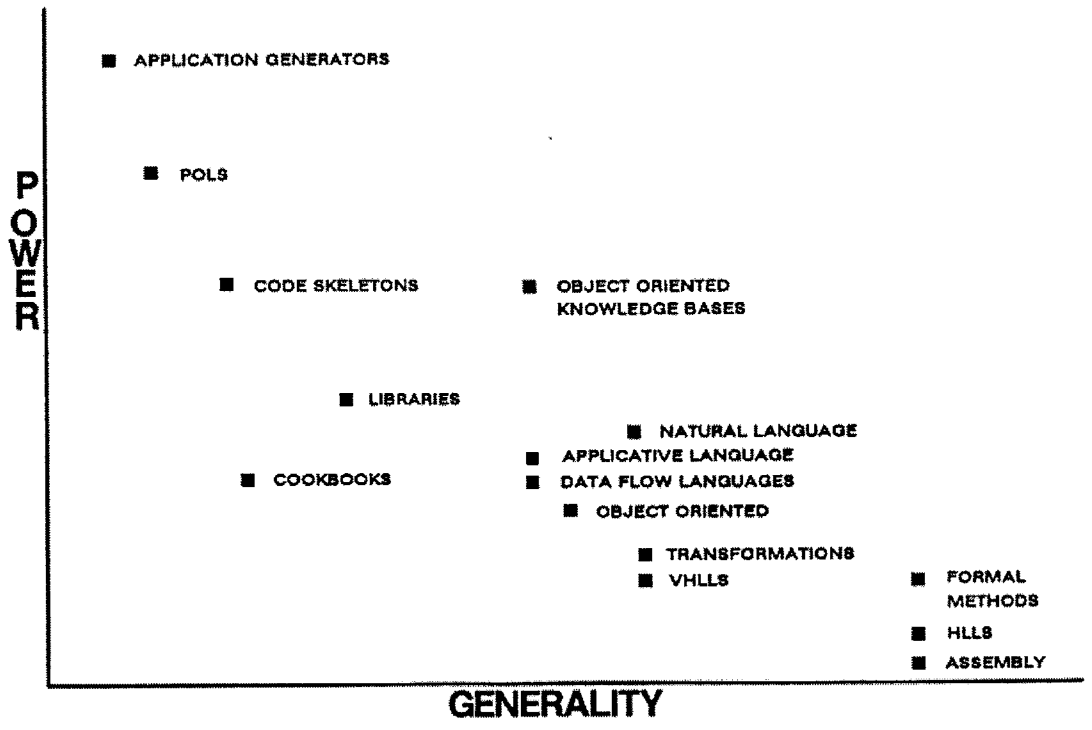


Figure 3-5. Hierarchical organization of reuse assets.

Contained within each layer are instantiated reuse assets. Asset bounds are affected by the change in parameters; a positive change in one parameter leads to a negative change in another (Biggerstaff and Perlis 1989). Figure 3-6 below displays the generality and specificity space. Generality is the degree of abstraction and is on the horizontal axis. Power is the ability of an asset to solve a problem and is on the vertical axis. One of the main dilemmas in reuse is creating assets general enough to apply to several applications, but specific enough to be functional. Application generators, which automates the creation of software programs, is high in power because the generators create functional programs based only on user requirements as input with no modification to the original code. However, application generators solve specific problems. As a result, generality and the degree of reuse to another problem are low. For example, payroll programs are intended to facilitate payroll processing. The application can be easily adapted to any organization's payroll model. The design and code, on the other hand, is not easily transferable to word-processing applications.



Figures 3-6 Generality versus power (Biggerstaff and Perlis 1989).

The second dilemma in creating reuse assets is deciding the component size. Component size is linearly related to payoff (Biggerstaff and Perlis 1989). The larger the asset, indicating a high-level complexity, the greater the payoff in reuse is expected. As the size increases, however, specificity increases. In specific functionality, the number of chances to reuse the component decreases. To be able to recycle the component, modifications must be made. Thus, to prevent cost increase, a balance must be established between component size and a reasonable payoff potential.

3.4 Barriers to Reuse

Reuse technology may solve the productivity problems in software development, yet many organizations fail to successfully reuse knowledge (Card and Comer 1994). Organizations continue to view reuse as a salvaging process, a passive method that occurs after core development. Card and Comer recognize economic, cultural, and technical factors contributing to reuse failure (1994). These factors contribute to an inherently problematic view of reuse engineering that prevents organizations from realizing the full potential of reuse.

Technology barriers occur mainly at the development level. Inadequate tools to search for available components hinder integrating existing knowledge to the present problem. Representation methods for abstraction and high-level development are not sufficient to fully characterize a knowledge, which discourages building reusable assets. Also, architecture mismatch obstructs effortless integration between reusable design and code assets. The popularity of graphical user interfaces, databases, networking, and multitasking complicates the interface between knowledge assets and asset reusability (Neighbors 1994). The current state of reuse technology encourages reuse on a small scale to a very specific solution. Thus, organizations treat reuse as a technology-acquisition problem and not a technology-transfer (Card and Comer 1994).

Economic barriers occur mainly in the managerial level. Because a functional reuse program requires a critical mass of components and a great number of organizations to make use of these assets, developing reuse engineering calls for a high initial capitalization. The time delay after developing such a program also incurs heavy costs. Consequently, managers are unwilling to initiate a reuse program without a substantial evidence of quality benefits. Without managerial support, a strong reuse program is low priority and lacks funding.

Finally, cultural factor is not confined to one level in the development cycle. Cultural barrier describes an environment where reuse is overlooked or discouraged (Biggerstaff and Perlis 1989). Developers display a hesitance or lack desire to learn and use another developer's work, also known as the Not Invented Here (NIH) factor (Card and Comer 1994). Developers trust and intimate understanding of personally created knowledge assets also contributes to NIH. At the management level, managers' inability or disregard for reuse engineering bolsters NIH. Resources invested in creating new assets for each project limits efforts to solve more challenging problems or increases development time.

3.2 Summary

Chapter three introduced the idea of software reuse. A strong reuse program includes recycling all knowledge assets created or produced in the software development cycle. Reuse engineering has the greatest potential to increase productivity while decreasing cost and development time. Cultural, economic and technology barriers need to be overcome before an organization is able to implement a strong reuse program. Implementation and specific reuse activities in the software development process are discussed in the next chapter.

Chapter 4

Reuse in Collaborative Software Engineering

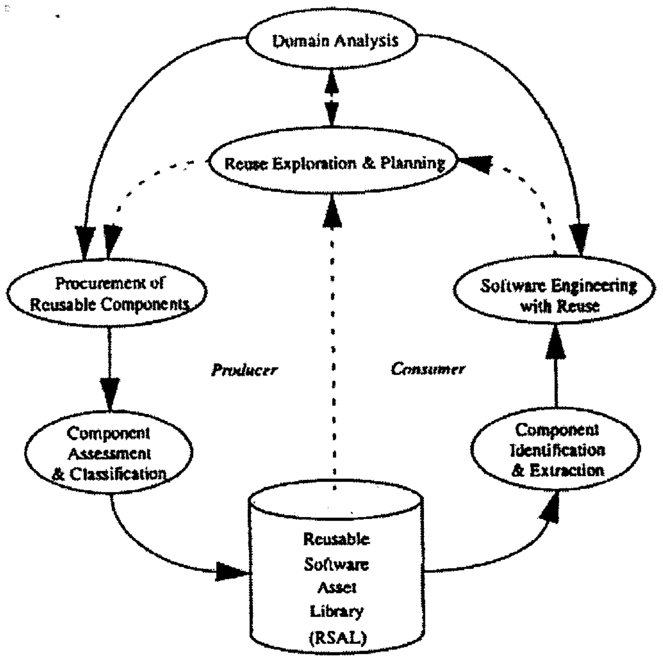
Software reuse is more than creating and reusing component libraries. A formal method integrated in the software development cycle is necessary for an organization to fully realize the reuse potential discussed in the previous chapter. Reuse activities in the design and code phases decrease development time and recycle resources, ultimately lowering development cost. Standardization of assets enables knowledge resources to be shared among development teams and inherited by teams responsible for the next phase of the software. Without repeatedly accessing existing resources, only limited benefits of reuse is realized. Chapter 3 introduced reuse engineering, discussed the economic and social trade-off of a reuse environment, and provided background information for reuse structure. In this chapter we outline a formal activities in each phase of reuse engineering.

4.1 Reuse Engineering Process

Software reuse illustrates the reuse engineering principal:

Reuse Engineering := [Reverse Engineering]Forward Engineering.

In summary, this principal states that reuse engineering is assigned to forward engineering and can be assigned to reverse engineering. Reuse engineering includes both reverse engineering and forward engineering activities. Figures 4-1 shows an overview of reuse process.


Domain Analysis
Reuse Exploration & Planning
Procurement of Reusable Components
Software Engineering with Reuse
Producer
Consumer
Component Assessment & Classification
Component Identification & Extraction
Reusable Software Asset Library (RSAL)

Figures 4-1. Overview of reuse processes (Gall and Klosch 1992).

The reuse engineering process defines activities for two specific reuse goals:

- Creating reusable assets.
- Integrating existing reusable assets in software development.

Each objective requires its own process model, shown in figures 4-2 and 4-3.

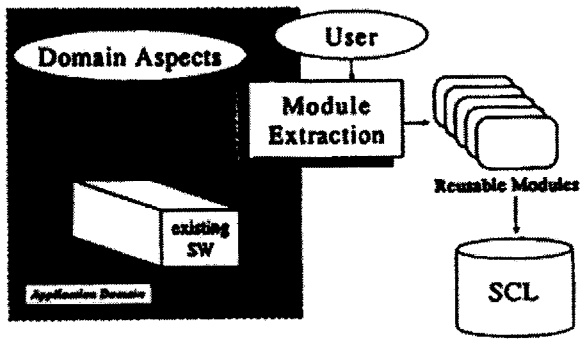


Figure 4-2. Reverse engineering model (Gall and Klosch 1992).

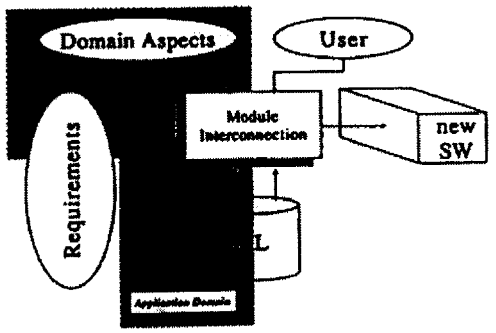


Figure 4-3. Forward engineering model (Gall and Klosch 1992).

Figure 4-2 models a domain to extract reusable assets from existing assets, known as reverse engineering. This same model may also be applied to the creation of assets process. Figure 4-3 models forward engineering, a process to integrate existing software assets to form a functional solution. Together, these separate processes create the two-phase reuse engineering lifecycle model shown in Figure 4-4.

The reuse lifecycle is composed of the following eight steps: Abstraction, Analysis, Selection, Generalization Attribution, Retrieval, Specialization, and Interconnection.

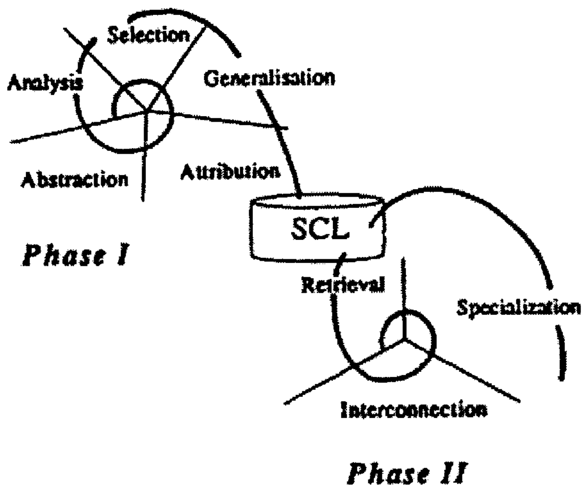


Figure 4-4. Two-phase reuse engineering lifecycle (Gall and Klosch 1992).

Phase I of the lifecycle, which includes Abstraction, Analysis, Selection, Generalization, and Attribution, which creates. The abstraction phase uses reverse engineering to salvage reusable components of existing assets. This step is mainly to recycle components, particularly code and design. High-level abstractions of components are represented in graphical form in this step. Next, in the analysis phase, the results of abstraction are examined for modules that are used repeatedly. In selection, some of these modules are selected for reuse. A generic model of these components is created in generalization, which becomes a template for later reuse. Models define interfaces and data types at the highest possible level. This template can later be parameterized to create customize components depending on the application. In attribution, generic modules are given identifiable features based on functionality and data type. Prieto-Diaz in 1989 introduced faceted classification for formal reuse. Faceted classification differentiates modules based on predefined characteristics and stores these modules in a software components library (SCL). The SCL is underlying tool for Phase II of the reuse lifecycle.

Phase II of the lifecycle reuses existing components, which encompasses Retrieval, Specialization, and Interconnection. In retrieval, users access the SCL for components to be used in development. These components are parameterized during specialization to address a particular specialization. Finally, these components are integrated with other modules to form a functional application.

The SCL mainly supports code reuse. However, the reuse lifecycle can be implemented for design and document reuse as well. For example, a architecture component reuse may take the following steps:

1. Define generic problems after examining several software systems (Abstraction).
2. Characterize the problem for this domain or any part of it (Analysis, Selection).
3. Create a general framework specification adaptable to several different projects using generated mechanisms (Generalization, Specification, Attribution)
4. Parameterize general framework specification to create concrete components. (Retrieval, Specialization, Interconnection)

Reuse engineering is the result of contributions by people at all phases of software development.

4.1.1 Creating Reusable Assets

Reusable Assets are mainly created through reverse engineering or are designed for reusability. Reverse engineering, as shown in Figure 4-2, is the postprocessing of software products to determine the fundamental design concept. Other assets are created specifically for reusability. Specific methods to make assets reusable will be discussed in chapter 6.

A method to characterize and store these assets for easy retrieval must also be included in the reuse process if these assets are to be reused. Storage will be discussed in chapter 6.

4.1.2 Reusing Existing Assets

Forward engineering, shown in Figure 3-3, is either ad-hoc or systematic. Ad-hoc reuse is a salvaging technique where past assets are applied as an afterthought or only when necessary. Systematic reuse is a formal integration of reuse habits in the development cycle. As discussed in chapter three, systematic reuse has the greatest potential to increase productivity while maintaining quality.

Developers in the past mainly practiced ad-hoc reuse. Ad-hoc reuse is an informal method of asset reuse based completely on the preference of the individual. This type of reuse mainly focuses on code reuse through a private component library. Developers are self-contained entities and reuse is at each's discretion. Ad-hoc reuse achieves less than 50% reuse (Sarshar 1996) and less than 25% increase in productivity (Biggerstaff and Perelis 1989).

Systematic reuse is a formal method of integrating reuse in the software development lifecycle. This approach to reuse is a shift in paradigm from individual craftsmanship of software to mass production (Sarshar 1996). Reuse activities must be included all phases of the development life-cycle in order to substantially increase total development productivity (Biggerstaff and Perlis 1989).

4.2 Reusability Principles

Reusability is a software quality factor that adds value to a process or product. IEEE defines the following attributes as essential to a strong reuse program. The achievement or degree of achievement of each criterion measures reusability of an asset. Each of the following attributes are necessary in making reuse theory concrete.

4.2.1 Simplicity

IEEE defines simplicity as "those attributes of the software that provide implementation of functions in the most understandable manner". Simplicity practices avoid activities that increase the complexity of an asset. In reuse engineering, an uncomplicated module is desirable because developers can easily and quickly understand these modules. In a simple design structure, modules are independent and should flow in a top-down format. Modules should have the following characteristics:

- Independence
- Detailed descriptions including input, output, processing, and limitations
- Single point of entry and single point of exit
- Compartmented database
- Unique functions. Duplicate functions do not exist.

In a simple code structure, a programming standard is established at the beginning of the coding phase to enforce duplicate standards across applications. This standard becomes a knowledge asset to support simple and easily understandable code. Simple code structures have the following characteristics:

- Module is not self-modifying. Loop indexes should not be modified and loops should end logically.
- Minimal number of nesting levels.
- Input variables to a statement should be minimized to a reasonable level. Single use variables and unique naming structures are encouraged.
- Extraneous code is removed.

4.2.2 Independence

Machine independence is "those attributes of the software that determine its dependency on the hardware system" (IEEE 1999). Independence attempts to decrease the amount of external factors a component relies on. An independent asset can be easily exported from one system to another. Factors are grouped into software system independence, machine independence, and degree of independence. Software system independence includes the following characteristics:

- Common standard language
- Dependence on library routines minimized
- No operating system references
- Minimize ratio of three systems references/total Line of Code

Machine independence attempts to free assets from physical hardware restrictions. Products that are machine independent is:

- Commonly available on workstations
- Free from references to input/output
- Independent from character and code size
- Understood by several workstations

4.2.3 Document Accessibility

Documentation is the knowledge legacy of software development cycles and reusable assets are the infrastructure that enables reuse engineering. Document accessibility is necessary for users to understand the basic functionality and purpose of a reusable asset. Criteria include:

- Clear document structure and easily understood descriptions of procedures, functions, and algorithms

- Adequate indexing of assets to be allow easy access to information
- Clear figures and graphs depicting data flow and control
- Identify performance parameters and limitations
- Comprehensive description of interfaces and architecture

4.2.4 Application Independence

Application independence is "attributes of the software that determine its dependency on the software environment" (IEEE 1999). The environment includes operating systems, utilities, and input/output routines. Independent applications should:

- Limit specific references to database schema
- Create global data standard for adding comments detailing data manipulation, data's origin, and data's use.
- Create parameter input/output standard for adding comments detailing data's composition and use.
- Create requirement that specific references to computer architecture must be localized.
- Limit or avoid the use of microcode instruction statements.
- Develop functional processing algorithms non-unique to the system's application.

4.2.5 Generality

The goal of reverse engineering is to create general assets. Generality is "those attributes of the software that provide breadth to the functions performed" (IEEE 1999). General assets should:

- Reference other modules as least as possible, and modular references should be relatively low compared to the total number of modules.

- Separate input, processing and output functions within an application.
- Define constants only once.
- Limit processing by data value or volume.
- Separate application and machine dependent functions.

4.2.6 Modularity

IEEE defines modularity as "those attributes of the software that provide a structure of highly independent modules" (1999). Modular systems are more easily separated, abstracted, and reused. Modular systems are:

- Structured hierarchically with a top down design.
- Representative of only one function.
- Content, common, and external coupling minimized in requirements.

4.2.7 System Clarity

System clarity is an attribute to provide for definite modular boundaries and clarifies the input/output functions. Criteria includes:

- Create requirements to separate input/output functions from computational functions
- Isolate input/output functions from computational functions
- Combine similar characteristics of functions to form unique top level functions

4.2.8 Self-Descriptiveness

Self-descriptive modules are those modules with describes implementing the function. Modules are defined by the following characteristics.

- Define a standard format for organizations of modules
- Define a standard format for comments to be included in assets
- Add substantive comments
- Variable names are descriptive
- Allow only one statement per line

4.3 Reuse Activities

Reuse activities include all form of asset and process reapplication and modifications. Below are reuse activities common to all organizations.

- **Design and Code Scavenging (DCS).** An ad-hoc model for reuse, design and source code scavenging represent the lowest level of reuse. Design and code fragments of existing systems are scoured for applicability in another domain. The objective is to reduce development time in the software development cycle. The productivity of this activity is dependent on the experience of the scavenger and the information detail in each asset.
- **Source Code Components (SCC).** Source code components are a more organized scavenging techniques because products may be packaged in a code library that is easier to reuse. As a result, SCC is more effective because components are designed and developed for reuse. SCC is most effective in a strong reuse engineering program.
- **Program Schemes (PS).** Program schemes are designed specifically to be reused and contain high level structure separating the abstract part from the fixed part. In addition, instantiation and usage details are provided within the code for developers to quickly comprehend. Program schemes are organized in a PS library system.
- **Very High-Level Languages (VHLL).** VHLL construct abstract models of assets to facilitate software development. VHLL contains inherent knowledge within its

template for the compiler to extract specifications to realizations. This type of reuse activity is a concise and formal notation for abstraction.

- **Transformation Systems.** Transformation systems are mapping techniques between two distinct objects. Systems may be used to integrate modules in a project or development between phases. The first phase requires very high level abstraction providing specifications of a system. The second phase of TS is customization of an asset to a solution. TS has thus far been an interactive model requiring human guidance.

4.4 Summary

This chapter discussed reuse in the software development cycle as well as quality factors required of a strong reuse program. Additionally, general reuse activities and implementation were presented. The next chapter will examine a specific case of reuse, reuse in the collaborative software development environment.

Chapter 5

Case Study: ieCollab

In a nine-month period beginning in September 1999, the ieCollab project team examined the problem of distributed collaboration, implemented a structured approach to solving distribution as a barrier to collaboration, and built on a solution introduced in previous years to enhance collaboration. The team itself was geographically distributed, and experienced similar frustrations working dispersely that ieCollab is meant to solve. At the end of the project cycle, the team failed to deliver a functioning product which fulfilled the earlier objectives. Because of the geographical and temporal barriers constraining this project, ieCollab presents a strong case for formalizing reuse activities. Although reuse engineering was not a formal phase in this project, team members practiced ad-hoc reuse of old assets and current assets. This case study evaluates the development of ieCollab, the effectiveness of reuse in supporting the software development process, and reuse opportunities to increase the effectiveness of distributed collaboration.

First, this chapter will provide background on ieCollab, including objectives and team organization. Next, this chapter will discuss ieCollab in the context of distributed and temporally separate collaboration. Opportunities for reuse will then be explored. Finally, this chapter will conclude with reuse in support of distributed, multi-year collaboration.

5.1 Initial Project Development

At the beginning of this lifecycle, the ieCollab team developed a structured approach to solving the problem of collaboration. The project team examined the collaborative software development process as well as the problem of distributed collaboration.

5.1.1 Purpose of Project

Webster's dictionary defines collaboration as "working, one with another, cooperate" and to "cooperate, usually willing, with an enemy nation." Collaboration, sometimes valuable and sometimes unfavorable, is necessary to function effectively in the globalizing environment. Organizations are interacting with greater frequency at lower levels of the organization and organizations themselves are becoming increasingly dispersed. Manasseh, in his 1999 thesis, defines collaboration as a specific form of cooperation. "Instead of using the word cooperation, it is more suitable to use collaboration in this context. Collaboration, unlike cooperation, signifies interaction between the various members of the company and not only interaction between the companies themselves" (Manasseh 1999). To function efficiently and cost-effectively, a reliable method of communication and cooperation is necessary to support the new team structure.

The purpose of this project was to develop technology to enable distributed collaboration using distributed collaboration. Both the technology developed and methodology will be evaluated for effectiveness and efficiency. Recommendations based on the problems encountered during this experiment will be implemented in the next development cycle. Thus, this project imitates a real-world software development cycle by implementing a feedback mechanism.

5.1.2 Class Objectives

Introduced in Chapter 2, ieCollab, and its predecessor CAIRO, is a communication tool utilizing the Internet to enable efficient collaboration between distributed teams. This project focused on distributed software development teams developing and implementing large-scale software. The objectives were to not only build ieCollab, but to study distributed team interaction in a structured environment. This experiment concluded with problems and recommended solutions, both for the process and product, to be implemented and tested by next year's development team.

5.1.3 Requirements

The team was constrained from the start by technical and organizational requirements to limit the variability in this experiment. Constraints on team organization were incorporated to ensure geographically dispersed interaction between virtual teams. Organizational limitations include:

- Working groups, a subset of the team, must be composed of people from CICESE, PUC, and MIT when possible.
- Interaction must occur through a technology medium. Acceptable means of communication are electronic mail, CAIRO, NetMeeting, and ICQ. Face-to-face interaction between distributed parties is prohibited.

In addition, constraints on technology were added to increase functionality of the product. Technical limitations are:

- Build on previous years' application to accurately model software development organization because software applications rarely begin from a clean slate.
- Develop in a multi-platform environment. ieCollab needs to execute on all operating systems in order to be useful to parties in dissimilar environments.
- Final product must be easily accessible to all remote parties.

5.1.4 Team Structure

The overall project is structured according to the incremental model for software development. Based on recommendations of the last two years, this year's development team followed a formal structure in creating its contribution to the project. Members were responsible for specific needs in the cycle, and interaction and communication within the team was important to ensure a quality application was created.

5.1.5 ieCollab's Lifecycle Model

The ieCollab team reused last year's process model with slight modifications. A business/marketing phase to emphasize the customer presence in developing software solutions. The ieCollab team was broken into working groups based on the different phases of the software development cycle, shown in Figure 5-1. Development phases in ieCollab's development lifecycle are: Business/Marketing, Requirements Analysis, Design, Programming, and Testing. Supporting development are the Project Management, Quality Assurance, Configuration Management, and Knowledgement Management phases. The programming and testing of release 1 and release 2, corresponding to project version 1 and version 2, were accomplished. Participants chose specific primary and secondary roles based on interest as well as experience.

Business/Marketing

The first phase in the lifecycle is the Business/Marketing phase. The marketing phase identifies the market needs and technology trends, targets customer niches, and creates a competitive strategy for the product. After a market niche is identified, the business phase defines the product goals based on the market analysis report of the available opportunities. The product goals identify features of the product based on customer needs. Activities in this phase may include customer surveying, competition identification, trend analysis, and strategy construction. The goal of this phase is to visualize a product that meets customer needs.

Requirements Analysis

Based on the identified customer needs, the Requirements Analyst team formulates software requirements and specific functionality of the application. The software requirements determine the software constraints, the usage level provided, and customer interaction with the application. Activities in this phase include creating use cases and write requirements specification. The goal of this phase is to understand how the user will react to and employ the determined solution to their identified needs.

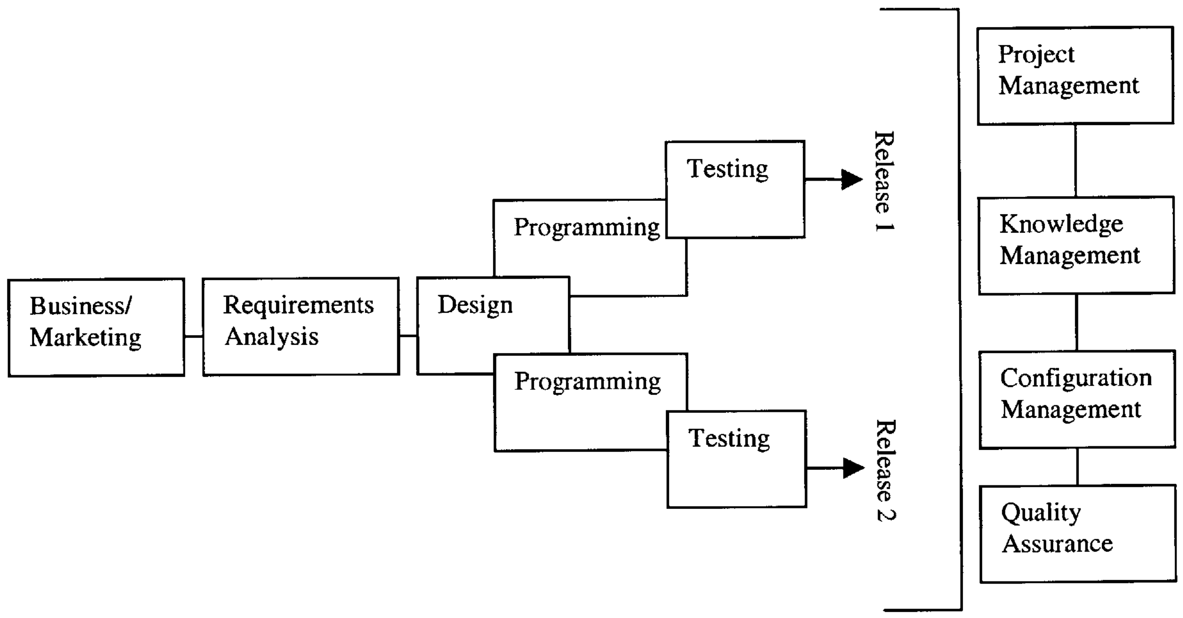


Figure 5-1. Software lifecycle phases in the ieCollab

Design

The Design team creates a methodology for the implementation of requirements specified by the Requirements Analyst team. Designers create modeling diagrams to translate user requirements to the Programming team. In addition, this team designs the system architecture and provides technical details such as the coding and connectivity protocols used. The main activity in this phase is creating modeling diagrams, both integrated and compartmentalized. The goal of this phase is to clarify the functions within an application and to guarantee customer requirements are incorporated.

Programming

The Programming team implements the user requirements to create the functional application according to the specified design and implementation in the previous step. The functional application is then interfaced to any external systems or integrated with existing components. The main activity in this phase is to write the code, compile, debug, and comment the code behind the application. The goal of this phase is to instantiate user-defined requirements.

Testing

The compiled code and running application is transferred to the testers for requirements and design verification and application functionality. The Testing team develops test cases applied to the program in order to assure correct and consistent analysis and design. The testing phase attempts to use the completed application as any user would and to identify errors in application. Activities in this phase are running the application as a user with little background knowledge and inputting incorrect values into arguments. The goal of this phase is to determine all exceptions in the application.

Project Management

The Project Management Team organizes, plans, monitors and controls the project to ensure on-time delivery of a quality product. The project manager supports each role by facilitating interaction between each team. The activities in this phase include monitoring production in weekly reports and resolving resource, schedule, or human conflict. The goal of this phase is to develop a quality product while minimizing risk and unforeseeable schedule variability.

Quality Assurance

Maintaining and achieving a standard level of quality in both product development and process lifecycle is the responsibility of the Quality Assurance Team. The quality assurance team monitors the quality level of the product and the end-product in addition to reporting any discrepancies in the process. Activities in this phase include walkthroughs, verification, and validation during and after the completion of each phase and submission of work products to monitor requirement fulfillment in addition to monitoring quality practices in software development process. The goal of this phase is to ensure that quality is achieved and documented.

Configuration Management

The Configuration Management Team support programmers in organizing, controlling, and documenting their work for control and change status. This team manages the different versions of code released through Concurrent Versions System (CVS) and finalizes the work and functional products. The main activity in Configuration Management is to handle and append versions to work products or functional products. The goal of configuration management is to track releases of products and record changes or fixes to the system.

Knowledge Management

The final support team instantiated in this cycle is the Knowledge Management Team. The Knowledge Management team is responsible for creating and enforcing a standard format for project documents and maintaining a library of all old and current documents. Knowledge management activities include creating and maintaining a web repository to store all work and functional products. The goal of knowledge management is facilitate necessary shared knowledge between collaborators and control in information overflow.

5.1.6 Organization

Teams not only need to coordinate with collocated members, but interact as well with dispersed members. In accordance with organizational constraints, each team was composed of members from all three locations when possible. Each team consisted of a team leader and primary members. Identified in the Figure 5-2 below are team members, originating school, and primary roles.

The project managers of ieCollab were Limanksy, Abbott, and Arantes at MIT and Garcia in CICESE. Project management responsibilities were distributed in order to have a strong presence in many locations. The business/marketing team consisted of Mills, Vemulapalli, Wariyapola, and Kyauk from MIT and Solari from PUC. The requirements analysis team was composed of Maria and Polo from PUC, Rosa from CICESE, and Krishnan, Lehman, and Ng from MIT. Chen and El-Solh from MIT, Machorro from PUC, and Rafael and Moran from CICESE composed the design team.

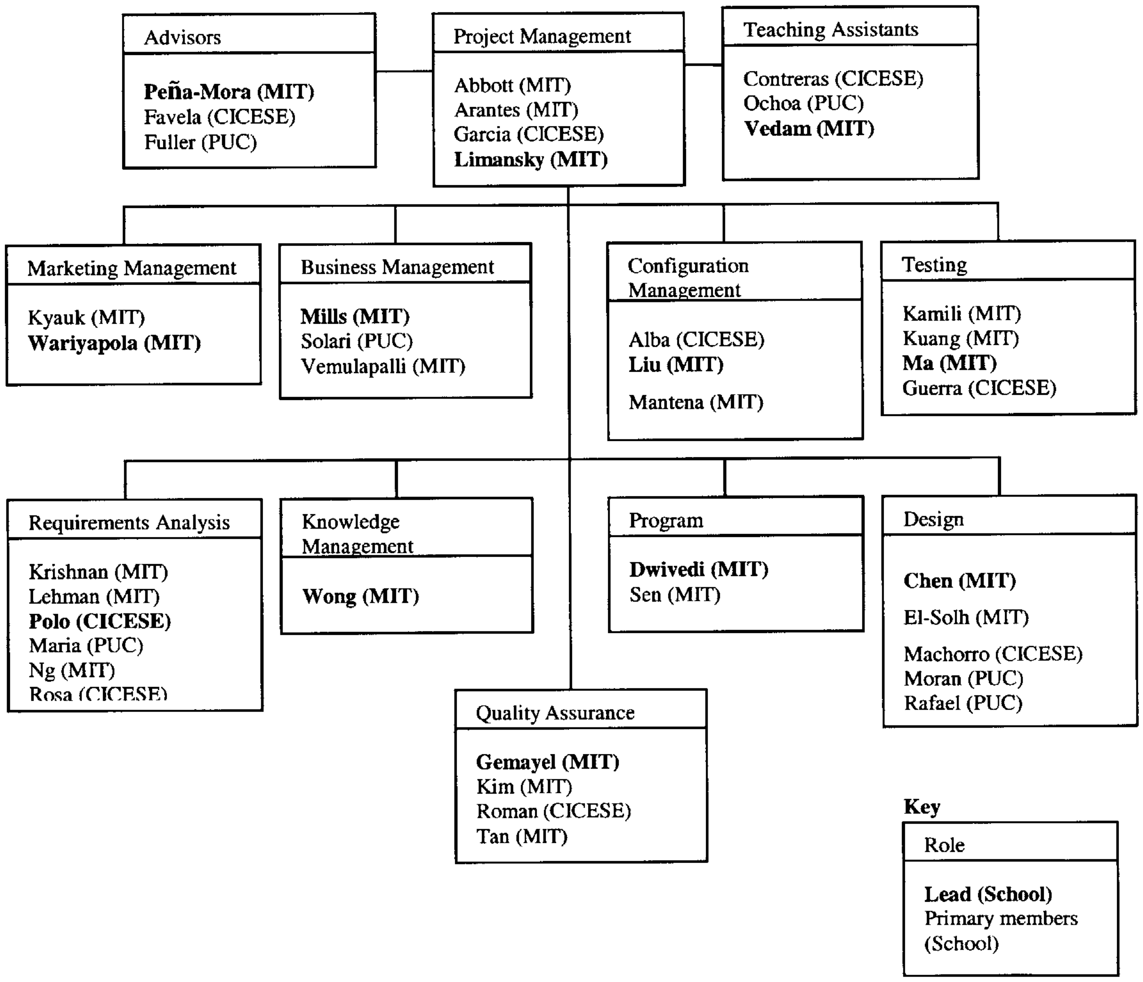


Figure 5-2. ieCollab Organization chart

The programmers were Sen and Dwivedi from MIT. The testing team was made of Kamili, Kuang, and Ma from MIT and Guerra from CICESE. The quality assurance team members were Roman from CICESE and Gemayel, Kim, and Tan from MIT. The configuration team was composed to Liu and Mantena from MIT and Alba from CICESE. Finally, Wong was the sole member of the knowledge management team. In addition, the team was supported by Professors Peña-Mora, Favela and Fuller and Teaching Assistants Vedam, Contreras, and Ochoa from MIT, CICESE, and PUC, respectively.

In addition, each member of ieCollab also assumed a secondary role. Most members assumed a secondary role as programmer, as the most resources were required in that area.

5.1.7 Project Schedule

The project was limited to nine months, the length of the Master of Engineering program at MIT. Lectures on the software development process and collaborative lab assignments to familiarize the team with the development process as well as distributed collaboration occupied the first term. The project managers also scheduled the beginning of the project to coincide with the end of the lectures. Thus, the business/marketing phase was completed during the first term and the requirements analysis phase was begun at the end of the first term.

The remainder of the project was completed in the second term and MIT's Independent Activities Period. The project ended with delivery of the most completed application package in mid-April. The entire project was developed in eight months.

5.1.8 Product Development

This year's project team decided to follow the incremental model of development. In an incremental model of development, different parts of the application are developed in a waterfall lifecycle nearly simultaneously with feedback between parts of applications. ieCollab's project lifecycle is shown in Figure 5-3 below.

The ieCollab team chose to follow the incremental model of development because of the time constraint. One of the benefits of incremental is rapid development time because project goals are separated from each other and completed in increments. In addition, the incremental model incorporates quality improvements and customer feedback in the next increment of the project.

This year's contribution to ieCollab and CAIRO used two increments. The first increment developed version 1, meeting management, of ieCollab. The second increment was intended to develop version 2, transaction management. In practice, phases of each increment were combined due to time constraints, as shown in Figure 5-1. The business/marketing, requirements analysis, and design phases were combined into one single increment. The coding and testing phases for each increment were kept separate because resources to develop each were limited. Version 1 was completed prior to version 2 of the product.

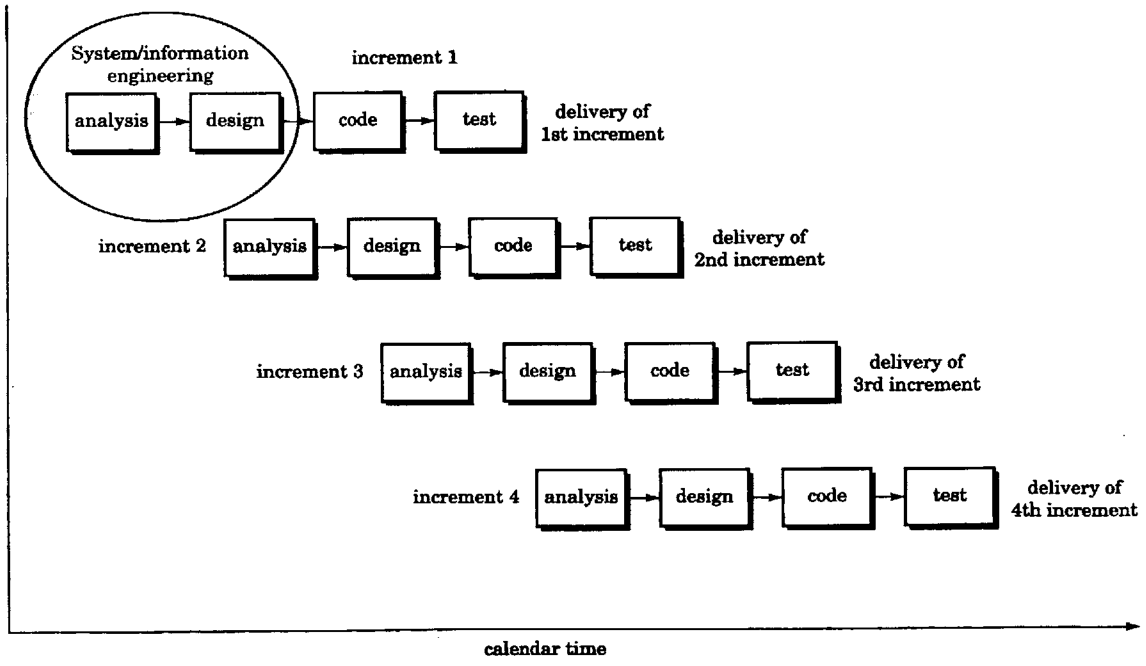


Figure 5-3. Incremental model of development (Pressman 1997).

5.1.9 Knowledge Assets

Knowledge assets are those assets that facilitate knowledge flow between collaborating parties. In ieCollab, these assets took the form of work products or functional products. Assets from previous years were available to team members and each team in the current year produced their own assets (shown Figure 5-4). Work products are inter-team assets and were valuable in communicating a previous phase's ideas to the next phase. Work products often took the form of diagrams, models, or text and enabled the next phase to understand the previous phase's ideas. Functional products were used mainly by the current team to understand their roles and were intra-functional team assets. Functional assets took the form of team plans and status tracking.

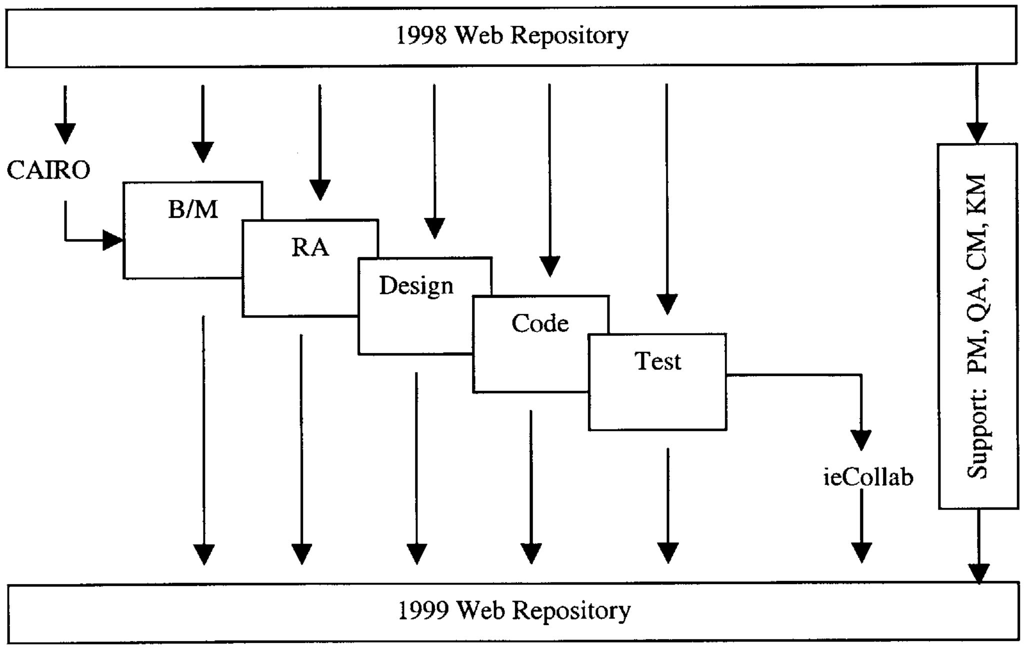


Figure 5-4. Asset reuse and creation in ieCollab.

This year's team began with the CAIRO application, in a compiled form, and a limited number of functional and work products. CAIRO functional products inherited include: Project Management Plan, Quality Assurance Plan, Testing Plan, and individual theses. Business plan, design specifications, test cases, CAIRO, quality assurance and control records are CAIRO work products passed onto ieCollab. Knowledge assets were stored in two separate web repositories and only one, holding functional plans, was accessible to all team members.

CAIRO functional assets were archived to the project web repository, discussed in section 5.1.9.2.1. CAIRO work assets were stored at http://boxster.mit.edu/1.120/home.nsf/HomeLeft/test_left. This website is a link basically to last year's class web repository.

The ieCollab project team also created assets to be shared among the current group and as experience legacy for the next year's development team. Section 5.1.9.2 discusses the repository used in this project.

5.1.9.2 Web Repositories

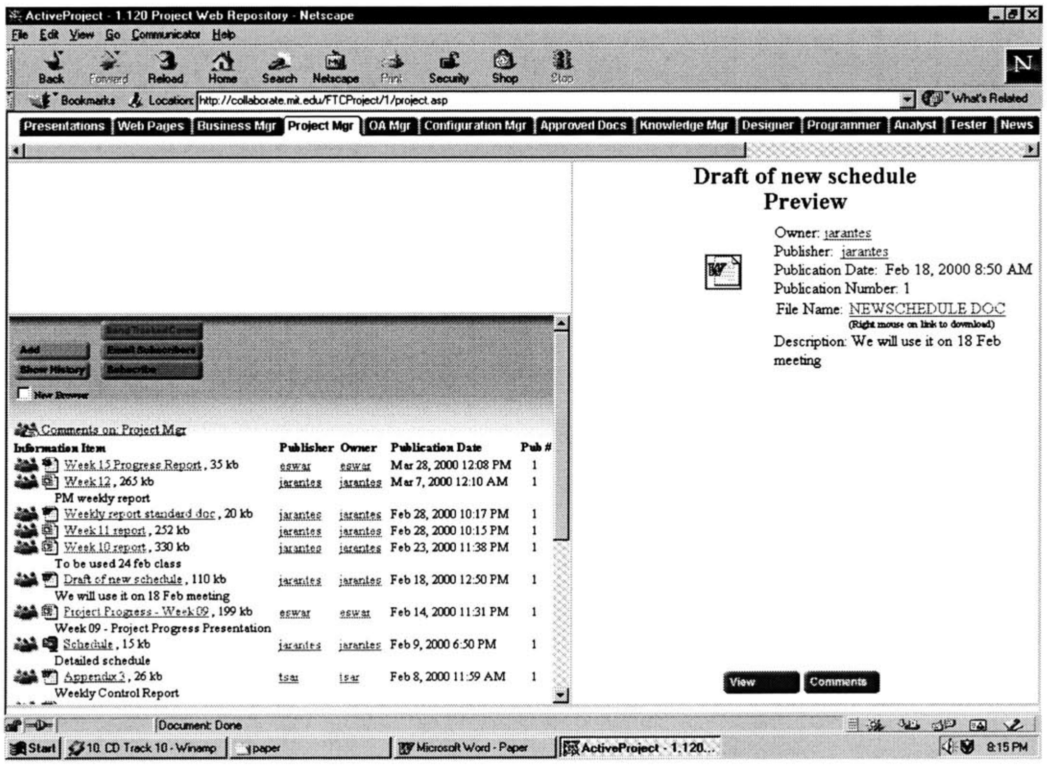
The ieCollab team began with an extensive collection of knowledge resources. Past theses, project proposals, group reports, and meeting minutes were all archived as information resource for the proceeding years' teams. This year's team also added its own assets to the collection. The collection is ultimately the project's memory.

5.1.9.2.1 Project Web Repository

The project web repository housed current and archived assets. Archived assets were mainly past theses from 1997 and 1998 CAIRO teams, not team plans or work products. This web repository enabled inter-team and intra-team collaboration as a central database of information to leverage knowledge to all members as well as supported multi-year collaboration by providing archived assets.

Knowledge assets are housed in the project web repository, located at http://collaborate.mit.edu/1.120.html. The web repository resides on one of MIT's Civil and Environmental Engineering department computers under the direct supervision of Professor Peña-Mora.

The web repository layout is shown in Figure 5-4. The layout is three viewing panes, each controlling different options. The top pane provided the option to view a specific functional teams' assets. The left pane listed the assets by order of upload with options to select an action to a particular document. The right pane offered detail options on the asset.


ActiveProject - 1.120 Project Web Repository - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://collaborate.mit.edu/FTCProject/1/project.asp
What's Related
Presentations Web Pages Business Mgr Project Mgr OA Mgr Configuration Mgr Approved Docs Knowledge Mgr Designer Programmer Analyst Tester News
Add
Show History
Email Subscribers
Subscribe
New Browser
View
Comments
Document: Done
Start
10. CD Track 10 - Winamp
paper
Microsoft Word - Paper
ActiveProject - 1.120...
8:15 PM

Comments on: Project Mgr

Information Item	Publisher	Owner	Publication Date	Pub #
Week 15 Progress Report, 35 kb	eswar	eswar	Mar 28, 2000 12:08 PM	1
Week 12, 265 kb PM weekly report	jarantes	jarantes	Mar 7, 2000 12:10 AM	1
Weekly report standard doc, 20 kb	jarantes	jarantes	Feb 28, 2000 10:17 PM	1
Week 11 report, 252 kb	jarantes	jarantes	Feb 28, 2000 10:15 PM	1
Week 10 report, 330 kb To be used 24 feb class	jarantes	jarantes	Feb 23, 2000 11:38 PM	1
Draft of new schedule, 110 kb We will use it on 18 Feb meeting	jarantes	jarantes	Feb 18, 2000 12:50 PM	1
Project Progress - Week 09, 199 kb Week 09 - Project Progress Presentation	eswar	eswar	Feb 14, 2000 11:31 PM	1
Schedule, 15 kb Detailed schedule	jarantes	jarantes	Feb 9, 2000 6:50 PM	1
Appendix 3, 26 kb Weekly Control Report	tsar	tsar	Feb 8, 2000 11:59 AM	1

Draft of new schedule
Preview

Owner: jarantes
Publisher: jarantes
Publication Date: Feb 18, 2000 8:50 AM
Publication Number: 1
File Name: NEWSCHEDULE.DOC
(Right mouse on link to download)
Description: We will use it on 18 Feb meeting

Figure 5-4. Project web repository.

Storage

Storage privilege was not restricted. Any member of the current team was able to store assets. Users logged into the website with a username/password combination. Documents were arranged chronologically by upload date. Figure 5-5 shows the organizational structure of the library.

Documents are listed with the following information:

- Publisher. The party who uploaded the document.
- Owner. The person responsible for creating the document.
- Publication Date. The date that the document was uploaded.
- Document Control Number. The document version number.
- Size. The size of the document.
- Description. A short description of the document appears below the asset entry.

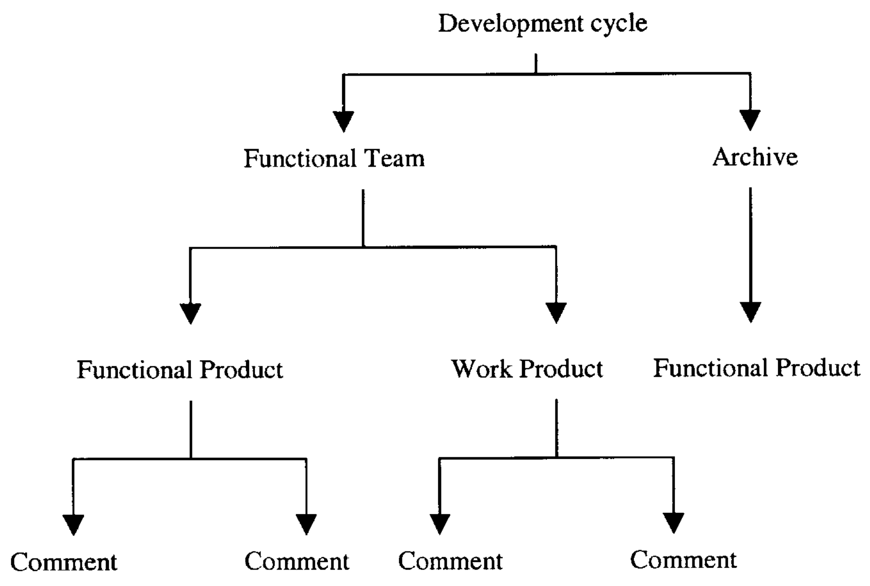


Figure 5-5. Project web repository library structure.

In addition, parties may add comments about to each document through the comment option. However, comments are separate documents as well and require uploading to the site. Finally, emails may be sent to all subscribers with the option to notify when a new document has been uploaded.

Retrieval

All members of the current development team had access to the project web repository. Members must have prior knowledge of the web structure to efficiently search for assets. Assets are retrieved by development cycle, then functional team, and finally document. More than one version of a document may exist, but versions are not differentiated. No search function exists for documents.

Documents were either downloaded or previewed. Both options forced required opening a third-party text editor, such as Microsoft Word. Documents were not viewed within the web browser. The only difference is that documents previewed were not first saved to the user's account. In effect, the object was destroyed after exiting the preview application. Comments were also required to be downloaded before viewing. Archived documents, however, if in Acrobat .pdf format, could be viewed within the browser.

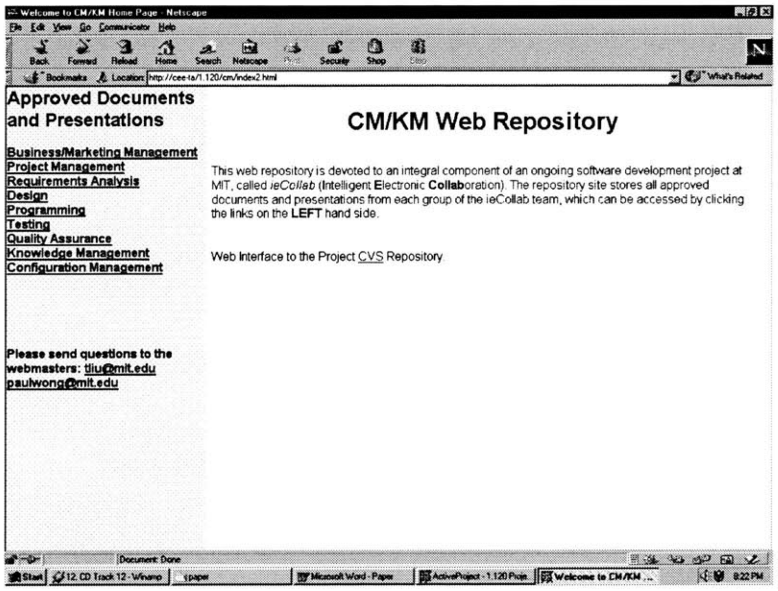
5.1.9.2.2 CM-KM Web Repository

All drafts were held at the project web site, but finalized documents were held in the CM-KM Final Drafts web repository shown in Figure 5-6 and located at http://cee-ta.mit.edu/cm/index2.html. This web repository mainly benefited the current team by differentiating drafts and past assets with the finalized asset. Shown in Figure 5-7 is the organization of the repository.

The layout of the web repository is show in Figure 5-6. This repository is a one-pane screen with functional teams listed on the left hand side. After selecting a functional team, the user is shown a list of finalized assets. The user can then select the desired asset for download. However, different from the project web site, this web site does not provide any information other than the document title.

Storage

Only members of the configuration or knowledge management team can upload documents. Assets listed do not include description nor any information about the document. Users must have a previous knowledge about the system for efficient navigation and use.


Welcome to CM/KM Home Page - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://cee-ta/1.120/cm/index2.html
What's Related
Document: Done

Approved Documents and Presentations

Business/Marketing Management
Project Management
Requirements Analysis
Design
Programming
Testing
Quality Assurance
Knowledge Management
Configuration Management

Please send questions to the webmasters: tliu@mit.edu paulwong@mit.edu

CM/KM Web Repository

This web repository is devoted to an integral component of an ongoing software development project at MIT, called *ieCollab* (Intelligent **E**lectronic **Collab**oration). The repository site stores all approved documents and presentations from each group of the ieCollab team, which can be accessed by clicking the links on the **LEFT** hand side.

Web Interface to the Project CVS Repository

Figure 5-5. CM-KM Final Drafts web repository.

Retrieval

Users must have prior knowledge of the system and the desired asset in order to conduct efficient search. Users retrieve by functional team then by document. Only one version of each product exists. However, no search function is available for this repository.

5.1.10 Available Resources

Several resources were available to the ieCollab team in developing this project. The project team used the both technology and external human resources in development and support of the product. The most widely used resources are described below.

5.1.10.1 Development

Development resources are those resources used by the project team to directly create the product. These resources are UML, JAVA, CORBA, and Windows NT/98 machines.

UML

Unified Modeling Language is a method for specifying, visualizing, and documenting artifacts of a proposed object-oriented system. UML was mainly used by the business, requirements analysis, and design team for specifying, in increasing degrees of specificity, the requirements and implementation of the proposed solution.

JAVA

The project team used JDK 1.2.2 package to create the product. In addition to creating new classes, packages and prewritten classes were accessed from the Java website. Located at http://java.sun.com, this web site provided packaged source code and detailed information on class input, output, and use. The code was not in compiled form and is specific to version and implementation. Because the project team decided to program in Java due to the short development time, the project team heavily utilized this web site as a source of pre-written code. **J**ava **D**ata**b**ase **C**onnectivity (JDBC) classes were also used by the development team and obtained from this website.

CORBA

The project team made use of the widely accepted **C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA) in the application. CORBA is a protocol that allows the client to access objects written in different languages. CORBA separates the interface and implementation layers, while providing a low-level networking code stubs for different interfaces.

Windows NT/98

Windows NT/98 workstations were the main technical hardware used to develop ieCollab and its work products and functional products. Machines were widely available in all three distributed locations.

CAIRO

The project team began with CAIRO, the latest version of the product with virtual meeting environment capability. CAIRO is an application that supports virtual meetings between distributed parties. The project incorporates formal and casual chat protocols, log database, and whiteboard capabilities. CAIRO was developed with JAVA using JDK 1.1 for Windows.

5.1.10.2 Support

Support resources are resources that indirectly create the product by supporting development activities. The ieCollab project team made use of web repositories, CVS, and human resources in this development cycle.

Web Repository

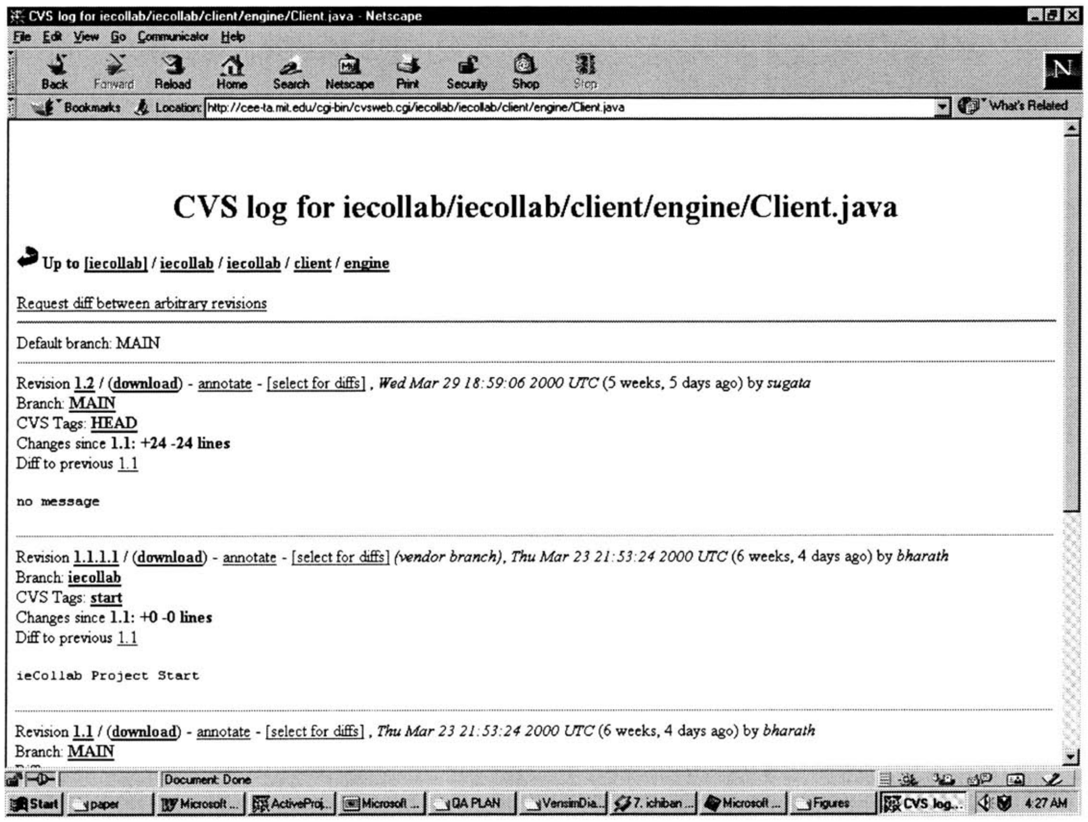
ieCollab assets were shared through a web repository. Please refer to section 5.1.10.2 for further details.

CVS

Concurrent Versioning System (CVS) is used to track the changes made to the code and identify a version number between code packages. The Configuration Management Team supplied a web interface to CVS for access to the code stored in CVS, shown in Figure 5-6. CVS customizes a code deposit by

- Author
- Publish date
- Last change date
- Branch
- CVS tag
- Changes made
- Previous version number

One function not utilized by the ieCollab team is a comment extraction function. This would allow the user to extract all the comments from the code file and display the comments in a file. With this function, the user could determine the purpose and functionality of the code and its potential for reuse. CVS also acted as a code repository where distributed team members could access any submitted code.


CVS log for iecollab/iecollab/client/engine/Client.java - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://cee-ta.mit.edu/cgi-bin/cvsweb.cgi/iecollab/iecollab/client/engine/Client.java
What's Related
Document: Done
Start paper Microsoft ... ActiveProj... Microsoft ... QA PLAN VensimDia... 7. ichiban ... Microsoft ... Figures CVS log... 4:27 AM

CVS log for iecollab/iecollab/client/engine/Client.java

Up to [iecollab] / iecollab / iecollab / client / engine

Request diff between arbitrary revisions

Default branch: MAIN

Revision **1.2** / (**download**) - annotate - [select for diffs] , *Wed Mar 29 18:59:06 2000 UTC* (5 weeks, 5 days ago) by *sugata*
Branch: MAIN
CVS Tags: HEAD
Changes since **1.1**: **+24 -24 lines**
Diff to previous 1.1

```
no message
```

Revision **1.1.1.1** / (**download**) - annotate - [select for diffs] *(vendor branch), Thu Mar 23 21:53:24 2000 UTC* (6 weeks, 4 days ago) by *bharath*
Branch: **iecollab**
CVS Tags: **start**
Changes since **1.1**: **+0 -0 lines**
Diff to previous 1.1

```
ieCollab Project Start
```

Revision **1.1** / (**download**) - annotate - [select for diffs] , *Thu Mar 23 21:53:24 2000 UTC* (6 weeks, 4 days ago) by *bharath*
Branch: MAIN

Figure 5-6. CVS code repository

Human Resources

The team accessed the experiences and ideas of Professor Feniosky Pena-Mora and Padmanabha Vedam, a graduate student in MIT's Information Technology program. Professor Pena-Mora introduced the project three years ago to the 1997 DiSEL team and has been with the project since. Vedam was a member of the 1998 DiSEL team and worked on the previous year's iteration of CAIRO. Both were informal resources for this year's project team.

Another human resource is members of the current team. However, the degree of knowledge and experience varied greatly from member to member because many came from civil and environmental engineering backgrounds. Some of the students, particularly members from CICESE and PUC, had computer science backgrounds and were thus equipped with the technical skills to create the product. This resource was used informally as well.

5.2 ieCollab

ieCollab is the project team's solution to enable collaboration between distributed parties. Based on the Application Service Provider (ASP) model, ieCollab is a web-based application that provides meeting management tools and document sharing as shown in Figure 5-7.

ieCollab is a both a service and an application. ieCollab is an application service provider that enables distributed teams to collaborate in a virtual meeting environment through the Internet. The ASP model is composed of the client, the server, and the database, shown in Figure 5-8. All applications necessary to enable virtual meetings resides on the server, thus the client is a thin client that only needs to have Internet access. The database stores transactions, documents, and meeting logs. ieCollab also provides functions that supports virtual meetings. Communication tools such as chat, side-talk, whiteboard, audio, and video streaming allow users to communicate through several different mediums. Real-time synchronous sharing and editing of a document permits multiple party view and editing of the same document simultaneously independent of location. ieCollab users can track meetings and events through the Calendar Service. Institutionalized meeting protocols provide structured meeting environment online. Knowledge management stores meeting information and decisions in a centralized database accessible to multiple parties. Finally ieCollab's virtual facilities are secure and transactions are encrypted.

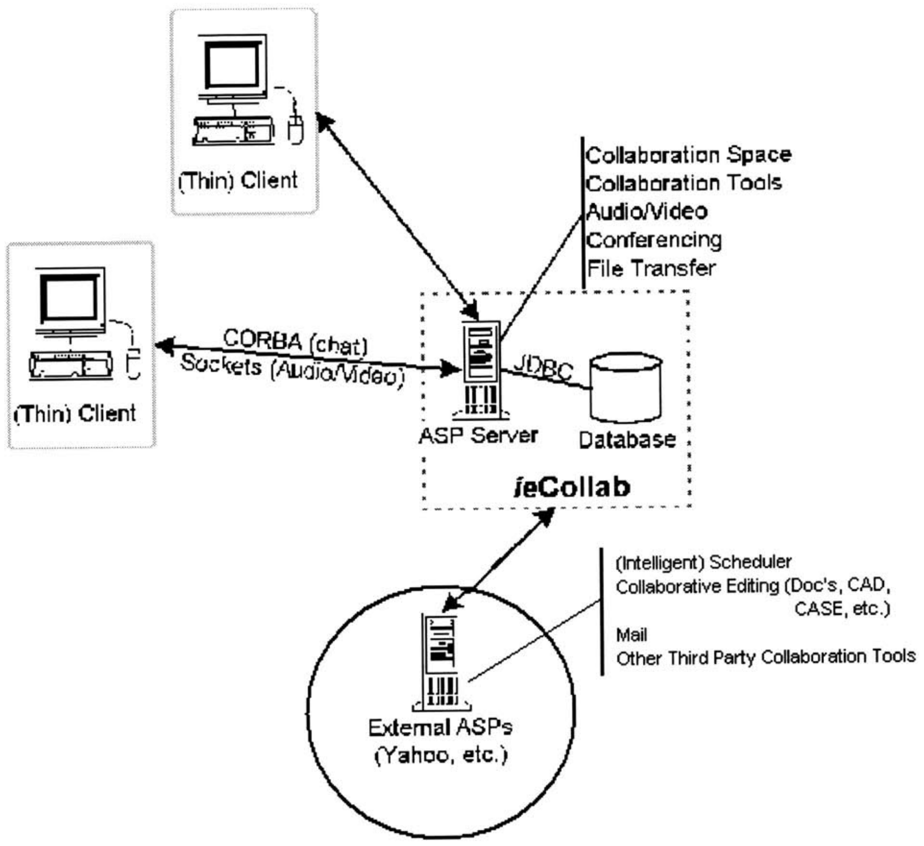


Figure 5-7. ASP model in ieCollab (El-Solh and Tan 1999).

5.2.1 System Design

ieCollab is designed as a three tier system as shown below. The web-enabled CollabClient is a thin client that forces most of the application to reside server-side. The client is connected to the ieCollab server with CORBA protocols. The ieCollab Database is accessed using JDBC classes by the ieCollab server. The knowledge Management service of ieCollab made it necessary to separate the database from the server. Thus, the architecture of ieCollab is three-tier.

The user layer is a Java Applet client that is responsible for basic user interface functions. The server side controls the interaction between the client and the database until the user logs into the CollabClient. A CollabUser object is then instantiated and supports user activity in the virtual space. The database is an Oracle database that contains stored procedures to improve search, storage, and retrieval performance.

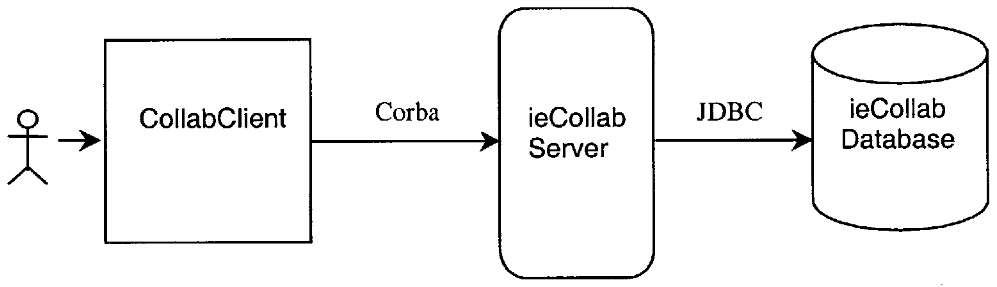


Figure 5-8. ieCollab's three-tier architecture (Hao 1999).

5.2.2 Development Process

ieCollab was divided into four different increments, each increment would add another feature to the overall product. Version 1 is the Meeting Management, which allows users to setup and manage meetings online. Version 2 is the Transaction Management, which tracks meeting management usage and allows users to connect to other ASPs. The Collaboration Server, version 3, provides interactive collaboration tools for communication between dispersed parties. The Application Server, version 4, allows synchronized viewing and editing among collaborators.

ieCollab was scheduled for completion in two development cycles. This development team was to have completed versions 1 and 2. Next year's project team is scheduled to complete versions 3 and 4.

5.2.3 Package Delivery

The development team completed the basic functionality and core classes in Meeting Management (version 1) and Transaction Management (version 2), but was unable to integrate the two versions because of the limited development time. Next year's class will inherit the separate versions. However, foundation work for the business/marketing phase was completed that covers all versions of the product. These assets can be reused in the next cycle of development.

The final package included ieCollab version 3 and version 4 and knowledge assets generated in this development cycle. The final package was submitted in late May.

5.3 Reuse in ieCollab

Reuse appeared in ieCollab in three forms: software development process, the web repository, and knowledge assets. Using these products, reuse facilitated distributed and multi-year teams by collaborating more efficiently.

5.3.1 Process

In software reuse, the goal of process reuse is not merely to implement the same process in all development cycles and project, but to improve and modify the cycle depending on the domain. Development methodology aims to be high-level and abstract in order to encompass several project situations. High-level processes are standardized from experiential data and applied across organizations and projects. Development methods are customized depending on the domain. Improvements are made to the current process based on feedback and suggestions from the previous development cycle.

The changing make-up of the development team requires the development process be modified and improved from the year before. Although a process may have been successful the year before, factors may change that force the development team to adjust for these new constraints. For example, previous years had specific roles, most of which were implemented in this cycle. However, the CAIRO development team also had included a 3D Interface Development Group and User Interaction and Awareness Hardware Group, which was replaced by the Design Team.

The 1998 CAIRO team implemented the spiral model of software development (Manasseh 1999). CAIRO features were developed in three cycles. Each phase of the software development cycle was implemented in each cycle. At the end of each cycle, the entire product was tested and a new design was created for the next cycle. The product evolves after each cycle and grows into a functioning application. Figure 5-9 from Manessah shows the adapted spiral model by the CAIRO team (1999).

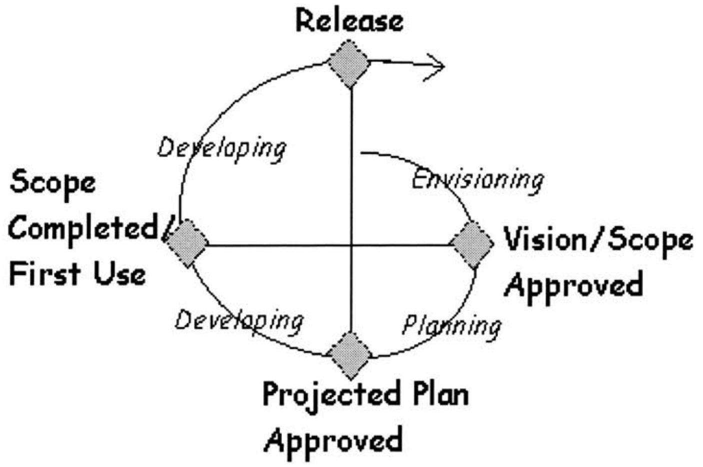


Figure 5-9. CAIRO 98 team's spiral development method (Manessah 1999).

The ambitious objective set at the beginning of the cycle required a development process that allowed simultaneous development of versions 1 and 2. Thus, the ieCollab team chose not to use the spiral method but instead implemented the incremental method (Figure 5-3). Versions 1 and 2 were developed simultaneously, with slight overlap between phases, although the coding and testing were accomplished separately. Consequently, product feedback and process improvements were not available until after both versions were developed.

As stated earlier, the goal of process reuse in a development cycle is to not merely reuse efficient methods, but improve upon previous processes as well. A successful reuse process is a modified development method relative to the previous cycle. ieCollab successfully implemented a reuse process model based on the suggestions of earlier teams. Yang in 1998 suggested that specific roles are assigned to members and a structured process implemented. Manasseh in 1999 suggested definite goals and project requirements is specified at the beginning of each project. The 1999 project team tested these suggestions in a refined process model. Thus, process reuse supports multi-year collaboration.

5.3.2 Product

Product reuse in ieCollab occurred as either work products or functional products. In this development cycle, 1998 CAIRO assets were reused and ieCollab assets were created. This section examines ieCollab's reuse of CAIRO assets and the reusability of ieCollab assets.

5.3.2.1 CAIRO Assets

CAIRO assets are the knowledge legacy of past development teams. Documenting infrastructure as well as product development thinking allows others to benefit from experience gained in one development cycle. CAIRO assets available to the ieCollab project team were the CAIRO software application, requirements analysis, design specifications, test cases, Project Management plan, Business plan, and Quality

Assurance and Control logs. Reusability of these products is measured according to the criteria set in chapter 4.

The goal of document reuse in ieCollab is to provide the experience and knowledge of software development lacking in the team as a whole. Several CAIRO documents were available for ieCollab use. Each phase informally reused last year's documents, referencing ad-hoc the repository.

The level of detail and reusability varied from asset to asset. A few of the assets failed to provide enough detail to be reused by the development team. The change in product paradigm also decreased the applicability of some of the documents.
In addition, documents could not be easily accessed. The only mode to retrieve CAIRO assets was to access the teaching assistant Vedam for these documents. He would then upload the specific document to the current project web repository. Complete access to all documents was prohibited. The web repository housing previous years' documents could only be accessed by users with username/password privilege. Hence, the inaccessibility of reuse documents was another barrier to reuse.

However, the greatest deterrent to this type of asset reuse is the Not Invented Here factor. Teams lacked the motivation and desire to read the documents and apply earlier knowledge. Thus, though a few documents were available, this type of asset was useless to this year's development team.

CAIRO

CAIRO is an application to facilitate distributed meetings in a virtual setting. CAIRO embodies the meeting management features of ieCollab version 1, making CAIRO a strong candidate for reuse. The project team tested CAIRO during the first half of the project but experienced technical difficulty in launching and running CAIRO.

The code package could not be used because the ieCollab team was unable to obtain a high-level design architecture to provide a reasonable explanation of each class's functionality. The code itself was proprietary and given to the current project team in compiled form. Previously developed classes were unusable because the code was not easily accessible and the documentation is lacking. In addition, the previous application was coded in JDK1.1, an older version of the latest package available. Consequently, reusing assets would have a required a layer of code to transform CAIRO into a usable form for ieCollab. As a result, CAIRO code was unusable in its inherited form and code components were not reused. Thus, CAIRO as a pre-existing and compiled application, was not reusable.

Requirements Analysis

The requirements of CAIRO could not be easily accessed by the requirements analysis team. In addition, the requirements analysis team perceived the underlying paradigmatic change would date the previous specifications. Thus, the requirements analysis team did not reuse CAIRO requirements specification.

Design Specifications

The fundamental shift in business model invalidated any CAIRO design specifications. CAIRO is a two-tier system where the database is not contained in a separate layer. ieCollab, as discussed in section 5.2.1, is a three-tier system with its own database. Thus, only a portion of the design could be reused. However, the portion that was available for reuse did not provide enough technical detail to be reapplied by the next project cycle. CAIRO's design team failed to provide sufficient UML diagrams and technical specifications, including detailed class methods and separate cases. Thus, ieCollab's design team was required to create its own design.

Test Cases

ieCollab's testing team developed separate test cases for the next generation of CAIRO because the team focused on the integrated functionality of a three-tier system. In addition, test cases were developed specifically for ieCollab methods. Thus, CAIRO test cases could not be used.

Business Plan

Because ieCollab and CAIRO are fundamentally different products, CAIRO's business plan could not be reused. In addition, since CAIRO's business plan was entered into MIT's 1K Entrepreneurship Contest, copyright rules prevented its use in ieCollab's business plan. However, the structure of the plan and the requirements were reused in the current business plan. The current business team based the level of detail necessary on last year's plan. In addition, the feedback to the plan, organization, or end-product was implemented in this year's team as suggestions.

Project Management Team Plan

The project management team plan was reused extensively as a model for the type and quantity of information required. Because this year's project managers chose a different process model, CAIRO's development plan could not be completely followed. Parts of the plan that could be reused, such as scheduling, were implemented in this year's Project Management plan.

QA Team Plan and Control Documents

CAIRO's quality assurance team provided detailed instructions and improvements to a strong quality assurance program in CAIRO. As a supporting role to development, Quality Assurance engineers are independent of the type of product being created. Thus, the methodology to assure quality could easily be reused in ieCollab. CAIRO quality assurance engineers (QAE) implemented specific activities, such as walkthroughs, validation and verification. This year's team selectively implemented last year's quality assurance process, such as walkthroughs and validation, and also chose to examine the process improvements as well.

5.3.2.2 ieCollab Assets

Each of the work phases created one or both types of work products, as detailed in the table below. MM refers to Meeting Management and TM refers to Transaction Management.

The Business/Marketing team only provided work assets in the form of the 1K Business Plan entry and the Marketing report, which is integrated in the Business Plan. The Requirements Analyst team provided both functional and work products in the Meeting Management Requirements Specification, the Transaction Management Specification, and the Requirements Team plan. The Design team also provided both types of reuse assets in the form of the Meeting Management Design Specification, the Transaction Management Design Specification, and the Design Team plan. The Programming team supplied work and functional products in the Programming Team Plan, the Programming Standards, and ieCollab. The Testing team created the Testing Team Plan, Test Cases, and the Testing report, thus fulfilling all types of assets. The Project Management team not only generated a Project Management Plan describing the project structure, but also provided weekly reports describing the progress and productivity of each team. The Quality Assurance team generated a Quality Assurance Plan and Walkthrough, Verification, and Validation logs. The Configuration Management team supplied the Configuration Management Plan in addition to supporting CVS documents. Finally, the Knowledge Management team provided the Knowledge Management team plan in addition to the User and Technical Manual.

5.3.2.2.1 ieCollab Work Products

ieCollab development work products generated in this cycle are Requirements Specification, Design Specification, Code Package, and Test Cases.

Requirements Specification

ieCollab's requirements analysis document provides detailed user specification for Meeting Management and Transaction Management. The structured layout of the document facilitates finding information quickly in the document. However, the size of each document may deter users from reading and using the document.

In addition, the level of detail provided in the Requirements Analysis document allows next year's development team to integrate these requirements for versions 3 and 4 of ieCollab.

Table 5-1. Assets generated by ieCollab functional teams.

Phase	Work Product	Functional Product
Business/Marketing	▪ Business Plan – 1 K Entry ▪ Marketing Report	
Requirements Analysis	▪ MM Requirements Specifications ▪ TM Requirements Specifications	▪ Requirements Analyst Team Plan
Design	▪ MM Design Specifications ▪ TM Design Specifications	▪ Design Team Plan
Programming	▪ MM Code Package ▪ TM Code Package ▪ Programming Standards ▪ ieCollab	▪ Programming Team Plan
Testing	▪ MM Testing Specifications; ▪ TM Testing Specifications ▪ Testing Report	▪ Testing Team Plan
Project Management	▪ Project Management Proposal ▪ Weekly Reports	▪ Project Management Plan
Quality Assurance	▪ Quality Assurance Verification Report ▪ Design and Requirements ▪ Quality Assurance Programming Standards Checklist ▪ Quality Control Report ▪ Quality Assurance Validation: Client Interface and Requirements Analysis ▪ Comment Template	▪ Quality Assurance Plan
Configuration Management	▪ Change Notice Form ▪ Change Authorization Form ▪ Change Request Form ▪ CVS Tutorial	▪ Configuration Management Plan
Knowledge Management	▪ User Manual ▪ Technical Guide	▪ Knowledge Management Plan

Design Specification

The level of detail in the design specification varied depending on the document. Meeting Management design specifications provided more UML diagrams and a greater level of functionality detail than the Transaction Management document. The design specifications did not provide enough detail for collaboration between different functional teams, and will not provide enough detail to next year's development team. Thus, the design reuse does not facilitate distributed or multi-year collaboration.

However, because the design was modular, the design team was able to separate the product by version and assign different members responsibility to complete each task. After the tasks were completed, each of the members then had to collaborate to integrate each of the different parts of the design together.

Code Package

Java programming language, in which ieCollab is coded, allows convenient reuse for next year's class. Java code is already available in a reusable form. The development library contains downloadable packages and is available on the web in a searchable organization. The packages provide enough detail for users already familiar with Java to efficiently comprehend and reuse classes.

In addition, the modular nature of object-oriented languages allows different parts of an application's code to be created by different programmers. The programming leaders assigned different responsibilities to different members of the team, and teams. In effect, utilizing a reuse-based programming language facilitated dispersed and collocated collaboration.

ieCollab programming infrastructure was set up to facilitate code reuse. The programming team created programming standards which institutionalized the format and content of each class, providing the same level of detail in each class (see Appendix A). As a result, geographically and temporally separated collaborators can effortlessly comprehend each other's work.

Test Cases

Because a complete product was not delivered, the testing team could not perform a complete test of ieCollab. However, the test team delivered test cases for both the transaction management and meeting management functionalities, and these can be reused in next year's development cycle.

In addition, the test cases focus on both low-level and high-level testing. That is, general functionality tests and tests for specific methods are both provided. The general test cases can be reused after all versions of the product is completed. Specific test cases can be reused to ensure overall completeness as well.

5.3.2.2 Functional Products

The experience legacy of ieCollab inherited by next year's development team are the functional team plans. Each team was required to create a team plan detailing objectives and activities of the team according to IEEE standards. By standardizing the type of information required in each plan, the level of detail in each document is at least the minimum required for comprehension by separate parties. Although the lifecycle model may differ between development cycles, the phases within each cycle and the activities by each group may be similar. Thus, next year's team may reuse functional plans to facilitate knowledge collaboration between functional teams.

IEEE functional plan requirements also provides a template of the layout of the document. By following this layout, information within the document can be searched for and comprehended more quickly. Thus, the size of the document may be mitigated by the structured, searchable layout.

5.3.3 Effectiveness of web repository

Central to any reuse program is a systematic software development library (SDL). The SDL stores all knowledge and experience created in a cycle to be used by the next cycle. In addition, the SDL supports distributed collaboration because all documents are stored in a central database and is accessible to all regardless of physical and time barriers. In ieCollab, the web repository is used as a central information center for past knowledge and present assets. In multi-year collaboration, ad-hoc reuse of the web repository limited its effectiveness in passing knowledge. In distributed collaboration, the presence of web repository was formalized into activities and procedures and thus used more effectively.

Repositories, when used to its maximum potential, are effective in enforcing collaboration. Repositories are ieCollab's software asset library. This project has two main web repositories: the project web repository at http://collaborate.mit.edu/1.120.html to hold draft documents and the CM-KM Frozen draft repository located at http://cee-ta.mit.edu/1.120/CM/index.html to hold final drafts. However, each is currently designed as a somewhat structured "drop-off" for documents to be shared with others, limiting its reusability. The third repository is a code library to provide a central database for the created code.

5.3.3.1 Project Web Repository

The project web repository, discussed in section 5.19.21, was not an effectively used source of reuse knowledge. One reason may be due to the site's format. Although archived documents were separated from current documents, all archived documents were displayed in one section and in a list in no specific order.

Because not enough detail is provided in the description for users to know what is provided in the content of each document, users had to search and read through every document for the desired information, if present at all. Motivation to use the web repository, and the knowledge in it, decreased.

Accessing the web site for specific information was time intensive. Users were first introduced to the web repository, then prompted to click to enter username/password. Next, users had to input username/password and click to enter. Then users had to click once on the document desired, click a second time when a tiny icon appeared on the icon screen, and click to save the document or click to view. If the user clicked to view, it was necessary for the user to click a document icon again to preview the document. The high clicking ratio to target output acted as a barrier to accessibility and thus document reuse. Users became weary of using the site.

The project web repository violated several quality factors presented in section 4.2. The high click ratio, the number of clicks required before the user can obtain the desired information, and the unstructured storage of the website violated the simplicity principle. In addition, the website did not provide clear direction in usage. As a result, users attempting to comment on a document placed comments for one document in several different locations. Thus, the project web repository is a low reuse medium.

Within the same development cycle, the ieCollab web repository provided a standard medium to share information with the dispersed project team. The website is partitioned based on teams and documents generated by each team were uploaded and accessible to all. The effectiveness of this website was greater than for multi-year collaboration because teams were forced to apply the research of the previous phases. However, as a medium to ease reuse, the project web repository impeded ease of access to reuse assets and incurred the same frustrations as mentioned above for multi-year reuse. One additional frustration is that the web repository is record of work in progress. Thus, many drafts of the same document were posted. Participants felt frustrated not knowing what the most recent draft was and how much of the document is changed.

5.3.3.2 CM/KM Final Draft Repository

The CM/KM web repository served to clarify the final documents from the drafts. The CM/KM web site provided access only to the frozen documents of the current development cycle. Similar to the project web repository, this site was organized by team and final team plans, presentations, and other documents were presented in a list. No other information concerning the content of the document, version number that was frozen, or date. The library may have been useful to inter-team collaboration in accessing a prior phase's documents. However, each phase overlapped and sometimes occurred concurrently. As a result, teams rarely accessed this web repository. The site may be more useful to next year's development team.

5.3.4 Barriers to Reuse in ieCollab

The ieCollab project did not obtain a high level of reuse during development. Barriers to reuse may be attributed to technical and cultural factors.

- **Large time investment.** Understanding reuse assets developed by another party is time consuming. The user must intimately understand how an asset modifies outputs to product the desired result. In addition, a comprehensive understanding of the asset is required before reusable modules can be integrated. Next, the creator of a reuse asset must understand all the context to which his asset may be used. He needs to create assets with enough general knowledge of the system to be abstract, yet detailed enough to be useful. Consequently, reusing assets requires great time investment, impossible in the rapid development time required in this project.
- **Large resource investment.** Both human and technical resources are required to instill a strong reuse program. Human resources are needed to be responsible for the additional activities required in a reuse program. Technical resources, such as a reuse database, are needed to support these activities.

- **Not Invented Here Factor.** Although work and functional products were available to the project team, member did not reuse much of the available assets. The Not Invented Here Factor states that developers are more eager to create an entirely new application that build on existing systems. ieCollab team members were eager to begin solving the collaboration problem, until they realized they had to integrate CAIRO into the system.
- **Non-dedicated Technology.** Technical resources, such as a dedicated database and server, is necessary for reuse to enable collaboration in engineering. Although a web repository was available for the project team, the server was often offline and members could not access the assets stored in the repository. Having a dedicated server to is necessary to leveraging knowledge to all collaborators.

However, each of these factors may be attributed to culture and an automated process of reuse may potentially weaken these barriers. Suggestions for a strong reuse program are discusses in Chapter 6.

5.3.5 Capability Maturity Level

ieCollab is only capable of Level 2 maturity. Although a semi-structured reuse program exists in ieCollab, reuse practices are still voluntary. The process model and suggestions for improvements is not necessarily implemented in the next cycle. Assets are not consistently available and reused optimally. Finally, the web repository is not optimally designed for reuse engineering. In summary, feedback does not modify or optimize reuse activities in ieCollab.

5.4 Problems in ieCollab

Although the ieCollab team's end-product was different from the CAIRO team, this team encountered similar problems in the development of ieCollab. These problems include:

- **Lack of specific technical knowledge.** Similar to the 1998 CAIRO team, the entering 1999 team had a strong civil engineering background but lacked information technology and computer science experience.
- **Lack of team interaction and coordination.** Collocated and dispersed teams did not effectively interact during the development phase. Several team members in PUC and CICESE left the project gradually during the year.
- **Lack of knowledge of resources and previous application.** The current project did not incorporate CAIRO in ieCollab. ieCollab is a completely new product that did not build on previous knowledge.
- **Lack of motivation.** A nine-month cycle is a relatively short development time to complete and launch a full-scale application. In addition, this research was only one of several responsibilities carried by team members. Without a clear objective, the team lost focus and motivation.
- **Lack of dedicated technology.** Often, team members could not contact each other through a stable medium. CAIRO could not be used successfully. Video conferencing and phone conferencing were limited due to expense. ICQ and NetMeeting provided an awkward form of communication. And collaborating through email incurred a time delay.

These problems may be mitigated by implementing a strong reuse engineering program in the software development cycle. Section 5.4.1 and section 5.4.2 discuss immediate improvements to the development process.

5.4.1 Distributed Collaboration

A strong software reuse engineering program may mitigate the distributed collaboration problems encountered by the CAIRO and ieCollab teams. Following are problems stated previously and suggestions for improvement supported by reuse activities.

- **Web Repository.** In addition to housing reusable assets, the web repository should also act as a central database of information for the entire project. Using the web repository as a central database of information may mitigate the lack of communication and interaction between teams and within teams. Scheduled meetings and completed meetings logs may be posted to the site to coordinate interaction between teams and within functional teams. In addition, decisions resulting from these meetings can then be shared with the entire team.
- **Templates.** Creating specific templates for each type of work product and functional product ensures enough detail is provided for collaborators. In addition, users of assets would also then know the amount of information required in each asset.
- **Assets.** Complete assets enables rapid development of an application. Existing functional assets could be reused by next year's class and not recreated. Pre-existing code packages could be integrated in the current project so that entire applications do not have to be recreated. Finally, existing work products is often applicable to many projects and should be reused. Because the quality of each of these assets have been tested, the time spent in coding, testing, and quality assurance is lowered, lowering overall development time.

5.4.2 Multi-year Collaboration

Effective use of multi-year collaborational assets may mitigate problems encountered during the development of ieCollab. Following are specific opportunities for multi-year reuse in ieCollab.

- **Provide Technical Knowledge.** The knowledge gained in one cycle can be passed on to the next cycle to decrease the "learning" time. Although learning basic knowledge is always required in any cycle, knowledge required to complete the project must be shared between parties facilitate efficient development.

- **Provide Process Familiarity.** Different from technical knowledge, process knowledge is experience gained from in one cycle. The lessons learned is invaluable and should be implemented as improvements to the next cycle until an efficient method is achieved.
- **Decrease Development Time.** Multi-year reuse supports rapid development because existing assets do not have to be recreated in each development cycle. In addition, the core product is built on and less development of the product is needed.
- **Create Shareable Assets.** Shareable assets encourage the user to develop assets comprehensible to all present and future collaborators.

A strong reuse program is a knowledge legacy that lowers the temporal barrier to reuse collaboration. Use of past assets can rapidly educate the current project team on available resources and technical knowledge required in a reuse. Thus, a strong reuse program encourages multi-year collaboration.

5.5 Summary

This chapter examined ieCollab as a case study for distributed collaborative software development. Reuse was used informally in this project and thus enforced informally. The project team encountered several problems that may be mitigated by formally implementing a strong reuse program. The next chapter will suggest a future reuse implementation scheme and the benefits from integrating this scheme in ieCollab.

Chapter 6

Future Development

Chapter 5 discussed the role of reuse in a collaborative software development environment. Although effectively implemented, a formal method of reuse needs to be installed before the full benefits to reuse is realized. This chapter will suggest formal reuse activities based on the problems encountered in the development of ieCollab. It will conclude with the benefits that may be gained if activities are implemented.

6.1 Future Development

The problems encountered during the development of ieCollab maybe mitigated by implementing a greater degree of reuse in the lifecycle. Following are suggestions for improvements to the next cycle of ieCollab.

6.1.1 Formal Integration of Reuse Technology in Development Process

Ad-hoc reuse is the least effective method of reuse implementation, as shown in chapter 4. Reuse activities must be imbedded in the development cycle to force member's to reuse assets. Werner, Travassos, and da Rocha in 1998 proposed a Software

Development Environment (SDE) to automate the reuse activities. The SDE is computational system to construct, manage, and maintain a software product. This model may be adapted to support a collaborative engineering environment, shown in Figure 6-1. The SDE emphasis is on automating processes and decisions in the development cycle.

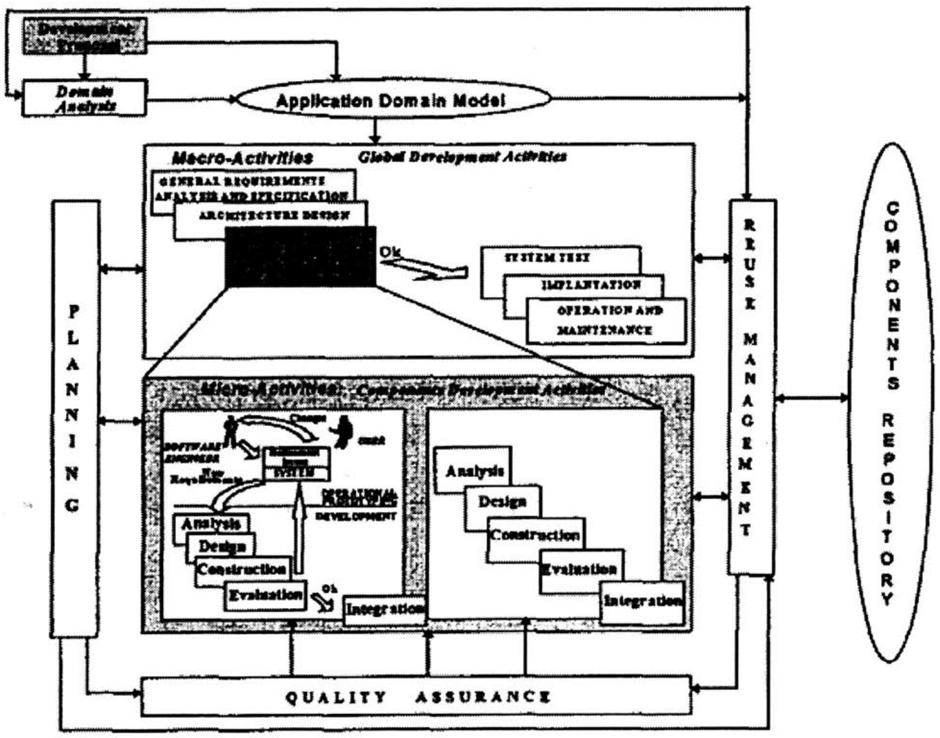


Figure 6-1. Software Development Environment (Werner, Travassos, and da Rocha 1998).

Under the proposed SDE framework, the development process combines both the incremental and prototyping models for reuse. Phases in this lifecycle are:

- **Development Proposal.** This phase analyzes the existing system and the proposed system for implementation feasibility. In ieCollab, this phase would have connected CAIRO functionality and niche to ieCollab.
- **Application Domain Model.** This phase develops a generic model of the application domain to be reused and detailed in later cycles.

- **Global Development Activities.** This phase provides a general idea of the system, yet with adequate detail to identify components and create incremental development. Activities in this area are General Requirements Analysis and Specification, Architecture and Design, System Test, Implantation, and Operation and Maintenance. Based on the application domain model, activities in this phase identify possible reuse opportunities.
- **Component Development Activities.** This phase encompasses the conventional software development cycle. Activities in this area include: Analysis, Design, Construction, Evaluation and Integration.
- **Support Activities.** Project Management, Quality Assurance, and Configuration Management activities are unchanged in an SDE. However, a new role is created to support reuse activities in the cycle, including creating new assets and integrating new assets. The Reuse Manager should have intimate knowledge of the storage, retrieval and search functions of the SDL. The Reuse Manager may also act as the interface between the SDL and users. Because this role enables knowledge sharing, it may be integrated with Knowledge Management.

6.1.2 Suggest Improvements to the Web Repository

In the previous chapter, teams noted both cultural and technical difficulties in accessing the repository. The current web repository is a basic document uploading and downloading site. In a reuse environment, the web repository is a software development library that supports all asset reuse. The web repository should be a more aggressive presence in the development cycle. John and Spiros-Theodoros introduce a reuse repository to store and retrieve all the assets generated in a software development lifecycle (1996). This model supports a collaborative engineering environment.

The proposed web repository is based on the software development phases and a hierarchical representation of the information stored in them. The basic element is the Reusable Object (RO). Every RO belongs to an RO class, which represents the phase in the lifecycle. All objects generated from each procedure can be modeled and represented in the object repository. RO classes are clustered according to corresponding lifecycle phases. Figure 6-2 shows the six levels in the information hierarchy.

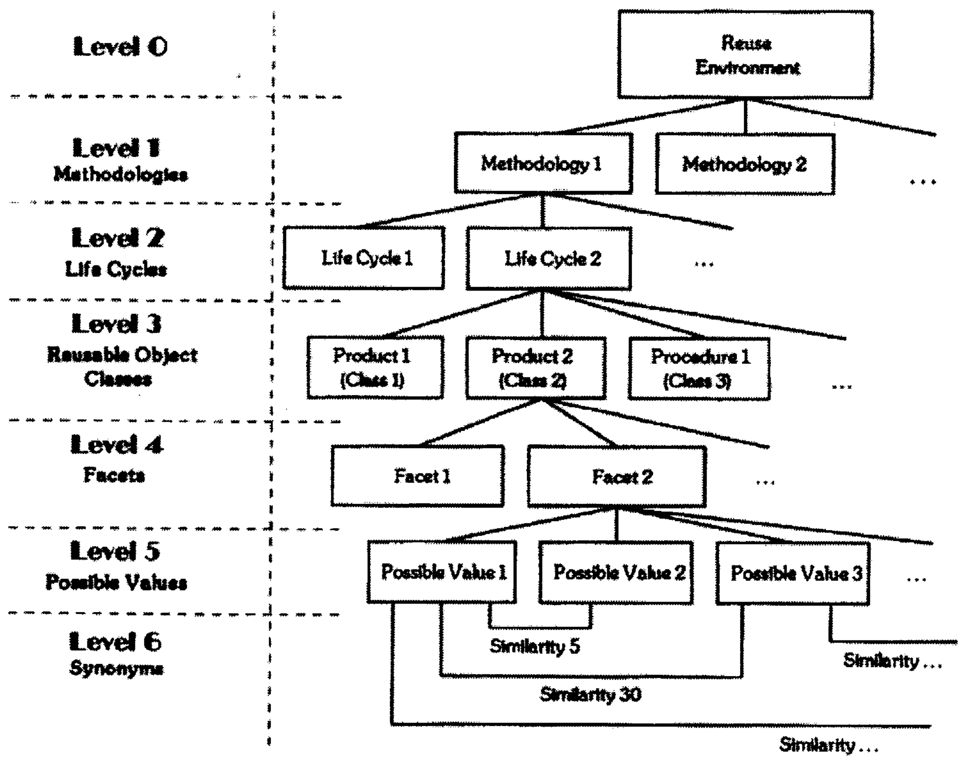


Figure 6-2. Proposed information hierarchy (John and Spiros-Theodoros 1996).

The levels of the hierarchy are the basic assets in a development process. They are

- **Methodology and Life Cycles.** The basic development process and thus the basic grouping concept in the repository.
- **Reusable Object Classes.** These are the basic applied procedures and products created or exchanged in a lifecycle process.

- **Facets and Possible Values.** The unlimited number of identifying characteristics of an RO class. However, each facet assigns has one and only one possible value to an RO class.
- **Synonyms.** Linear relationships between RO classes that can extend different procedures and products of the lifecycle.

However, not all assets are reusable in every project. Reuse maturity is a metric that defines the range of applications an object can be applied to. Greching and Biffl in 1993) define three general levels of reuse maturity:

- **Level 1: Project-wide Reuse.** The lowest RM, the component can only be reused in the project within which it is created.
- **Level 2: Field-wide Reuse.** The component can be reused in the project it is defined for and similar projects. This is the intermediate level of reuse.
- **Level 3: Global Reuse.** A component that can be reused in any situation is the highest level of reuse.

The organization of RO classes is shown in the figure below. The graph is a three-dimensional organization scheme with the lifecycle on the horizontal axis, complexity on the vertical, and versioning on the third. The simple algorithm for RO storage and retrieval is shown in the Figure 6-3.

Searches are conducted in two phases. The facets are searched for a complete match of the search criteria. If no objects are found, then the facets are searched for semantically similar possible values to the criteria. The search function can be completed as many times as the user desires.

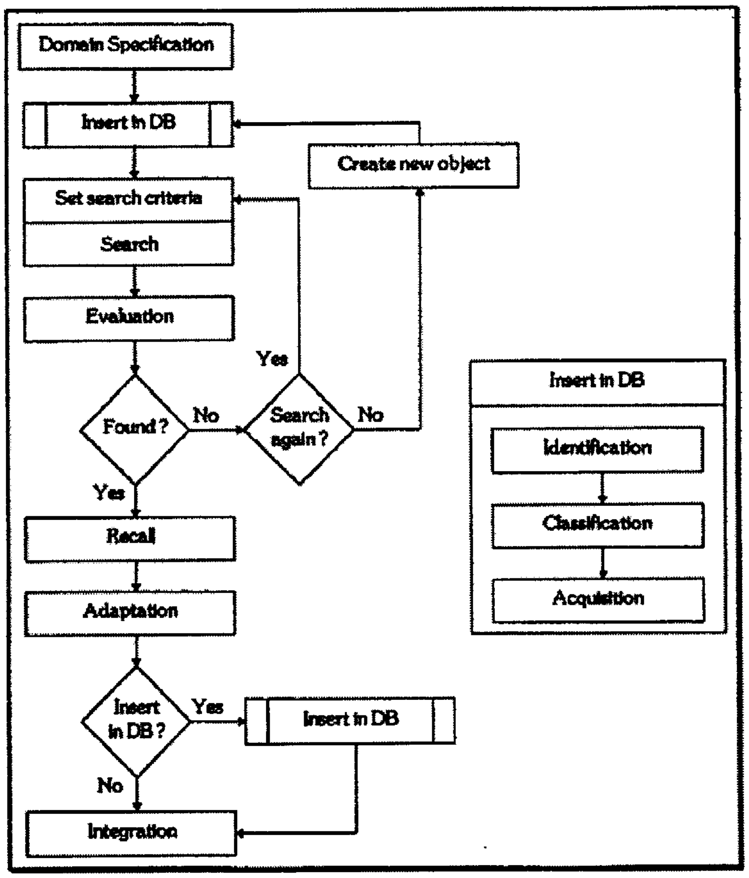


Figure 6-3. RO storage and retrieval algorithm (John and Spiros-Theodoros 1996).

6.1.3.1 Preconditions

Without a dedicated line, however, a better organized repository is useless. ieCollab's team members reported constant server unavailability error when attempting to access the website. Because of the amount of interaction this new architecture requires, it is important to have a dedicated line during any time of the day.

In addition to technical difficulty, organizational infrastructure is necessary to support and enforce software reuse. Users need to be trained in developing storable assets and in retrieving assets for use. However, repositories are educational tools and can be used to enable development of complex systems. Thus, repositories may also elevate an organization to the next maturity level.

6.1.4 Engineering of Future Assets

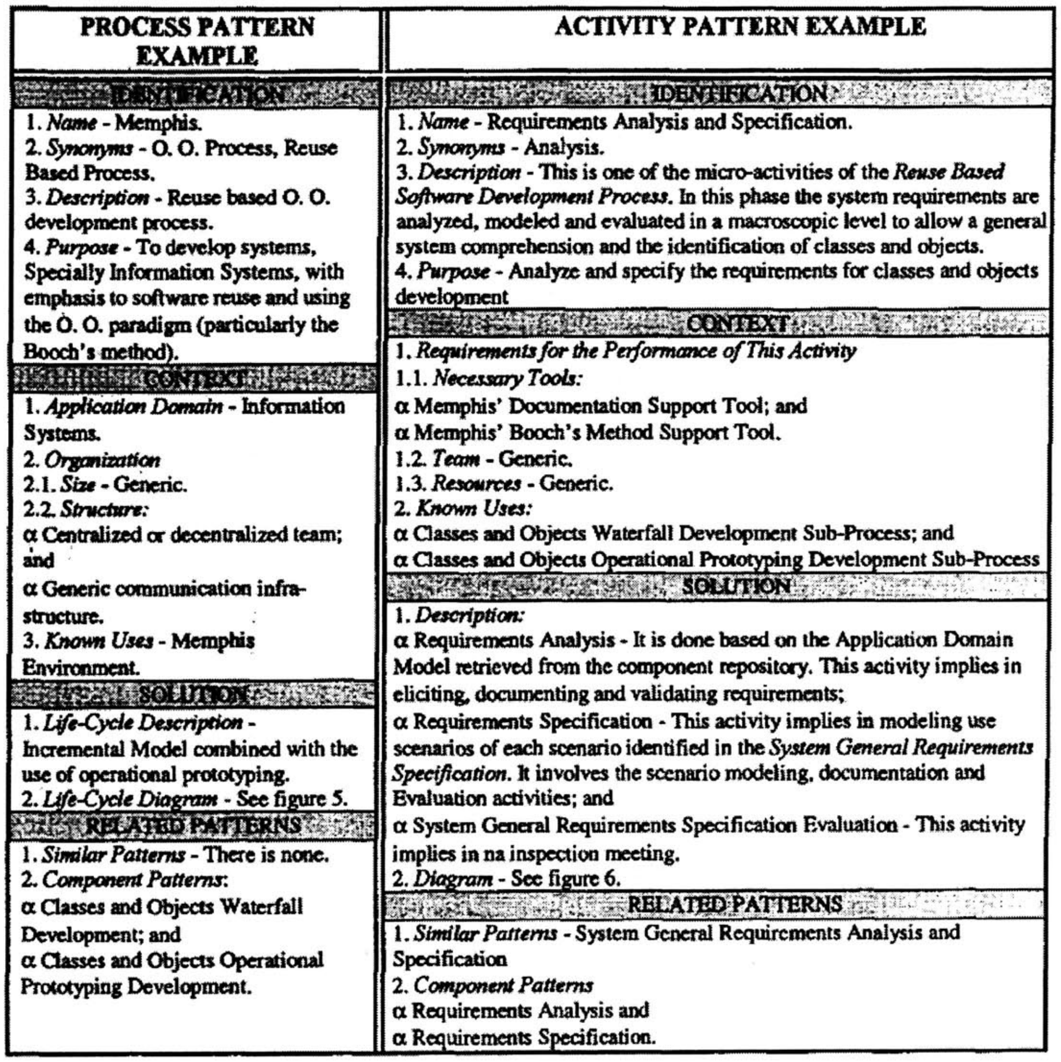
One of the greatest barriers to reusing assets in ieCollab was the team member's lack of motivation to spend the time to read, understand, integrate, and adapt existing components to the software development cycle. Most assets were semi-structured and followed a specific template created by the project management team. However, the lack of complete structure impeded quick comprehension and information extraction. In addition, the development team had no prior experience or knowledge of the type and extent of information to provide in a document. As a result, documents did not provide enough detail for the next phase of the cycle and will not provide enough detail for next year's development team to be used as reference.

In general, assets either are incomplete in detail, difficult to read, or both. Werner, Travassos, and da Rocha in 1998 suggest hypermedia and pattern technologies to decrease the psychological barrier to reuse. Patterns provide an information structure to ensure the document is complete. That is, to be accepted by the web repository, documents must fit the predetermined pattern for completeness. Hypermedia tools reduces the document's "unattractiveness" by taking advantage of media resources such as videos, graphics, dialogs to present the information.

An online, automated asset uses patterns to organize the document and hypermedia is used as the base technology. The document should contain four types of information and is illustrated in Figure 6-4 (Werner, Travassos, and da Rocha 1998).

- **Component Identification.** This section includes component name, synonyms, supercomponent and subcomponent references, purpose and problem description.

- **Context.** This section includes the application domain described in detail, applicability, and known uses.



PROCESS PATTERN EXAMPLE	ACTIVITY PATTERN EXAMPLE
IDENTIFICATION	IDENTIFICATION
1. *Name* - Memphis. 2. *Synonyms* - O. O. Process, Reuse Based Process. 3. *Description* - Reuse based O. O. development process. 4. *Purpose* - To develop systems, Specially Information Systems, with emphasis to software reuse and using the O. O. paradigm (particularly the Booch's method).	1. *Name* - Requirements Analysis and Specification. 2. *Synonyms* - Analysis. 3. *Description* - This is one of the micro-activities of the *Reuse Based Software Development Process*. In this phase the system requirements are analyzed, modeled and evaluated in a macroscopic level to allow a general system comprehension and the identification of classes and objects. 4. *Purpose* - Analyze and specify the requirements for classes and objects development
CONTEXT	CONTEXT
1. *Application Domain* - Information Systems. 2. *Organization* 2.1. *Size* - Generic. 2.2. *Structure:* α Centralized or decentralized team; and α Generic communication infra-structure. 3. *Known Uses* - Memphis Environment.	1. *Requirements for the Performance of This Activity* 1.1. *Necessary Tools:* α Memphis' Documentation Support Tool; and α Memphis' Booch's Method Support Tool. 1.2. *Team* - Generic. 1.3. *Resources* - Generic. 2. *Known Uses:* α Classes and Objects Waterfall Development Sub-Process; and α Classes and Objects Operational Prototyping Development Sub-Process
SOLUTION	SOLUTION
1. *Life-Cycle Description* - Incremental Model combined with the use of operational prototyping. 2. *Life-Cycle Diagram* - See figure 5.	1. *Description:* α Requirements Analysis - It is done based on the Application Domain Model retrieved from the component repository. This activity implies in eliciting, documenting and validating requirements; α Requirements Specification - This activity implies in modeling use scenarios of each scenario identified in the *System General Requirements Specification*. It involves the scenario modeling, documentation and Evaluation activities; and α System General Requirements Specification Evaluation - This activity implies in na inspection meeting. 2. *Diagram* - See figure 6.
RELATED PATTERNS	RELATED PATTERNS
1. *Similar Patterns* - There is none. 2. *Component Patterns*: α Classes and Objects Waterfall Development; and α Classes and Objects Operational Prototyping Development.	1. *Similar Patterns* - System General Requirements Analysis and Specification 2. *Component Patterns* α Requirements Analysis and α Requirements Specification.

Figure 6-4. Information types (Werner, Travassos, and da Rocha 1998).

- **Solution.** This section includes the problem solution within the context of the domain, implementation issues, source code and limitations. This section may include code packages, test cases, or explanatory text.
- **Related Components.** This section includes similar components and their distinctions as well as which components must be joined together.

Separating information into different layers has the advantage of quick information extraction. Existing documents are difficult to reuse because content organization varies even among assets created by the same functional team. Under this new organization, documents can be quickly and easily scanned for vital information.

6.1.5 Reverse Engineering

Existing assets have not been used effectively because they were not created to be reused. Reverse engineering is used to create reusable assets from existing assets. Following the steps outlined in chapter three, assets from 1997, 1998, and the current development cycle, should be abstracted for reuse opportunities. Document templates may be abstracted to inform the following year's development team the level of detail and type of knowledge needed in each asset. Class structure should be made reusable so code can be reused and instantiated in different cycles in addition to providing the next cycle's developers the structure of the component. This is important if the code is to be integrated in each subsequent development.

In addition, the storage scheme was not designed for reuse. Existing assets should be reversed engineering for reuse and stored in the software development library outlined above. This would also encourage developers to reuse assets if it is easier and faster for developers to reuse current assets then create their own.

6.2 Technology to Support Reuse Activities

Currently, no comprehensive tools exist to support the integrated reuse development. However, a variety of systems support specific aspects of reuse technology. Currently available toolsets are REBOOT, OPSEN, and RESOFT.

REBOOT is a database management system that provides a repository, a retrieval/search, component's metric and test services, a C++ code adapter, a browser and reengineering space. Version 2 is available on the market.

IPSEN is the Integrated and incremental software Projects Support Environment. IPSEN is designed as an integrated framework designed to link connects documents generated in this environment. It provides an editor for the development and maintenance of a classification scheme and three retrieval methods.

RESOFT is a Reusabiltiy-based Software Development system. This system provides automatic or interactive generation of software specifications, component storage, query formulation, components search and browsing. RESOFT is runs in a UNIX environment.

6.3 Feasibility

Although the benefits to reuse in ieCollab's development cycle are numerous, the inherent structure of the project may not facilitate reuse. Reusing and identifying reuse opportunities incur a steep learning curve. Thus, in a limited development time, extensive and complicated reuse structure may detract from precious development time and discourage reuse. Reuse activities are an additional layer of knowledge that must be acquired for successful completion of this project.

In addition, the great number of resources required to promote reuse may limit implementation. A dedicated connection to a "always on" server providing a software development library will probably constrain already limited amount of resources.

Furthermore, unless reuse engineering is automated in the development cycle, software users must be provided with incentives to practice reuse. Thus far, these suggestions have not provided a strong automation program. Automating reuse is not likely to occur in ieCollab because of the great initiation costs and steep learning curve.

students that participate in the project

Therefore, the ieCollab project team may only find it feasible to implement reuse activities suggested here on a smaller scale. A web repository currently exists, and should be developed into a software component library. Enforcement of reuse may not be automated, but should be encouraged externally through cooperative incentives, thus gaining the benefits of reuse.

6.4 Benefits

Although software reuse may incur an initial learning and cost curve, the benefits to the overall project outweighs these drawbacks. Discussed in Chapter three, the benefits may also apply to reuse in ieCollab as well.

- **Rapid development.** In a rapid development environment, reuse assets already exists. The project team therefore does not have to recreate the assets in every cycle.
- **Lower costs.** Resources invested in creating reuse assets are amortized over several project lifecycles. Technology used to create assets in one lifecycle may be expensive. By reusing these assets, the development team does not have to incur the same cost in subsequent development cycles. Thus, cost per project is lowered.
- **Improved product quality.** Reuse assets have already been tested and assured for quality. Reusing these assets improves overall product quality because past assets, core to any project, have already been tested. Improved product quality also facilitates rapid development.
- **Increase organization's process capability.** Implementing and utilizing feedback mechanisms through software reuse increases an organization's level of maturity. ieCollab, in achieving another level of maturity, may thus be able to better predict current capability and productivity.

6.5 Conclusion

When used optimally, software reuse supports distributed collaboration by enforcing a standardized medium of communication between dispersed parties. Asset standards are necessary for storage and retrieval in a SDL and thus force all parties to force assets in the same manner. Hence standardizing assets for easier comprehension between parties.

Within multi-year collaboration, the software development extends the life of existing assets to enable collaboration between temporally dispersed parties. Reusing assets increases productivity by shortening the development time and allows teams to build on each other's application in order to solve more complex problems. In addition, reuse assets are modular and are thus able to adjust for emerging technology. A strong reuse engineering program should be strong integrated in the development cycle.

This thesis provides a general overview of possible reuse opportunities in supporting distributed collaboration. The suggestions for improvements should be taken as an introduction to activities. Additional observations and implementation is required before benefits to reuse can be truly assessed. However, this thesis does suggest a method for initiating reuse engineering, thus improving overall collaborative software engineering.

References

Arango, G. (1994) "A Brief Introduction to Domain Analysis." *Proceedings of the ACM Symposium on Applied Computing 1994.*

Belanger, David and Krishnamurthy, Balachander. (1994). "Practical Software Reuse: An Interim Report." *IEEE.* Los Alamitos: IEEE Computer Press.

Biggerstaff, Ted J. and Perlis, Alan J. (1989). *Software Reusability. Volume I: Concepts and Models.* New York: Addison-Wesley Publishing Company.

Biggerstaff, Ted J. and Perlis, Alan J. (1989). *Software Reusability. Volume II: Applications and Experience.* New York: Addison-Wesley Publishing Company.

Card, Dave and Comer, Ed. (1994). "Why do so many reuse programs fail?" *IEEE Software.* Los Alamitos: IEEE Computer Press.

Chen, Yufeng; Howe, W. Gerry; and Warsi, Nazir. (1997). "A Multimodeling Framework for Complex Software Reuse". *IEEE.* Los Alamitos: IEEE Computer Press.

Chu, William and Yang, Hongji. (1996). "A Formal Method to Software Integration in Reuse". *IEEE.* Los Alamitos: IEEE Computer Press.

Doublait, Stephane and Lissoni, Cristina. (1997). "Processes for Systematic Software Reuse." *IEEE.* Los Alamitos: IEEE Computer Press.

El-Solh, Wassim and Tan, Nhi. (2000). "Design Specification for ieCollab Version 1 Meeting Management." From http://collaborate.mit.edu/1.120.html.

Favaro, John. (1990). "What Price Reusability." Proceedings from the First Symposium on Environments and Tools for Ada. New York: ACM.

Feldman, Raimund; Geppert, Birgit; and Robler, Frank. (1999). "An Integrating Approach for Developing Distributed Software Systems—Combining Formal Methods, Software Reuse, and the Experience Base". *IEEE.* Los Alamitos: IEEE Computer Press.

Gall and Klosch. (1992). "Reuse Engineering: Software Construction from Reusable Components." *IEEE.* Los Alamitos: IEEE Computer Press.

Hao, Chen. (1999). "Design Specifiction for ieCollab Version 2 Transaction Management." From http://collaborate.mit.edu/1.120.html.

Hor, Joon. (1999). *Social Interaction in Collaborative Engineering Environments.* M.Eng Thesis, MIT Department of Civil and Environmental Engineering.

John, Halaris and Spiros-Theodors, Geropoulos. (1996). "Reuse Concepts and a Reuse Support Repository." *IEEE.* Los Alamitos: IEEE Computer Press.

Jones, T. Capers. (1984). "Reusability in programming: A survey of the state of the art," *IEEE Trans. Software Engineering*, vol. 10, no. 5. Los Alamitos: IEEE Computer Press.

Krueger, C.W. (1992). "Software reuse," ACM Computing Survey, vol. 24 no. 2. Vancouver B.C.: ACM Press.

Lim, Wayne C. (1994). "Effects of Reuse on Quality, Productivity, and Economics." *IEEE Software.* Los Alamitos: IEEE Computer Press.

Manasseh, Christian. (1999). *Project Management of Geographically Distributed Software Teams.* M.Eng Thesis, MIT Department of Civil and Environmental Engineering.

Morandin, Elisabetta; Stellucci, Gianfranco; and Baruchelli, Francesco. (1998). "A Reuse-Based software Process Based on Domain Analysis and OO Framework." *IEEE.* Los Alamitos: IEEE Computer Press.

Mili, Hafedh; Mili, Fatma; and Mili, Ali. (1995) "Reusing Software: Issues and Research Directions." *IEEE.* Los Alamitos: IEEE Computer Press.

Neighbors, James M. (1994). "An assessment of reuse technology after ten years". *IEEE.* Los Alamitos: IEEE Computer Press.

Paulk, Mark C.; Curtis, Bill; Chrissis, Mary Beth; Weber, Charles. (1993). "Capability Maturity Model for Software Version 1.1. Technical Report."

Prieto-Diaz, Ruben and Frakes, William B. Advances in Software Reuse. (1993). *Selected Papers from the Second International Workshop on Software*

Reusability, March 24-26, 1993, Lucca, Italy. Los Alamitos: IEEE Computer Society Press.

Sarshar, Marjan. (1996). "Systematic Reuse: Issues in initiating and improving a Reuse Program." *Proceedings of the International Workshop on Systematic Reuse, Liverpool, 8-9 January 1996.* Great Britain: Springer-Verlag.

Tracz, Will. (1994). "Software Reuse Myths Revisited." *IEEE.* Los Alamitos: IEEE Computer Press.

Tracz, Will. (1988). *Tutorial: Software Reuse: Emerging Technology.* Washington, D.C.: Computer Society Press.

Vincent, James; Waters, Albert; and Sinclair, John. (1988). *Software Quality Assurance. Volume I: Practice and Implementation.* New Jersey: Prentice Hall.

Werner, Claudia M.L.; Travassos, Guilherme H.; da Rocha, Ana Regina C.; de Cima, Alberto M.; da Silva, Monica F.; and de Vasconcelos, Jr., Francisco. (1998). "Memphis: A Reuse Based O.O. Software Development Environment." *IEEE.* Los Alamitos: IEEE Computer Press.

Zand, M.K. and Samadzadeh, M.H. (1994). "Software reuse: Issues and perspectives". *IEEE.* August/September 1994.

Appendix A: Programming Standards

Programming Documentation Standards and Guidelines Version 1.0

Programming Team

Prepared by Gyanesh Hari Dwivedi (ghd@mit.edu)

Date - 01/28/2000

References and Links

Programming Standards http://soils.ecn.purdue/~wepphtml/moses/standard/progstd.html
Programming Standards Document http://cs1.mcm.edu/~tmiller/fall99/programmingStandards.html
C++ Coding Standards http://www.btrust.com/services/training/CdStds/CPPCoding.htm
C++ Programming Standards: File naming conventions http://www.isip.msstate.edu/projects/speech/education/tutorial/standard/cpp_filenaming.htm

---

Outline

1. INTRODUCTION

The purpose of this document is to provide the ie-collab programming team the programming documentation standards that would be followed during the development cycle.

2.0 FILE HEADER

Each source file should contain the following:

1. The name of the file (In JAVA this is the name of the class itself)
2. A short description of the purpose of the code contained in the file
3. The date it was originally written
4. The original author
5. A modification Log (**Adding** or **Modifying** file content) containing
    a. The date when the modifications where made
    b. The person/persons who made the modification
    c. A short description of the modifications
6. Description and Listing of any particular hardware or software requirements that the code in the file might have for proper execution.

7. Description and Listing of modifiers used other than that specified in this document itself

8. **Any relevant design documentation reference**

3.0 Function Header

The following information should precede the function body

1. What action is performed by the function

2. Description of the parameters passed to the function as arguments

3. Description of the return value

4. Date on which it was first created

5. Original author

6. Any source that aided in the design of the function (Web URL, Book, Person etc)

7. A modification Log (**Adding** or **Modifying** file content)

    i. The date when the modifications where made

    ii. The person/persons who made the modification

    iii. A short description of the modifications

8. Description and Listing of any particular hardware or software requirements that the code in the file might have for proper execution.

9. **Any relevant design documentation reference**

4.0 File Name

The file name in JAVA is the name of the class. The name should be kept **SAME** or **VERY SIMILAR** to that in the design documents.

5.0 Class Layout

THIS SECTION LAYS OUT THE CLASS LAYOUT SPECIFICATIONS

1. The name of the class should be SAME or VERY SIMILAR (change should be documented in the file header) to that in the design documentation

2. The names of the class member variables and class functions should be SAME or VERY SIMILAR (change should be documented in the file header) to that in the design documents

3. In a file, define classes in the following order:

    - Public

    - Protected

    - Private

4. Break each class into subsections in the following order

    a. Constructors

    b. Operators

    c. Methods that access the object variables

    d. Methods that change the object variables

    e. Parent class method overrides

    f. Other implementation functions

5. Each of the class subsection should be separated by a **blank line** and also a **single line comment** indicating the subsection

6.0 Variable Declaration Comments

Provide a short description of the logical role played by each variable if it is not clear from the name

7.0 Use of i, j , k

The use of i, j, and k as variable names in loops(**DO WHILE** & **FOR**) should not be used except at places were the variable has **NO** significance of its own. Other than the loops

Even then a brief description of what the variable is looping about should be given at the place where the variable is being declared. Other than these loops they should never be used.

8.0 Naming Conventions

1. While naming the variable we will follow the JAVA and not the C manner of putting names

2. The following modifiers for the variables should be used

Data Type	Modifier
Boolean	f
Integer	i
Long	l
String	s
Double	d
Date/Time	dt
Currency	cur
Object	obj
Label	lbl
Panel	pnl
Text Box	txt
Image	img
Image List	il
Frame	fra
Horizontal Scroll Bar	hsb
Vertical Scroll Bar	vsb
Timer	tmr
Thread	thr
Index	idx
Button	btn
Database	db
Recordset	rs
Property	prop
Field	fld

3. A more detailed list of Data Types and their Modifiers would be available shortly in the next version 1.1.

9.0 STATEMENT GUIDELINES

- **General Statement Guidelines:**
    - Separate function arguments by a space to improve readability
    - Code infinite loops using an empty for statement **(for (;;))**
    - Place variable declaration one per line indenting additional declarations of the same type per line below.
- **do-while**
    - Place the do portion of the do-while statement on a separate line
    - Place the while portion on the same line as the closing brace
    - Do-while statement example:

```
        do
        {

        .....;

        } while(<expression>);
```

- **function definition** and **for** loops
    - Place the do portion of the function statement on a separate line as that of opening brace . **Your opening and closing brace should be in the same column number**

```
int Function()
{
 .....;
}
```