Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

COMPONENT-BASED SOFTWARE ENGINEERING

by
Zhiyuan Wang

To solve the problems coming with the current software development methodologies, component-based software engineering has caught many researchers' attention recently. In component-based software engineering, a software system is considered as a set of software components assembled together instead of as a set of functions from the traditional perspective. Software components can be bought from third party vendors as off-the-shelf components and be assembled together.

Component-based software engineering, though very promising, needs to solve several core issues before it becomes a mature software development strategy. The goal of this dissertation is to establish an infrastructure for component-based software development. The author identifies and studies some of the core issues such as component planning, component building, component assembling, component representation, and component retrieval.

A software development process model is developed in this dissertation to emphasize the reuse of existing software components. The software development process model addresses how a software system should be planned and built to maximize the reuse of software components. It conducts domain engineering and application engineering simultaneously to map a software system to a set of existing components in such a way that the development of a software system can reuse the existing software components to the full extent. Besides the planning of software development based on component technology, the migration and integration of legacy systems, most of which

are non-component-based systems, to the component-based software systems are studied. A framework and several methodologies are developed to serve as the guidelines of adopting component technology in legacy systems.

Component retrieval is also studied in this dissertation. One of the most important issues in component-based software engineering is how to find a software component quickly and accurately in a component repository. A component representation framework is developed in this dissertation to represent software components. Based on the component representation framework, an efficient searching method that combines neural network, information retrieval, and Bayesian inference technology is developed. Finally a prototype component retrieval system is implemented to demonstrate the correctness and feasibility of the proposed method.

COMPONENT-BASED SOFTWARE ENGINEERING

by
Zhiyuan Wang

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer and Information Science

Department of Computer and Information Science

May 2000

APPROVAL PAGE

COMPONENT-BASED SOFTWARE ENGINEERING

Zhiyuan Wang

Dr. Franz J. Kurfess, Dissertation Advisor — Date
Associate Professor of Computer Science, NJIT, Newark, NJ
Associate Professor of Computer Science, Concordia University,
Montreal, Quebec, Canada

Dr. D. C. Douglas Hung, Committee Member — Date
Associate Professor of Computer Science, NJIT, Newark, NJ

Dr. David Nassimi, Committee Member — Date
Associate Professor of Computer Science, NJIT, Newark, NJ

Dr. Pengcheng Shi, Committee Member — Date
Assistant Professor of Computer Science, NJIT, Newark, NJ

Dr. Yongming Tang, Committee Member — Date
Assistant Professor of Computer Science,
Fairleigh Dickinson University, Teaneck, NJ

Dr. Ke Liu, Committee Member — Date
Adjunct Associate Professor of Computer Science, Concordia University,
Montreal, Quebec, Canada
Senior Software Engineer, AT&T, Middletown, NJ

BIOGRAPHICAL SKETCH

Author: Zhiyuan Wang

Degree: Doctor of Philosophy in Computer and Information Science

Date: May, 2000

Undergraduate and Graduate Education:

Doctor of Philosophy in Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey, 2000

Master of Science in Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey, 1999

Bachelor of Science in Electrical Engineering,
Shanghai Jiao Tong University, Shanghai, P.R. China, 1995

Major: Computer Science

Publications:

Zhiyuan Wang, Franz J. Kurfess, "Component Retrieval with Neural Associative Memory," submitted to *European Conference on Artificial Intelligence*, 2000.

Jason T. L. Wang, Steve Rozen, Bruce A. Shapiro, Dennis Shasha, Zhiyuan Wang, Maisheng Yin, "New Techniques for DNA Sequence Classification," *Journal of Computational Biology*, Vol. 6, No.2, pp. 209-218, 1999.

Jason T. L. Wang, Bruce A. Shapiro, Dennis Shasha (editors), Zhiyuan Wang (contributor), *Pattern Discovery in Biomolecular Data: Tools, Techniques and Applications*, Oxford University Press, New York, 1999.

Zhiyuan Wang, Philip B. Johnson, Jason T. L. Wang, Cathy H. Wu, "Biological Software Development on the World Wide Web," in *Proceedings of the 6th International Conference on the Fuzzy Theory and Technology*, pp. 423-426, 1998.

Gung-Wei Chirn, Jason T. L. Wang, Zhiyuan Wang, "Scientific Data Classification: A Case Study," in *Proceedings of the 9th International Conference on Tools with Artificial Intelligence*, pp. 216-222, 1997.

This dissertation is dedicated to
my parents and my beloved wife

ACKNOWLEDGMENT

The author would like to take great pleasure in acknowledging his research advisor, Dr. Franz J. Kurfess, for his kindly assistance and remarkable contribution to this dissertation. He not only served as the author's research supervisor, providing valuable and countless resources, insight, and institution, but also constantly gave the author support, encouragement, and reassurance. Without his help, this dissertation could not have been published. Many thanks are given to Dr. Jason T. L. Wang, who helped the author establish a solid academic background. The author also thanks Dr. David Nassimi, Dr. D. C. Douglas Hung, Dr. Pengcheng Shi, Dr. Yongming Tang, and Dr. Ke Liu for actively participating in his committee. Special thanks are given to the author's beloved wife, who gave the author courage when he faced challenges, who gave the author inspiration when he solved problems, and who gave the author tremendous help when he needed it the most. The author can not thank too much for what she has done and sacrificed for him.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

1.1 Motivation

This decade, especially the late half of the decade, has witnessed a dramatic expansion of computer usage. Computer technologies appear in every aspect of human life. All kinds of traditional services, such as shopping, banking, trading, etc, are available via computer and Internet. The result is the rapid increase of demand of computer software in both size and complexity. Although a wide variety of methodologies has been recognized in the software engineering community, software development is still facing lots of problems, such as inability to deliver, exceeding budgets, missed due date, poor performance, high maintenance costs, etc. These problems partially result from the fact that software development methodologies do not emphasise reuse. For a long time, software development has been a process that starts from scratch and is conducted without reusing the result of other software development activities. To better understand the problem, let us compare the development of a car to that of a software system.

Examining the composition of a car, we will notice that the components of a car are made by different manufacturers. Car manufacturers do the design, buy components that they need from different component manufacturers, and assemble them into products. That way, the car manufacturers can concentrate on the design to ensure the overall quality of their product. They do not worry about how the components are made. They only need to know the specification of components to determine if the components fit in the overall design or not. They might be making some important components such as engine and transmission themselves. But no resource is unnecessarily wasted on

making other components such as brakes, shock absorbers, tires, etc. The quality and availability of such components is guaranteed by the component manufacturers. Some components can be bought off the shelf, and some need to be specially ordered from the manufacturer. The component manufacturer then applies the same strategy to develop their product. They do the design and buy necessary components from other manufacturers. This strategy recursively applies to each component manufacturer. The key to this strategy is the reuse of available components. It allows the designers and developers to focus on important issues and frees them from having to worry about the details of implementing each component.

On the other hand, during the development of a software system, such a strategy of reuse is rarely found. Typically, in the design stage of a software system, designers design the software system and decompose it without knowing or unwilling to find out if there are existing components out there that may perform part of the function that the system is looking for and can be integrated into the new system. The result is the difficulty of reusing components even though they may be available. In the development stage, the developers implement each component of the system without knowing that the same functionality may have been implemented by other developers. Few reusable components can be used during this stage and most of them are on the binary library level such as input, output routines. The lack of reuse of existing components in the software development processes results in the waste of human resources, difficulties in managing quality, and even failure to delivery.

The ideal development strategy of software systems should be the same as that of cars: buy components and build system, briefly "buy and build". Unfortunately the

current strategies to develop software system are "buy or build". On one hand, the software system is bought from outside sources, like Microsoft Office. On the other hand, the system is custom-made and usually built from scratch. The advantage of buying software system is the cost efficiency. Because the systems are sold as products, the development and maintenance cost is distributed to customers. Therefore the cost is relatively low. However, since it is aiming at a wide variety of customers, the software system is limited to be for general purposes. So it may not fit some customers' demand. As a matter of fact, lots of software systems, such as stock trading system, are not available as a ready-to-use product. The custom-made software system can meet customers' special needs. However, since that kind of software systems needs to be built from scratch, the development and maintenance cost is high.

First proposed by McIlroy [49], component technology has been seen as a promising approach to overcome the above problems. However the component technology seemed to go nowhere until recently, when it was on the verge of success. There are two main forces to resurrect component technology and make it possibly the most important milestone in the software engineering community. The first is technology evolution. During the thirty-year period since McIlroy had first predicted mass-produced components, people in the software engineering community had to face the reality that no technology was available to build and assemble software components and no platform independent languages were available. The technical difficulty impeded the component technology from being taken seriously. Now with the help of component middleware, such as CORBA [54], DCOM [64], and EJB [51][74], it is the first time that building a software system from reusable software components is feasible.

The other factor to make the component technology more appealing than ever is the high business pressure. With the fast growth of current economics, a typical software system's life span has been reduced to 2 years or so. New releases are coming out semi-annually or even quarterly. The traditional software development strategies are not sufficient anymore, since without reusing components a lot of time has to be wasted to implement functionality that has been implemented by existing software components. Component technology solves this problem by facilitating the "buy and build" strategy in software development. Software components can be bought at a relatively low cost compared to they being developed and maintained repeatedly by individual developer. The components then are composed into a software system that accommodates users' special business needs. With the help of component technology, a custom-made software system can be accomplished at a low price and in a short time.

1.2 Objectives and Outline

Component technology is not something that appears suddenly over a night. The concept of component technology has been there in the software engineering community for nearly thirty years. However, there is still no clear standard and guideline to software development on how to facilitate component reuse. There are some key issues to be solved before the component technology can be successfully used in practice. These issues are:

- Component planning
- Component building
- Component assembling

- Component representation
- Component retrieval

Component planning is the issue of how to decompose a software system into components such that it maximizes the reuse of software components. Component planning requires careful analysis in the application domain to extract the most common functionality and build components to achieve the functionality. Component building focuses on technologies that build components. This is the best solved issue among the five issues. Major programming languages now support the concept of components, such as JAVA. The de facto standards of component assembly are CORBA, DCOM, and EJB. In his doctoral dissertation [72], Y. Tang proposed a methodology to facilitate the automatic assembly of software components into software systems. His research result is a step towards software assembly lines that automatically build software systems. After components are built, they need to be distributed into persistent storage for future usage. Component representation and retrieval focuses on how to store and retrieve components efficiently and accurately.

The objective of this dissertation is to build an infrastructure to facilitate component-based software development. We will study four of the above five issues, namely component planning, component building, component representation, and component retrieval.

This dissertation has 7 chapters. In Chapter 2, an overview of the current state of component technology is given. The definition of a component is also justified. Chapter 3 proposes a software development process model for component-based software

development. The proposed process model focuses on the processes that facilitate the component technology in software development. It addresses how a software system should be planned and built to maximize the reuse of software components. The proposed software development process model conducts domain engineering and application engineering simultaneously to map a software system to a set of existing components in such a way that the development of a software system can reuse the existing software components to the full extent. A definition of team roles for component-based software development is also given to maximize the output of a component-based software development team in this chapter. Chapter 4 deals with the issue of component building. Unlike most of the current researches on component building which usually study the technologies to build components from scratch, an opposite and new approach is given to build components from existing software systems. This chapter also studies how to integrate components built from existing software systems with Internet applications to extend traditional services to online customers. In chapter 5, we propose a framework to represent components in component repositories. This framework is the starting point of our component retrieval method. In chapter 6, an efficient method of searching and retrieving components from component repositories is proposed. This method combines neural associative memory, information retrieval, and Bayesian inference technology. A prototype of a retrieval system is also implemented. Experiments are conducted to test the feasibility of our proposed method.

CHAPTER 2

COMPONENT TECHNOLOGY

More and more often when reading a computer magazine, one can not help noticing a word: component. It often appears in the context of component technology, component-based technology or component-based software engineering. However it is hard to find a definition of the term component. From the context, it is noticeable that sometimes a component is actually a chunk of code. More specifically it is a library function. People call it a component instead of a library function. Sometimes, a component is something you can use "as is" in your software system to utilize its provided functionality without having to bother to implement the functionality yourself. Sometimes a component is even a template or a framework for design works of software systems. Then what really is a component and what is component-based technology? We will try to answer these questions and give some background knowledge about component and component technology in this chapter.

2.1 Component Definition

What is the definition of component? The reality is there is no consensus on the definition of component. Here we list some popular definitions of component.

According to Clemens Szyperski [70], "a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition." This definition emphasizes deployment and explicit contextual

dependency. According to this definition, dependency is inevitable for software components.

Cited in [70], Philippe Krutche from Rational Software states: "a component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces." Here the functionality of a component is stressed. Krutche believes that components should be almost independent of other components. This definition does not stress the deployment characteristic of components.

In his famous book, "Software components with Ada" [7], Grady Booch defines: "a reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction." This definition focuses on the resemblance between components and modules. It does not further stress any unique characteristic of components.

Ivar Jacobson, in his classic book "Object-Oriented Software Engineering" [29], states: "By component we mean already implemented units that we use to enhance the programming language constructs. These are used during programming and correspond to the components in the building industry." In this definition, Jacobson implies that components are only used during implementation. This definition does not specify whether or not components should be independent of others, nor does it indicate the deployment of components.

Although the above definitions emphasize different aspects of component, they have one thing in common: all of them relate components to the implementation of a

software system. In his book "Software Engineering with Reusable Components" [63], Johannes Sametinger gives a wider concept of components: "Components are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status."

This definition addresses two kinds of components. One is at the implementation level such as individual functions to perform specific functionality. In addition to that, it explicitly claims that items that "describe" rather than "perform" functionality can also be viewed as components. This implies the qualification of design level work such as "framework" and "design pattern" as components.

We believe that Johannes Sametinger's definition is the most comprehensive one and best describes components. When describing the components that "perform" functionality, Clemens Szyperski's definition is the best one. It reveals the most distinctive difference between reusable component and reusable library functions. A component can be deployed as a part of a system whereas a library function has to be included in a program to provide its functionality. We give our definition of components by combining the definitions of Sametinger's and Szyperski's:

> Components are self-contained, clearly identifiable pieces that describe and/or perform specific functions with clear interface for reuse. A design level component is a unit to describe design issues with explicit domain restrains. An implementation level component is a unit for independent deployment and subject to third party composition with contractually specified interfaces and explicit context dependencies.

2.2 Why Components ?

The notion of components has been brought into the software engineering community for a long time. First presented by McIlroy [49], the idea behind software components is to implement components that perform some specific functions and when a software system is under construction, they can be used as building blocks to build software systems with ease and quality. Use of components is an well-adopted strategy in many mature engineering disciplines. However, being on the research papers for 30 years, the idea of reuse of components to build software system does not seem to get enough attention from the real world. Software engineers still build their systems in the traditional procedural way. Now under certain forces, the software development strategy has finally started to change to component based approach. These forces are:

- Available technology
- Business environment
- Observation from other engineering disciplines
- High flexibility

Available technology

Because of the dramatic increase of size and complexity of today's leading software systems, the old development strategies with very little reuse are no longer able to produce high quality software systems in a timely way. In order to resolve this so-called software crisis, software component reuse has gained a lot of attention from the software engineering community and is regarded as the best possible cure by some researchers.

Due to the characteristics of software engineering, however, it does not seem easy to apply component strategy to it. It might be the only engineering discipline that deals with something that does not have any visual information to the observers. After the idea was presented for the first time, it has been a tedious 30 years for people who tried to adopt the idea and failed. The reasons are complicated. One obvious obstacle is the technology difficulty. Not until recently, technology has become available to facilitate component technology such as assembling components. Thanks to the rapid development of computer technology there has not been a better time than today to adopt component technology. We have almost everything we need. All the technology that may facilitate the adoption of component technology is in hand. They may not be perfect but make component technology possible. Among these, the technology to build components and assemble those into a software system is the most important one.

For the past decade, software systems have been moving from centralized ones to distributed ones. Accordingly, technologies like CORBA, DCOM, and EJB were invented to facilitate the distributed computing environment by allowing part of a software system to communicate or to integrate with other parts of the systems which may be residing on different machines and developed under a totally different environment. This kind of technology makes it possible to assemble a software system from different components developed in different environments. These technologies also act as a pioneer to be studied as to how and in what way software components should be built and assembled.

Business environment

Another compelling factor of the merge of component technology is business pressure. In today's market, no software company can afford a development cycle of more than 2 years. A much shorter cycle, probably only of half to 1 year, is desirable by most companies. Apparently, the old fashion of building software systems from scratch is not preferable anymore. Reuse becomes the only way to get out of the woods. By reusing existing components, a lot of resources and time are saved. To save significant time, today's reuse can not just stay at library function level. That does not save much. Therefore, large size components with specific functionality are needed. Besides code reuse, the reuse of design idea such as design pattern and framework is also desirable.

Observation from other engineering disciplines

Another reason for using components stems from the observation of other engineering disciplines. As we stated repeatedly, other engineering disciplines have demonstrated the power of using components. When a car engineer builds a car, he decomposes the system into small components or subsystems, such as body, drive train, chassis and so on. He does not care about how they are built. He leaves these details to the engineers who are responsible for them. He only cares about what specification he needs for those subsystems and how to assemble those parts in a way such that the best can be pulled out of the configuration. For each engineer who is responsible for the subsystems, the same discipline is applied. For different cars, most of the same subsystem may be reused. This is the common case in car industry. However, one does not see the analog in software engineering. Software engineers do decompose a large system into subsystems, however

this decomposition is not systematic. Different engineers can decompose the same software system in different ways. The chance of reusing the decomposed parts in another system is therefore very slim. This directly results in the long development cycle of software systems. So from the lesson we learn from other engineering disciplines, component reuse is the best way to save resources and to lower the costs to build software systems.

High flexibility

The final force to give component technology an advantage over other methods is flexibility. In the traditional sense, people only have two ways, each on one extreme, to get a software system. On one extreme, they have to build a software system specifically for their needs. A development team is dedicated to that system. They usually do not get much help from other teams. The result is a customized system that very well fulfills the desired functionality. The development and maintenance costs are high, though. The other extreme of this scale is to buy a software system or a package. This way, the cost is relatively low and there is practically no maintenance burden at all. However the disadvantage is also apparent. Software being bought usually aims at a general purpose. It analyzes the market needs and finds the most wanted feature for the product and implements it. The reality that it can not implement all the wanted features makes it less preferable when a specific function is needed. When buying a software system, the buyers try to find the closest possible one to their needs. When a software system is bought, it is almost impossible to reengineer the software to make it serve better. Software development companies do not leave much room in their product for

modification. So what is the best way to get a software system? Our answer sits in the middle of the scale, buy the parts and build the whole.

The hybrid approach of buying and building a software system makes it appealing to the software engineering community. In this approach, a project is started like those starting from scratch. In the middle of the development, however, components are bought from third party component manufacturers either as a standard product or as a custom-made one instead of being built in the team. As a result, efforts are saved and bug-free components are available for assembly. Those components that can not be bought then are built by the develop team. Finally, when all the components are available, the assembly takes place and the final product is done. This approach improves the quality of software systems, saves time on the development, and significantly reduces the cost of maintenance. Since the functionality has been isolated in different components, when a problem occurs it is relatively easy to find the responsible component and pinpoint the problem in the component. When an upgrade or new functionality is needed, only the corresponding component needs to be examined, modified, or replaced.

2.3 Component, Object, and Module

From the description above, someone may have questions already in his mind. The concept of a component seems similar to that of objects and modules. Components and objects look alike in the sense that they are a piece of code that implements some function and hides its internal implementation. A module looks like a component because modules are a part or a subsystem of a software system and also perform some specific functions just like components. Sometimes people do use these terms interchangeably

and a clear distinction among these three is hard to draw. This section will explain the common and difference among components, objects and modules.

Component

As we pointed out before, the concept of component includes not only code segments that perform functions but also items that describe functions, such as system architecture and design framework. Apparently this high level describing ability is not found in objects and modules.

According to the definition by Clemens Szyperski, a component is an independent deployment item. This implies the characteristic of components of separating themselves from other components. In order to achieve this, a component has to be self-contained. The ideal case is that it does not contain anything defined or implemented in other component. Some object-oriented purists argued that components should show inheritance like objects do. Thus two components may be strongly connected. We do not support this claim. We believe that a component has to be independent and self-contained. This avoids environment incompatibility when components are deployed.

Components have a published interface for third party assembly. Object also has its interface for it being used by other objects. The difference between these two is that the interface of a component could be either a procedural interface or an object interface. The procedural interface is a function that when invoked by other components, the internal service of the invoked component can be accessed by the calling component. Object interface is defined on an object. The interface can not be reached without the existence of the object. An object interface is of course more preferable than the

procedural one. But as we will see in the later chapter regarding legacy system componentization, a procedural interface is inevitable, especially when we wrap a legacy system into a component and add an interface on the top. However there is a tricky way to turn a procedural interface into an object interface. A reference class can be defined and it has its own interface. In that interface, the procedural interface of a component is called. This way, the procedural nature of the component interface is hidden in the reference class. To other components, only the reference class and the associated interface are noticed.

Object

Object oriented technology so far has been the most successful programming technology in the software engineering community. Actually without it, component technology may be impossible. Objects share a lot of common characteristics with components. They all have interfaces to publish the services. They all encapsulate internal data and hide implementations. Differences between objects and components are equally clear. First, an object is an instance of a class. A component is not. An object has to be instantiated through its constructor. A component is a unit of deployment. The concept of instantiation does not exist. The concept of inheritance, we believe, should not apply to components, neither should the concept of polymorphism. Components are units of deployment, so it is more likely to be context constrained and dependent on the underlying component architecture. Objects do not have this kind of constraints.

Module

In the traditional software development strategy, a software system is decomposed during the design stage into several subsystems. The subsystems may be again decomposed. These subsystems are considered as modules in a system. Modules are close to components in the way that they are all parts of a system and they are composed together to form a complete system.

The difference between modules and components are rooted in the underlying component technology. In order to be reused and composed by a third party, a component should be as general and independent as possible. Generality gives components a better chance to be reused by other software systems and independence makes a component deployed independently. Modules do not have these characteristics. Modules are usually designed within a software system. Function separation is the main criteria when doing modularization. Software engineers usually do not keep generality and independence in mind. The result is that modules are usually not general enough and can not be reused in another systems without major surgery. Modules may have strong connection with other modules, which makes it impossible to deploy one module without taking other modules together. It is advocated by all software engineering books that when doing modular design, least possible interconnection or coupling is appreciated. However in the real world, because modules are usually only for a specific system, software engineers usually do not bother to try to get the least possible interconnection between modules.

Another difference is that components are for deployment and third party composition and assembly, so the interface of a component has to follow the syntax of the underlying component architecture, such as CORBA, DCOM, and EJB. Modules do

not have this kind of constraints and could be implemented in any form. Data encapsulation and hidden implementation is also an option for modules.

So from the above observations, we can see that object is the concept that focuses more on programming aspect. The emphasis of modules is on system architecture and function analysis. The component is between these two. It has some programming constrains and it also has some system architecture constrains. Moreover, as we pointed out, a component is a unit of independent deployment. This character is found neither in objects nor in modules.

2.4 Forms of Components

When we think of software components, we always think of program code, either a complete program or some part of it. Libraries in C and C++ and packages in Java are good examples of this kind of basic reusable units. When libraries and packages are included or imported in the program, they are used in an as-is form. Programmers do not change the code of libraries and packages. Adjustment is made through parameters.

However, a code segment is not the only form of components. We consider components as being able to both perform and describe functionality. So the following are qualified as components too:

- System and program architectural framework
- Prototype
- Data structure and algorithm
- Software life cycle process

System and program architectural framework

The framework may be the most important component to study and the most crucial factor in the success of component technology. A framework gives a direction for all following reuse procedures. According to [70], a framework is a set of cooperating classes, some of which may be abstract, that make up a reusable design for a specific class of software. This definition makes it clear that a framework is task oriented. A class of software that performs similar tasks could and should share a framework. Frameworks describe an overall design for a specific class of software, such as stock trading system, pay roll system, and so on.

A framework only specifies high level design, and leaves the implementation to the software engineers and developers. The famous model-view-controller (MVC) framework is shown in Figure 2.1. It is the most widely used framework for GUI design. Some frameworks come with their default or standard implementation for one or some parts of the framework. A framework in object-oriented technology usually specifies the interfaces of its parts and the interaction between each part. In component technology, however, the composition or assembly technology may have to be taken into account.

A framework can be hierarchically divided into sub-frameworks. A framework for a specific class of software may specify some lower level frameworks and gives specification for those sub-frameworks. Then these sub-frameworks have their definition and again may include some other frameworks that are one more level lower than them. The hierarchical framework structure is illustrated in Figure 2.2.

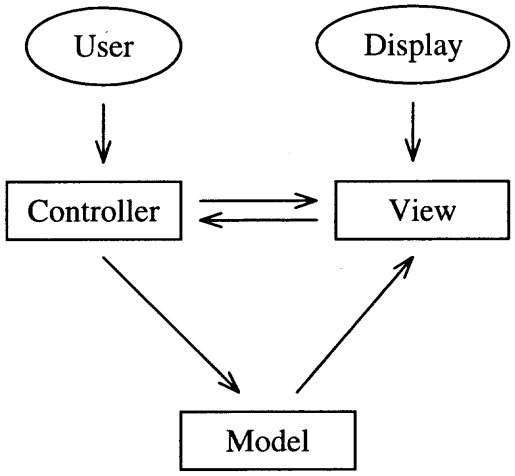

Figure 2.1 The MVC framework

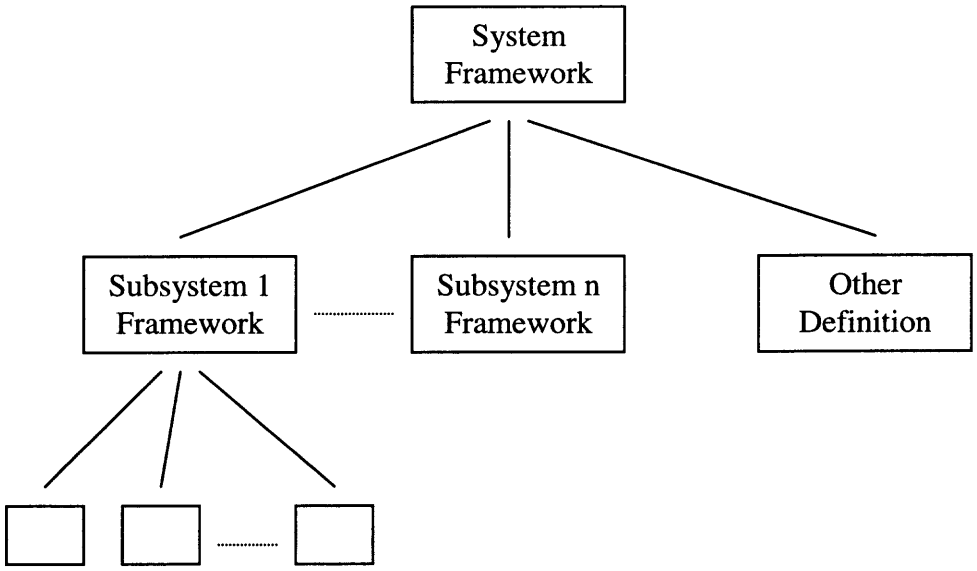

Figure 2.2 Hierarchical system framework

A program framework is like the system framework. The difference is only that a program framework applies to the program that implements functions. So a program framework can be thought of as a framework to specify how to implement a specific class of functions. For example, a program framework to authenticate user information may have an object to represent the user information, an object to represent authentication information and an interface to authenticate the user by comparing user information and authentication information.

Prototype

Prototypes are a widely used development method for reuse. Reuse of prototypes can eliminate the redundant cost of testing the feasibility of similar software systems. When a big software system or a part of it is analyzed, it is always simplified to its simplest form. Then a prototype can be built based on that. After the successful testing, the result of prototype testing can be served as the starting point of the software system and be stored. When another similar system is being developed, a good and quick development strategy can be given based on the results of the previous prototypes.

When a project needs to adopt new technology such as JAVA, CORBA, and EJB, there are not very many software engineers and developers familiar with them. It is hard to fully utilize the advantages of these new technologies when applying them for the first time. That is another scenario when prototype is coming into play. The value and feasibility of the new technology is studied by making a prototype. If it is successful, then the prototype can be reused for similar tasks.

Data structure and algorithm

It is easy to understand why data structures and algorithms are reusable components for experienced software developers. For a certain class of tasks, a certain class of data structures and algorithms is better than others. For example, people use an array to do bubble sort since it is easy to manipulate the swap operation. People use heap sort to sort big amount of data rather than using bubble sort since heap sort has lower time complexity. So for a class of similar tasks or functions, taking into account the restrictions for the implementation, such as time restriction and space restriction, a certain class of data structures and algorithms may be preferred. These data structures and algorithms are reusable components.

Software life cycle process

This is always used as an important software development strategy. The development of a software system always goes through several steps or stages during its life cycle to ensure the quality of software. Usually a standard procedure is defined. However, depending on the different natures of software systems, a class of software systems may not have to go through the same procedure as another class of systems does. Each class of software systems may come up with its own unique life cycle process. So software life cycle process can be reused for software systems with similar functionality.

2.5 Technical Issues

Component-based software engineering has changed the software engineering community dramatically. Although it is not formalized and standardized, people in the

industry recognize the value of the component-based approach. More and more companies have dedicated researchers to study the feasibility of component-based software engineering within the organization. Before we continue on, we would like to address some characteristics of component-based software engineering.

Most people are confused by the concept of component. They think components are only at the coding level which means components are a segment of reusable code. This idea seems normal but is wrong and may impede further develop of component-based software engineering. We believe that component-based software engineering is not restricted just to the implementation level. It actually consists of two levels. One is high level abstract component technology and the other is implementation level component technology.

The high level abstract component technology emphasizes the strategy and methodology of component technology. It takes on the way people analyze, design and implement software systems. It does not focus on the detail how the components are built and how they are assembled by the underlying building and assembling technology.

In reality, the high level abstract component technology is trying to answer questions as follows. How is a system designed at the interface level? Because components consist of interfaces that are responsible for communication among components, it is important to design a system in which components are communicating with each other strictly through interfaces. How to decompose a system so that further change and updates of its components is most likely to be efficient? How to build a component that is likely to be reused often and successfully? How to manage the generality and flexibility of components? What should the system architecture of

component-based software engineering be: layered or tiered? These are questions still under investigation. Some approaches have been given in [2] [17]. The result of research in this area will dramatically change the way software systems are built and the software engineers' primary roles and required skills.

The implementation level component technology focuses on how the proposed high level abstract component technology can be implemented. Predictably, the abstract component technology will not agree on every issue just like operating systems have their own belief on which scheduling algorithm is the best. As a result, implementation level component technology needs to solve technical difficulties and implement what the abstract component technology proposes. One of the issues for implementation level component technology to solve is as follows: How should the components be represented and be located? This is the core issue of the component technology. As in the definition of a component, there is no strict representation of component. It can be in all kinds of formats. Then how to represent a component certainly is very important. The representation also directly leads to the methodology of how to locate a desired component in component repository. Some methodologies have been proposed such as information retrieval technology, formal description technology and so on. However most of them focus on components that are implementation level functions. This kind of representation ignores the reality that some components are framework, architecture and even reusable ideas. A component representation framework thus is needed to standardize the way to describe all kinds of components and facilitate the search for desired components.

Assembly technology is another example of implementation level component technology. It is far from being well understood. Currently not many technologies support the idea of component assembly. The most popular one is remote procedure calls plus dynamic link libraries. Conventionally a segment of code in a program only can call another segment of code within the same program. This is called process boundary. In order to let one process utilize the service of another process, or to let processes communicate with each other, a wide variety of inter-process communication (IPC) mechanisms has been adopted by modern operating systems, such as files, pipes, sockets, and shared memory.

Soon after software engineers found out that low level IPC was difficult to program and error prone, a new technology, remote procedure calls, was introduced. RPC hides the fact that calls are made across process boundaries and makes it look like the caller and callee are on the same machine. This made it possible to write a segment of code somewhere and implement it somewhere else. Thus RPC is a very good candidate for gluing separate pieces of code together. Despite the convenience of RPC, people found the procedural nature of it did not work very well for object-oriented approaches. Thus interface definition languages (IDL) were proposed to eliminate the shortcomings of RPC. IDL changes the notion of functions to interfaces. Therefore IDL makes the inter-connection between objects very clear and easy to understand. The support of object and object interface makes it more suitable than RPC in the component world. In fact, in the current component technology, software engineers use IDL to bound components and use objects as components. Although it is not perfect, it is reasonable, since components can have a lot of forms and object is certainly one of them.

The component repository [70] is another core issue for implementation level component technology. It has been a consensus that without a component repository and its searching capacity, the component technology would be of no use. The repository is vital to locate desired and quality components. A detailed study of component repository is presented in the later chapters. Due to the tradition, some researchers use "component library" instead of "component repository". We will use the terms "component repository" and "component library" interchangeably in the rest of this dissertation.

Black box abstraction and white box abstraction is an issue under hot discussion for component technology. Black box and white box abstraction have been widely studied in software engineering. For object-oriented technology, black box abstraction for an object is preferred. Black box abstraction means that only the interface of an object is available for users. The implementation behind the interface is not visible for outsiders. Black box abstraction forces data encapsulation and it does not allow users to know inside details. On the contrary, a white box abstraction makes both the interface and implementation available for observation. The implementation in a white box abstraction could be either modifiable or not modifiable. Taking it one step further, some researchers refer to the white box of which the implementation is available but modification is prohibited as to glass box. Researchers call those that reveal partial implementation gray box in the sense that the implementation is not totally unavailable, nor totally available. In this dissertation, in a black box only the interface is visible, in a white box both interface and full implementation are available and in a gray box both interface and partial implementation are available. We do not further distinguish if the revealed implementation is modifiable or not.

Components can be white box, gray box, or black box. Black box is preferred during object-oriented programming. But in the component technology, it may not be preferred. Remember that components are built to obtain the maximum generality. However in the real world, higher generality means harder to fit in to a specific task. A black box leaves users no chance of studying and understanding the implementation of the component. Users buy the component and use it as it is. If the component is small, it is likely to perform its alleged function without problem. But if a component is small, it reduces little development cost. So customers would rather buy some large scale components that may take a long time to develop by themselves. Chances are that large components usually need to contain some special business logic that is unique to the customers. For example, the transaction systems in all banks may be similar. Most of processes are the same and can be realized in a component. But each bank has some unique functions in its transaction systems. When they buy a component for the transaction system, they are looking for those special functions too. So if a component only implements the common functions and comes with a black box abstraction, then there is no chance for the users to feel satisfied. Large components usually have to realize a large portion of common functions and meanwhile provide a means to accomplish the special functions. The only way to solve this is gray box abstraction. A component bought from providers should implement the common functionality. And enough space should be left for users to modify. The provider has to study the task of the component carefully so that the component can be built in a way that unique functions can be added easily. The reality is the larger a component is, the less generality it has. So to compensate that, gray box is needed and modification may be necessary.

CHAPTER 3

COMPONENT-BASED SOFTWARE DEVELOPMENT

In the previous chapter, we introduced the concept of component technology which has been considered as the hope for curing the so called "software crisis". Component technology has so many unique characteristics that in order to fully take advantage of them a new field needs to be explored in the software engineering territory. People need to think of software development in a whole new way. The result is component-based software engineering.

3.1 Introduction

Software development can be viewed as a process of mapping the requirements of customers into software systems that can run successfully on computers. The user requirements that can be viewed as the abstract representation of the software system describe the "what" aspect of the system and the implementation of the system can be viewed as to describe the "how" aspect of the system [14]. Traditional software engineering has developed a wide variety of software development process models. The following is a list of the most popular models.

Linear Sequential Model

LSM is sometimes called the "classic life cycle" or the "waterfall model". It was proposed by Winston Royce [61]. It usually consists of four steps: requirement analysis,

design, code generation, and testing. This model usually serves as the basis for other process models.

Prototyping Model

Prototyping model [8] is a loop that starts from gathering requirements. After that, a quick prototype design is given to the customers. The quick design focuses on a representation of those aspects of the software system that will be visible to the customers. Customers give their feedback on the prototype and a revised version is given based on the feedback. The process continues until a prototype is finalized. The prototype then serves as a starting point of the system implementation.

Incremental Model

This model combines elements of the linear sequential model with the iterative philosophy of the prototyping model [48]. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. The sequential model is applied repeatedly and the result of each sequential activity serves as the starting point of the next sequential activity.

Spiral Model

The spiral model, originally proposed by Boehm [6], is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. In the spiral model, a software system is developed in a series of incremental releases. Unlike the incremental model, in which each

sequential activity is equally weighted, during the early iterations of the spiral model, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced. It provides the potential for rapid development of incremental versions of the software system.

Each of the above four models has its own strengths and is well recognized by the software engineering community. However all the process models have defined processes that were mainly developed based on the procedural nature of software development. It does not facilitate the idea of reusing existing software components as building blocks to assemble software systems. Nowadays, component building and assembling technology is available via CORBA, DCOM, and EJB. In order to best utilize the strength of component technology, a process model that focuses on facilitating the reuse of existing software component must be given. In the following section, such a process model will be proposed.

3.2 Component Software Development Process

When developing a software system, two issues are very important, domain knowledge and user requirements. A software system works in a certain domain and the corresponding domain knowledge determines primarily how the software system would behave. The domain specifies a playground and restrictions for a software system. Requirements give specifications to the software system as to what functionality users want within that domain. It lays the objectives and restrictions for the software system. Domain knowledge and user requirements work together to define the software system.

In order to achieve the maximum reuse of components, a thorough study is needed toward the domain. This process is called domain analysis. Meanwhile, in order to build the software system from reusable components, requirements also need careful analyzing to make the reuse of components efficient. The result of domain analysis is a set of reusable components and the result of requirement analysis is a set of demanded components. So for component-based software engineering, the development of a software system includes two activities, domain analysis and requirement analysis. These two activities are occurring simultaneously.

It is unusual that when a software system in a domain is developed for the first time, the entire characteristics of the domain are well studied. If a domain is a relatively new one, every time an application is developed, a part of the domain that is covered by the application is studied. The results of the development activity, such as design documents, system architecture, implementation components, can be stored as components and reused for the future applications. The activity of domain analysis is continuously undergoing through applications until the domain is thoroughly understood, whereas the activity of requirement analysis is application wise. Figure 3.1 shows an outline of the development process of component-based software system. The details of the box "system assembly" are shown in Figure 3.2.

3.2.1 Domain Engineering

Domain engineering is a new area for software engineering [60][77]. The problem with current domain engineering research is that there is little awareness or research at the system development level and negligible work has been done on integrating domain

engineering with an overall software system development process. Our approach solves this problem by integrating domain engineering with application engineering together to define a component-based software system development process.

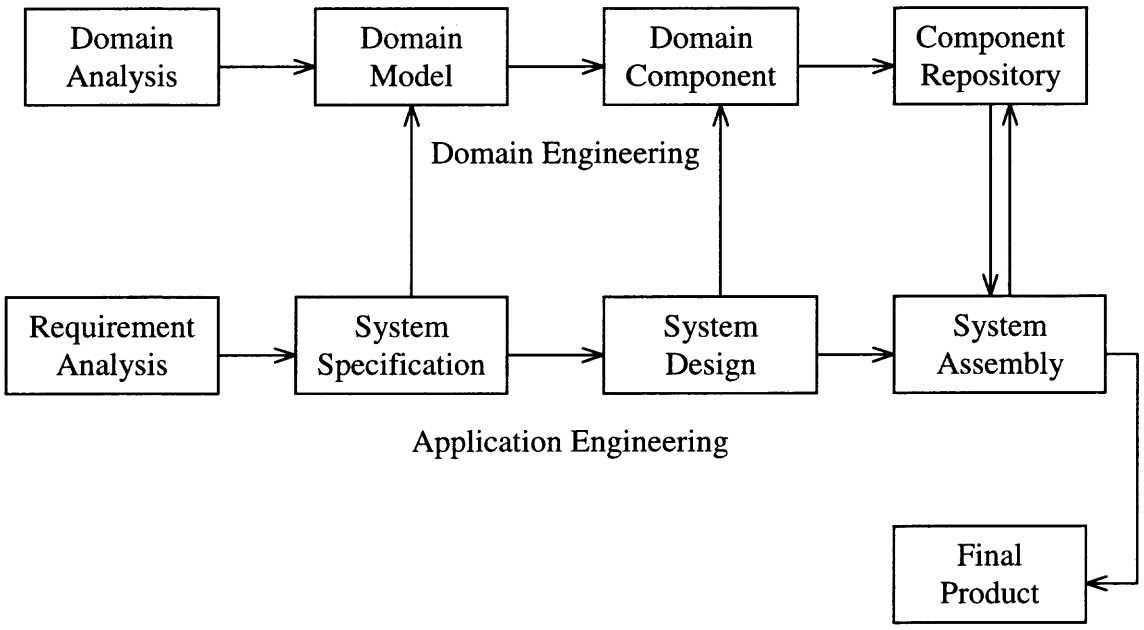

Figure 3.1 Component-based software development process model

Domain engineering is usually used interchangeably with domain analysis. In this dissertation, we distinguish between these two by considering domain analysis as a part of domain engineering. Domain analysis is defined as follows:

> The component-based domain analysis is the identification, analysis, and specification of common request. The result of component-based domain analysis should be a set of reusable components. The components that need to be realized include frameworks, design patterns, design

documents, system architectures, functional components, and interface, etc.

In the definition above, we use the term functional component to denote the implementation level component that performs a function.

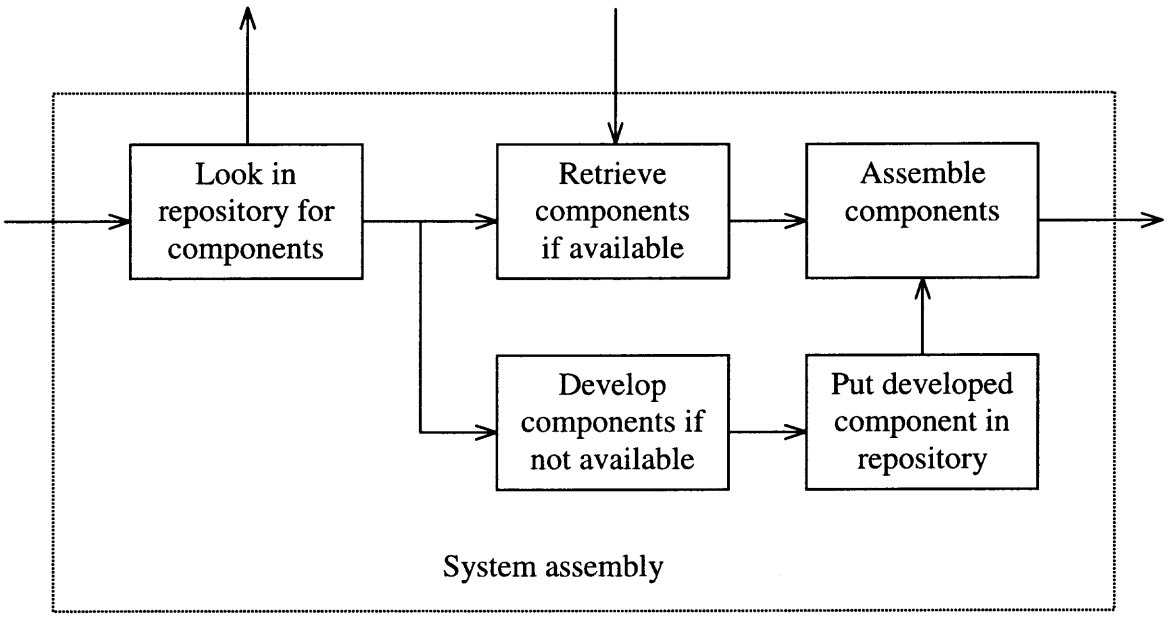

Figure 3.2 Detailed illustration of system assembly

Domain analysis is a complicated, continuous activity. By continuous, we mean that domain analysis does not stop until the specific domain has been fully explored. It is not like requirement analysis that aims at one particular application and only exists during the life span of the system development. The success of domain analysis is based on the completeness of domain knowledge. Domain knowledge can be obtained from the following aspects:

- Documentation
- Existing applications
- Expert knowledge
- Customer specialty

Documentation

Documentation is the direct way to acquire domain knowledge. Writing documentation is a good system development practice under any development environment. Ideally enough documentation should be available when domain analysis starts. Unfortunately, chances are there is not a lot available. Because at the beginning of domain analysis, the domain is not fully studied, therefore documentation is not widely available for that domain.

Existing applications

Existing applications are good examples and study cases during the domain analysis. Documentation sometimes is difficult to understand even if it is well written. Existing applications give a good study case. An application covers a certain part of the application domain. By learning the existing application, a part of the domain characteristics will be explored. Ideally, if we have a wide range of applications that cover the whole domain, the characteristics of the domain will be fully uncovered. However, usually domain knowledge is not fully mastered and each time when we are working on a new application, some new features of the domain will be revealed. That is why domain analysis is a long-term activity and has to be coupled with requirement analysis.

Expert knowledge

For an analyst, if he or she is not very familiar with the domain, then expert expertise becomes important. The truth is that for a large domain it is usually hard to grasp all the domain knowledge. A domain analyst can be fluent in one area of the domain, but it is hard to be in all areas. When some knowledge is beyond the understanding of the domain analysts, experts are needed to present the knowledge. One cannot assume analysts can learn everything by themselves. The expertise from an expert may point out a direct and shortest path to target. Without it, that knowledge may be obtained with much higher costs.

User specialty

Users are the ones who use the applications. They may not have the thorough understanding of the entire knowledge of the domain. However, they are very familiar with the specific applications. They are very clear about what they expect from the applications. As we stated before, each application covers part of the domain. Therefore users usually are experts in the particular area within the domain. Suggestions or feedback from users can be a very good source for domain knowledge.

After the domain knowledge is achieved, component-based domain analysis usually consists of the following steps:

- Abstract units from the domain knowledge
- Identify candidate reusable units
- Verify reusable units

- Generalize reusable units
- Identify reusable units by name and describe units

The steps of domain analysis are straightforward. Among these, the first step to abstract units from domain knowledge is the most difficult and the most crucial one for the success of domain analysis. When a domain analyst is abstracting units, the difficulty is in direct proportion to the volume of the domain knowledge. The most common way for domain analysis is to gather existing applications in the domain and abstract the units based on the applications. This way, it is easy to see clearly what functions this domain usually performs and what the relationship between these functions is. That is the reason why it is usually hard for the first project in the domain to develop. The above steps for domain analysis is continuous and recursive. It gets better and better over the time and finally reaches a point where the domain has been completely analyzed and all the possible reusable units are discovered. After that, any new application can easily map itself into a set of reusable components for development.

Identifying candidates for reusable components is the activity of finding frequently repeated components. For the applications in the same domain, they usually have a common set of functions needed to perform. These are the candidates for reusable implementation level components. They also have a common set of design patterns, system architecture, frameworks, etc. These are the candidates for describing components. Verifying reusable components is the activity to choose those most likely to be reused. Some of the verified reusable component candidates will be implemented by a domain-wise component development team that belongs to the application domain. Some

are left to the future application's develop team. Those being chosen to implement by the domain component development team usually are general, not likely to change across applications, highly repeated, and easy to adapt. After verification of the reusable components, each of them is given a unique name and proper description and passed on for implementation.

Domain models are the result of domain analysis. They can be drawn after the reusable components are discovered. A domain model is one level higher than reusable components. It is the abstraction of relation between components and collection of components. Some models are highly repeated such as front end GUI, back end server, data server and data storage. Each of them is a collection of components that are realized during the domain analysis and relationships among them are also discovered at the same time.

3.2.2 Application Engineering

Application engineering is the activity undertaken parallelly to that of domain engineering. Application engineering takes advantage of the results of domain engineering and also benefits from the completeness of domain engineering. It is easy to understand that domain engineering benefits application engineering since domain engineering provides solid background knowledge to application engineering. As we stated earlier, the domain engineering activity includes collecting existing applications. After a software system is developed, it becomes an existing application. Therefore it gives domain engineers another case study to better understand the domain.

Gathering requirements is the first step of application engineering. It is a part of the requirement analysis [18]. Requirements are gathered by system engineers who contact the users to get the details of what they want from the application. Requirements may contain every detail of the system if the users or the system engineers have a thorough understanding or knowledge toward the application. However, this is not always the case. Users usually only know what their objective is. They often do not have much domain knowledge. Therefore they cannot give a detailed requirement that can be mapped to system specification directly. A typical example is a bank that wants to develop a transaction system. They want the system to have certain functionality without knowing the details of how the system achieves the goal in the particular domain. System engineers have more domain knowledge and are able to transform user requirements into much more detailed requirements for the developers. When help is needed, they resort to domain engineers.

The commonly used technology during requirement analysis is use case analysis. Use cases [29] provide a description of how the system will be used. To create use cases, the analyst should first identify the different types of people or devices that use the system or application. These people or devices are called actors. Actors represent roles that people play, like the system operators. Actors actually could be anything that communicates with the system or application. Note the actors are not the same as users. Actors play only one role whereas users can play more than one role at the same time. For example, a system administrator can be the one who updates the hardware of a system and the one who maintains the system configuration. In this example, there are two actors, one for updating and one for maintenance, and one user, the system

administrator. Usually a system consists of actors of different levels. Primary actors are those working directly and frequently with the system and secondary actors are those usually doing supporting work. Use case analysis, like other analysis activities, requires recursive actions or multiple iterations to finish. Usually the primary actors are identified during the first iteration and secondary actors are identified during the following iterations.

When the use cases are identified, the system functions and process flows can be identified. System specification is achieved based on the use cases. When extracting system specifications, domain models can be referenced to better and faster achieve the specifications. During the system design, components are discovered from the system specification. A lot of technologies can be used to accomplish it. Most OO technologies fit very well in this area.

After the requirements are specified by the system engineers, developers take over the job to develop the desired system. Since the system design is the result of identifying reusable components, most components needed in the system should be available, especially for the applications that are constructed in a well-studied domain. Of course there are still specific components that are unique to this application. For these components, project developers have to build them specifically. Then those specially developed components are added to the component collection for the domain. The key to the success of this stage is the availability of a controlled repository that contains all the existing components for the domain or several domains. We will study the component repository in more detail in later chapters. The technologies to glue the components

together and to construct systems are beyond the scope of this dissertation. A more detailed study can be found in [32].

3.3 Team Roles in Component Software Development

A software system or application is developed by a development team. During the development of a software system, team members need to work on their own part of the application, help each other, and integrate their work into a complete system. In [27][35], detailed studies of team roles have been conducted to help the success of software development. Like traditional software engineering discipline, a good and clear definition of team roles for component-based software development can help achieve the maximum usage of team resources. However, the definition of team roles in [27][35] does not apply very well to component-based software development. New team roles need to be defined and added to accommodate the unique characteristics of component-based software development. In this section, we will define the team roles for component-based software development teams. In the definition, both traditional team roles and new component-based team roles will be presented to form a complete component-based development team.

A component-based software development team can be basically divided into three subteams. The first is the system engineering team, the second is the system development team and the third is the support team. These three teams work together to build a software system. The system engineering team is more like a front-end team that communicates with the end users whereas the development team is the one that works on the back-end and implements the system. The support team gives technical support to

both system engineering team and develop team. In the following subsections, we will define these three subteams and their unique team roles in detail.

3.3.1 System Engineering Team

The system engineering team works with end users to get their requirement and it also needs to communicate with domain engineers to get domain knowledge to help the developers during the development. A good system engineering team should have the following team members.

Project Sponsor

The project sponsor is the one that brings in the project from the end users. He or she is responsible for the project. The project sponsor is also responsible for getting enough resources such as funding and labors to the whole project. Usually the project sponsor works very closely with the end users and is very familiar with the business that the end users are in. The project sponsor takes the business from the end users and delivers the product (software system) to them. He or she should also be holding a certain level position so that he or she can resolve issues and even conflicts.

Domain Expert

The domain expert is in charge of the domain knowledge of the application. In the system engineering team, a domain expert should hold enough knowledge of the domain and a certain degree of the knowledge toward the specific software system. The domain expert works closely with the domain engineers to get the required domain knowledge. When

other team members have questions about the domain knowledge, he or she is the first contact in that matter.

Project Manager

For each project, there is a designated project manager. He or she coordinates the members in the system engineering team. The project manager ensures the smooth development of the software system, and that the product is delivered as agreed between the project sponsor and end users.

System Engineer

System engineers are working on the details of a project. They should hold the domain knowledge for the specific application. They write the detailed requirement for system developers and are responsible for answering questions from developers. They are the contact points between the system engineering team and the developer team. They also should be able to understand implementation details to some extent so that they can correlate business logic with the implementation. This understanding is important for facilitating future maintenance.

3.3.2 Development Team

The development team implements the software system. A component-based software development team has roles that can be found in a traditional software development team. In addition, there are some team roles uniquely belonging to a component-based software development team.

Component System Architect

The component system architect is the most important role in a component-based software development team. A sound software architecture provides an overall structure and a set of rules for managing the scale and complexity that is inherent in enterprise software development. Component-based software development requires a clear definition of system architecture before the implementation starts. A component system architect needs to be very knowledgeable in the specific domain. He is responsible for recognizing frameworks within the component-based software system. He defines the functionality of each framework and the interoperation among them, and ensures smooth integration of the frameworks. This is the most important role in the component-based system development process in the sense that it gives the direction of the development. If a wrong system architecture is proposed, no matter how good other team members are working, a failure of the system is inevitable. In order to facilitate reuse, a component system architect must have good awareness of existing frameworks and keep updated about the latest development in the domain.

Component Framework Architect

Imagine a component system architect as a master planner for a city's development. He divides the city into different functional areas, such as finance area, university area, and sports area. Then the component framework architect can be viewed as the planner for each of the functional areas. A component framework should accept the plug-in of components. It facilitates the interoperation of these components and lays restrictions on their interaction. These restrictions have to be specific and precise enough for

components, otherwise even the smallest inconsistency of component interaction can lead to system failure.

A component framework architect needs a thorough understanding of the existing frameworks in the domain. This ensures the minimum time to be used during the framework design. Modifications to the existing framework and creation of new framework are usually necessary due to the unique requirements of the system. When these are inevitable, a framework architect has to be careful in the following two issues: preservation of compatibility with most existing components and preservation of interoperability with other existing frameworks. A component framework architect also has to specify very precisely what the framework expects from and provides to a component. Only this way, developed components can be sure of quality and in working order when plugged in the framework.

Component Developer

Components are the building blocks of a software system. System architectures and system frameworks are the blueprint of a system and components are functional blocks to build the system according to the blueprint. Component developers are those who transfer the specifications on the blueprint into real components. On this level, component programming language and programming skill is important. Component development is not much different from that of traditional software development. Component developers have to be extremely careful when they analyze the specification. The restrictions laid out by system framework and architect have to be precisely met. Otherwise the inconsistency of component interactions can lead to system failure.

Component Assembler

Components do not perform any functionality when isolated from other components. For a component system, all the developed components need to be assembled to form a complete system. A component assembler takes the frameworks and components and assembles them together according to the requirement. Component assembler is a unique team role in component-based software development process.

The distinction between component developer and component assembler is difficult to maintain sometimes. A component developer may want to adopt ready to use components in his component as building blocks. In this case, he has to be a component assembler. Component assembly can be achieved by programming or some automated assembling tools. Currently, manual assembly such as IDL is the means for most component systems. Automation tools are not widely available. Java's Swing has some built in assembler. For example, a text field component can be dropped into a scroll bar component to form a text field with scroll bar. However this kind of automation tool is very limited and only good for specific application.

3.3.3 Support Team

The support team helps to ensure that the system engineering team and the development team have the necessary technical infrastructure and knowledge base to meet their needs. Most roles in this team are heavily influenced by the unique characteristics of the component technology.

Metrics Expert

A metrics expert is responsible for providing projects with guidance in performance estimation. He collects the metrics such as function points to estimate the effort being put in the system and the result out of the effort. This team role adjusts the resources assigned to each team and tries to maximize the result-effort ratio. It also measures the progress of adopting component technology and identifies the most effective reuse strategies. A complete survey of metrics models can be found in [20].

Reuse Expert

A reuse expert possesses certain knowledge about the existing reusable components. He is responsible for helping customers or developers to find the needed components. This goal is achieved usually by the utilization of a component library. So a reuse expert can also be called component librarian. The term component library is the same as component repository. We will use them interchangeably in the rest of this dissertation. A component library is where the components are stored permanently and it is open for search. A detailed study of the component library will be conducted in the later chapters. Component librarians are responsible for managing component libraries, checking components in and out of the library and using their insight to the library to help customers find the most appropriate component for their needs. Not like regular librarians who do not have to have the expertise or knowledge toward the books in a library, a component librarian usually needs to know the components in the library to some degree to better serve the customers. Also because a component library is not a real library but a

digital virtual library, the librarians need to have some computer knowledge to operate the component library.

Tester

Like traditional software development, testing is crucial for the success and quality of software systems. A component-based system tester needs to know testing strategies like traditional testers do. They are also facing new challenges from component-based technology. In addition to testing the functionality of components, their interfaces also have to be tested to ensure that they meet the requirement and can be reused for further systems. In order to guarantee the successful integration of components, the interactions of components should be tested too.

Legacy Expert

The componentization of legacy system is new to the software engineering community. Legacy systems are those that are old in terms of technology by which they were implemented but still perform crucial operations in today's business. Such systems are usually well developed and functionally stable but hard to integrate with newly developed system when migrating to new technology. Componentization of legacy system is to wrap legacy systems and make them integratable with other components. The methodology of componentizing legacy system will be studied in more detail in later chapters. A legacy expert provides advice on both the suitability of legacy assets for wrapping and the impact of proposed new components. He should not only hold

necessary knowledge of the legacy system but also be exposed to the latest technology to facilitate componentization.

Source Control Manager

Source code control is very important in traditional software development. It holds true for the component-based software development too. During the development or maintenance process, multiple developers could be working on the same component. In order to synchronize their work, any modification or development has to be put under control. A source control manager is responsible for the integrity of components: keeping track of the modification made on components, laying restrictions on the priorities to modify components, and making sure each developer is aware of the currently undergoing modifications on the component.

Table 3.1 shows which roles are traditional and which ones are unique to component-based software development.

3.4 Summary

In this chapter, we have proposed a development model to develop software systems based on reusable components. This model utilizes the unique characteristics of component-based technology. A domain-wide analysis and a system-specific analysis are conducted simultaneously and benefit each other. A definition of team roles for component-based software development is proposed. Some roles have corresponding counterpart in a traditional software development team with additional responsibility

attached to them. Some roles are unique to a component-based software development team, such as component assembler. The clear definition of team roles provides a convenient catalog of different skill sets that apply to different parts of a project. It ensures the skills of individuals to be used to their maximum and the best possible software systems being developed by the team.

Table 3.1 Traditional team roles and component-based team roles

Team Role	Traditional	Component-based
Project Sponsor	×	×
Domain Expert		×
Project Manager	×	×
System Engineer	×	×
Component System Architect		×
Component Framework Architect		×
Component Developer		×
Component Assembler		×
Metrics Experts	×	×
Reuse Expert		×
Tester	×	×
Legacy Expert		×
Source Control Manager	×	×

CHAPTER 4

LEGACY SYSTEM COMPONENTIZATION

Component technology is a recent software development technology for the software engineering community and industry. It has the potential to dramatically reduce the cost and time period of software development. Increasingly, more and more commercial enterprises are trying to employ sophisticated component technology such as OMG's CORBA, Microsoft's DCOM, and Sun's EJB to address key business applications.

Most of these business applications, however, have been implemented for a long time and are based on then advanced but by now obsolete technologies. These systems are called legacy systems [75]. Despite the old, outdated technologies, most of the legacy systems play an important role for their organizations. They are usually the core applications and have been maintained for a long time. Therefore legacy systems are very stable and robust in the sense that they are almost bug free and rarely out of service. It would be ideal if these legacy systems could be substituted by the new systems implemented using component technology. However, the size and complexity of those legacy systems prevents them from being replaced or rewritten in a short period of time and the critical position of those systems does not allow any kind of outage.

Then how can component technology be applied to these legacy systems in a timely manner? The answer is legacy system componentization. By being componentized, the legacy systems still provide the stable services that organization needs, and at the same time they can be treated as components and can be integrated with other components to compose a new software system which fully adopts the new software development technologies. Currently, the research of evolving legacy systems is

conducted at the object-oriented level [45] [57] [71]. It does not facilitate the idea of adopting component technology. However the results in this research area establish a very good foundation for the idea of legacy system componentization. The technologies used to evolve legacy systems are mainly wrapping [16][34][65][78] and extraction [11][28][43]. Most of the researches only adopt one of the two technologies. In this chapter, we will study the evolution of legacy system on component level and propose several hybrid systematic approaches to componentize legacy systems using both wrapping and extraction.

4.1 Legacy System

We have seen so many advantages of the component technology. It may be a major breakthrough in the computer technology history. We see many computer-related businesses going after component and component technology. We see some companies successfully adopting the technology already. But in the real world, the software industry does not only have new technologies. Nowadays software engineers and programmers use object-oriented and component technology to build applications and software systems. But there are also plenty of software systems developed in the 1960s, 70s, and 80s in the old procedural fashion. They are huge systems fully developed far before the emerging of the concept of component-based software engineering and even before the appearance of the object-oriented technology. These systems are called "legacy systems" in the sense, to some extent, that they are part of the heritage of the old technology. Most of them reside on mainframes. Despite the long time existence of these systems, they do not quietly disappear, as people would assume. On the contrary, many of them are still mission critical applications for the organizations which use them on a daily basis.

For those legacy systems, rewriting or redeveloping is obviously not a good choice. Abrupt abandoning of legacy systems and their instant replacement by something "new and better" is normally not an option [70]. Think of a legacy transaction system currently being used by a bank. It is old and it does not adapt easily to the new technology. But it works fine, rarely or never crashes due to any code problem. It is impossible to imagine that the executives would tolerate any problem caused by the use of a newly developed system. This is not saying that the new system must be less robust or of less quality than the old system. This situation occurs because usually the old transaction system is big and it would take too many resources and time to start over a new system. Also it takes a long period to test the new system to assure the quality. Under current business pressure, which is much more furious than it was 10, 20 or 30 years ago when the legacy system was developed, it is a conservative yet maybe the best approach for the bank to use the legacy system and gradually improve it instead of developing a whole new system that adopts the cutting edge technologies to replace the legacy system.

So how to deal with the legacy systems? Let us examine a transaction system again. As a legacy system, it may only be accessed by manual log-on and the operators have to manually interact with the system by sitting in front of a terminal and typing in command. This scenario is definitely not acceptable for today's high demand market. A lot of banks have offered online transaction, like Citibank. Customers can pay their bills online, transfer money between different accounts, and check recent credit card transactions. This kind of services is not possible if it does not adopt new technology such as Internet technology and component technology. Of course it is unrealistic for

banks to develop a totally new system to provide these functions. Again this is because the high business pressure does not allow any kind of service shortage. Besides these services still need to access data sitting on the legacy system. The core operations for the new system remain the same as those handled by the legacy system. The only difference is that the new services need to handle a large volume of requests simultaneously from the Internet. Manual operation does not exist any more. So in order to adopt the component technology to fulfill the market's new demand while still keeping the legacy systems in operation to provide stable and secure services, legacy system componentization is the best approach.

4.2 Componentization

From the component technology point of view, it is possible to componentize a legacy system by either wrapping it into a big component or decomposing it into several small and lower-leveled components each of which performs different functions, i.e. extracting components from a legacy system. The extracted components then can be wrapped, and reused or assembled with other components to build new systems. In fact as stated on the workshop on CBSE [9], component technology has to be able to adopt the old legacy system in its context.

In the following sections, we will discuss the methodologies to componentize legacy systems and more specifically, we will look into the detail of how to componentize and integrate legacy systems with Internet applications. Finally a case study is provided to illustrate the methodologies and detailed techniques presented in this chapter.

4.2.1 Wrapper

The basic strategy to componentize legacy systems is to use wrappers [78]. Due to the importance of the legacy systems, people sometimes are reluctant to modify the legacy systems, or even if they do, the modification will be very limited. The legacy system itself is not able to communicate with other components since it does not use the component technology and usually does not have a published interface for integration or assembly. In order to enable its integration, a published interface has to be introduced by a wrapper that wraps the legacy system and hides their implementation detail. Figure 4.1 illustrates a simple wrapper.

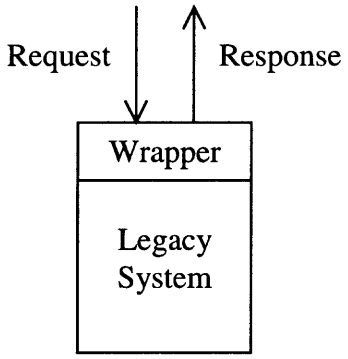

Figure 4.1 Illustration of a simple wrapper

A wrapper has two sets of functionality. One is to provide an interface to the public. Any system that is interested to integrate the legacy system needs to follow the definition of the interface. The other functionality of a wrapper is to translate the incoming message from the published interface to the native format that is understandable to the legacy system and do the same translation on the response from the legacy system. A wrapper can be used to hide a legacy system's dependencies and to

provide the same functionality as the legacy system does to other components. Wrapping technology is widely supported by the industry. We will talk about wrapping technology in more detail later.

4.2.2 Componentization Framework

A legacy system can be reused by other systems by the help of wrappers. But how is the quality of wrapping to be assured? We propose a methodology to describe the steps to wrap a legacy system. This methodology also involves extraction technology to find possible components in the legacy systems. The methodology consists of the following steps:

- Understand the business activity
- Understand the business logic
- Analyze and decompose the business logic
- Analyze the legacy system and map it with the decomposed business logic
- Decompose the legacy system on the implementation level (option)
- Wrap each unit of the decomposed system or wrap the whole system into a component

Understand the business activity

This step is to identify the importance of a business activity. Hundreds or even thousands of legacy systems can be used by an organization. Some of them are big and mission critical, but not all of them. For those big and important legacy systems, wrapping and componentization is the way to keep the functionality while enabling them to adopt new

technologies. For other less critical ones re-engineering or even rewriting may be the best way. By rewriting, the new component will strictly follow the definition of component and will not need wrappers which means less processing time (no overhead for the wrappers) and more flexibility. One should keep in mind that when performing componentization, a real component is always preferred. Wrapping is for the systems that are hard to be rewritten or re-engineered.

Understand the business logic

Business logic is composed of process flows of a business activity. It can be very complicated in terms of the degrees of process flows and the implementation. A good understanding of the business logic assures a solid foundation for further analysis. In this step, people should pay attention to the high-level process flows and do not have to worry about the implementation detail. A flow chart of the processes is helpful at this point.

Analyze and decompose the business logic

After understanding the business logic, a more careful look has to be taken. The process flows are further studied with emphasis on functionality. The activity of decomposing the business logic is taking place in this step. When performing decomposing, it should be kept in mind that generality is highly preferred. The business logic should be divided into units in such a way that the unit has a general functionality and it has a good chance of being reused by other systems. Minimizing the relation between units and maximizing independence of each unit is another point to be addressed. The closer the relation

between two units, the more likely they can be combined into one unit. This step is the key toward the success of wrapping components.

Analyze the legacy system and map it with the decomposed business logic

This is the most difficult step since the implementation needs to be touched and examined carefully. In the above steps, a high level component picture is drawn. However because of the procedural nature of the implementation of the legacy system, the high level decomposition of business logic is not guaranteed to be mapped or reflected in the implementation. This problem can be minimized if the system is developed strictly according to the object-oriented model. Unfortunately that is not the case for most legacy systems that were developed before object-oriented technology was available. So careful assessment should be done on the code level to see if it is possible to map the decomposition of the business logic to the implementation. If the mapping is successful, the implementation is decomposed into different units in the same way as the business logic is divided. If it is not, a compromise should be reached between the business logic and the implementation. The previous steps may have to be revisited to have another decomposition on the business logic that is closer to the implementation. There is no pre-determined method we can use in this step. However the componentization skill, domain knowledge and insight to the implementation are critical to the success.

Decompose the legacy system on the implementation level (option)

Assume we have finished all the above steps and found a good decomposition both on the business logic level and implementation level and they match each other very well. Now

the implementation decomposition steps into action. Some object-oriented technologies [11][53][66] can be used in this step. One can imagine that the approaches of decomposing and the amount of work depend on how and how well the system is implemented. If it follows the object-oriented model, not much work should be necessary. Another possible and more likely scenario is that no compromise is found between the business logic and implementation. The system can not be decomposed and remains monolithic. If that is the case, we do not need to do anything here. Pass the system on to the next step. The goal of this step is to decompose the implementation. So a thorough understanding toward the implementation is critical.

Wrap each unit of the decomposed system into a component or wrap the whole system into a component

The final step of our methodology is to wrap the decomposed units or the whole legacy system if it can not be decomposed. Basically what is needed in this step is to understand how each unit works as a stand-alone component. A wrapper is individually developed dependent on the characteristic of each unit or the unit is totally rewritten to become a real component. Several technologies can be used in this step [39][52]. A more detailed study will be conducted in the next section.

The above componentization framework answers a general question of how to componentize a legacy system. However detailed technologies may still have to be developed in each step of the framework. Hands-on experience plays a very important role most of the time.

4.3 Integration with Internet Application

The ultimate goal of legacy system componentization is to make it integratable in new software systems that adopt component technology. The most compelling force to move software systems to component-based area comes from the business reality. Internet technology has changed today's business models. Conventionally when customers need business service, personal presence to the service agent is necessary. For example, go to a bank to talk to an assistant about a home loan. With the help of Internet technology, most of the service can be obtained from the Internet and without any face-to-face interaction. As a result, development of new enterprise-wide Internet applications has been spurred in all aspects of business. Most of them link to the existing legacy systems. In this case, legacy system componentization could be viewed as preparation for the integration of the system to Internet applications. In the above section, we have presented a general framework of how to componentize legacy systems. In this section, we view the integration of legacy system to Internet applications as a special case for componentizing legacy system in the sense that the objective of componentization is limited to only the integration with Internet application. For this kind of legacy systems, we present a specific methodology that takes advantage of the characteristic of this special case.

4.3.1 Methodology

Time pressure comes as the most powerful influence on the decision of how to componentize legacy systems. The methodology we presented in the preceding section can be viewed as serving for a long term and ultimate goal. The result of applying it to the legacy system is the best possible componentization. Systems being reformed in this

way have achieved their maximum reusability for future applications. However this methodology can take a long time since a thorough understanding of the system is needed and some steps may have to be revisited several times to get the ultimate resolution. No quick work around is allowed in the methodology. This methodology from a long-term point of view is good but may not be suited for those systems that need a quick integration with Internet applications.

In reality, a lot of Internet applications need to integrate legacy systems to provide online services, such as online shopping and online trading. For this kind of legacy systems, we present another methodology that may not be as good as the previous one in terms of the thoroughness of componentization of the legacy system. However it gives a quick solution to answer the demand of today's high-pressure business reality. This methodology consists of the following steps:

- Examine the implementation
- Wrap the system
- Integrate with Internet application

Examine the implementation

For the quick solution, we are not as concerned about the business logic as we were in the previous methodology. The reason is simple: we do not care to decompose the system if we do not need to. As a legacy system, it is usually big and provides a variety of services in one system. We may only need part of the functionality provided by the legacy system for the new system. It is a better solution if we can decompose the legacy system and use the wrapped or reengineered components from the legacy system to build our new

application. In that way, we do not have any unnecessary implementation. However, this decomposition may be time consuming and needs a clear understanding of the whole system which is usually a problem if the system was developed a long time ago and people who contributed have left the organization. The quick solution therefore is to adopt the whole legacy system which provides a super-set of the services needed in the new system. Then wrap the whole system as a component and integrate it with the Internet application, provided we have found the system we need. The implementation needs careful examination. The input and output of the implementation is the most interesting part to us. Depending on how the system gets requests and how it responds, a variety of wrapping technologies is available. Contrary to the previous methodology we are not interested in implementation detail of the business logic. The I/O part of the implementation instead needs to be examined carefully.

Wrap the system

After careful assessment of the implementation, wrapping technology comes into play. Depending on the technology that was adopted to implement the legacy system, different wrapping technology will be used. Some wrapping technology will be presented in the next section.

Integrate with Internet application

The componentized legacy system then is to be integrated with the Internet application. In order to have a smooth integration, the developers need to study the interface of legacy

systems, and different integration strategies are available depending on the wrapping technology. In the following section we will discuss the integration strategy in detail.

The above componentization methodology is considerable simpler than the previous one. It thus provides a quick solution when it is urgently needed. The limitation is obvious, though: a lot of unnecessary implementation is kept in the new system and it does not facilitate any further integration with other applications. So ideally, this methodology is only applied for emergency situations and a thorough componentization of the legacy system is always favorable. A more practical approach is to quickly componentize a legacy system to provide services, and meanwhile conduct a thorough componentization on the system and replace the fast, componentized system at last.

4.3.2 Wrapping Techniques

In order to give an appropriate wrapper, we need to revisit the evolution of software development. The wrapping techniques presented in this section are based on the development technology of the legacy systems. During the first stage, most of the legacy systems were designed and developed on mainframes. At that time, users were typically provided interfaces via 3270 character-based terminals. Operators interacted with computer system via the terminals. This period of software development is referred to as 3270 era. Software systems developed in this period were mainly monolithic.

After that, software engineers used the "client/server" model to separate the functionality of software systems. The client/server model maximizes throughput and improves the economics of delivery because computing power is distributed among

mainframe computers, and workstations. In the client/server model, the client development environment provides end-user interface development, logic, desktop integration, and server subscription tools. This environment yields deployable clients to provide local intelligence and data and access to local data servers. Servers are typically responsible for the primary logic operations, access to enterprise data, enterprise modeling, and reasoning components of applications. The separation of client and server makes it possible for both to only concentrate on its own operation and development environment. This period is referred to as client/server era.

During this period, the main communication mechanisms between client and server are sockets for applications running across machines, and inter-process communication (IPC) mechanisms, like Unix pipes or message queues, for applications running on the same machine. Two kinds of clients exist in this era, "fat clients" and "thin clients". Fat clients are those that have most of the process or application done on the client side. They send a request to the server only when some data have to be fetched from the server or some key process has to be done on server. They are called "fat" because they handle most of the expensive operations themselves. Thin clients are those that only request service from server. The client itself usually only collects the required data, and sends it to the server. Some display the response from server. Most GUI applications fall in this category. In the client/server era client and server are developed on all kinds of platforms. The trouble is that for each server different clients have to be developed depending on the client's working platform.

Now the network technologies have brought us the distributed and heterogeneous computing environment [12]. An application no longer has to be on a single machine. It

can be distributed over different machines. Those machines are working together to provide services over a network. As a part of the network technology, Internet technology is the one people are very familiar with. Users access the Internet from a wide variety of machines. The standard user interface is an Internet browser. Almost all of the applications are running on the server side. Nowadays companies are providing automated service by publishing their service web site. Internet users can access these services by linking to the published service site. To keep their services competitive, companies have to provide online services. This period is referred to as Internet era. These online service applications are defined as Internet applications.

Legacy systems are usually referred to as those developed during the first two periods. Among these two, client/server systems are relatively easier to be wrapped. For 3270 applications, screen scraping is a mature technology for wrapping. Remember a 3270 system uses a 3270 terminal to communicate with the outside world. The idea behind screen scraping is to use a 3270 terminal emulator and fools the legacy system to think it is dealing with a 3270 terminal now. The emulator gets input or request from the user or other applications, translates it into the format of a 3270 terminal and sends it to the legacy system. The legacy system does not know the request is from an emulator. It runs the input and gives the response as it usually does. The emulator gets the response, translates it back and sends it to the calling application or user. Since the 3270 system is very old, in most cases companies that have them have already developed the emulator. The only problem is that usually this emulator appears in the form of a function library. So this kind of emulator does not satisfy the requirement of component interface very well. The wrapper of a legacy system should be in component-oriented fashion, not in

procedural fashion that the function library of emulator always gives. If the 3270 wrapper is in procedural fashion, then either a component based 3270 wrapper or a component-oriented wrapper around the current procedural wrapper should be developed.

Some of the client/server applications are actually based on 3270 systems. The server side is a 3270 system wrapped by a 3270 emulator. We do not treat it as 3270 system since the 3270 characteristic is hidden and transparent to the client. Basically the strategy to wrap a client/server system is to adopt component middleware. No matter what kind of communication mechanism the old system uses, it consists always of 4 parts. A client sends a request, the server receives it, the server sends the response back, and the client receives it. This scenario can be mapped into component middleware, such as CORBA, DCOM, and EJB, which provides exactly the same service. The interface defined on the client and server needs to be published to the other side. Note that here we actually wrap the client/server system into two components, the client component and the server component. Each has it own interface that can be reached through component middleware. This way, the communication between a client and a server is hidden by component middleware. An alternative to this approach is to explicitly use sockets. The interface can remain the same or be slightly changed to adapt to the characteristics of socket communication. Most component middleware that communicates between different machines is indeed based on socket communication. So they should not have a lot of difference. By using middleware, developers do not have to worry about the lower level implementation. However the advantage of sockets over component middleware is that it can be used to communicate through firewalls. Usually middleware can not talk

through firewalls. So if a server has to be reached on the other side of the firewall, using sockets is a solution.

Another way to wrap the client/server application is to use synchronized messaging middleware, such as BEA's Tuxedo [25], IBM's MQSeries [23], and Sun's Java Messaging Service [30]. An asynchronized messaging middleware provides a message pipe between any two computing environments. It ensures the delivery of each message and retains the messages in a persistent store until they are delivered. The communication between client and server then can be realized by the adoption of messaging middleware. The communication flow is as follows. A client puts its request in the messaging queue. A server retrieves the request from the queue then processes it. The result of the server's operation is put back in the queue. The client gets the response from the queue. The advantage is the persistent storage provided by the messaging service. When all servers are busy, the request can be kept in the queue until a server has time to process it. Another advantage is the relative ease to wrap the application. In the previous wrapping strategy, in order to use the component middleware, the input/output of the implementation may have to be changed dramatically to meet the requirement of component-oriented interface. With the messaging middleware, the input/output may only have to be redirected to the storage of the messaging queue. Therefore it provides a relatively easy way to wrap the application.

4.3.3 Integration

Internet applications are the applications that provide services to the Internet users. Some functionality provided by an Internet application already exists. It is just changed to

Internet-based. To provide that existing functionality, the corresponding system can be wrapped and integrated with an Internet application. The interface of Internet applications needs to talk to the wrapped interface of the legacy system. In the rest of this section, we will propose three different ways to integrate componentized legacy systems to Internet applications. Figure 4.2 shows a typical integration of a wrapped legacy system and a wrapped Internet application.

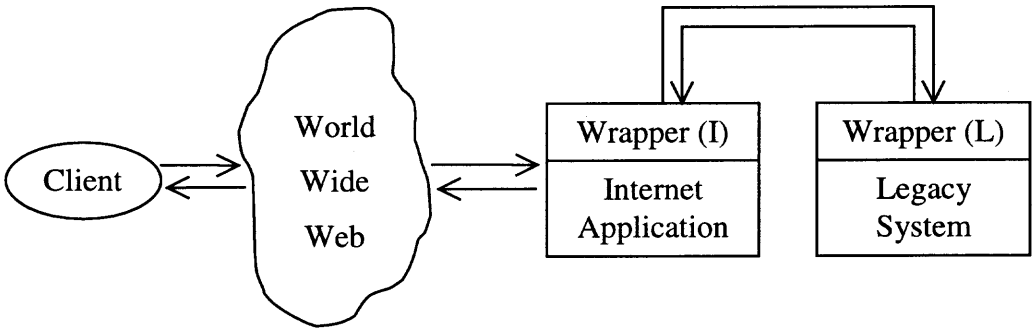

Figure 4.2 Typical integration of legacy systems and Internet applications

Interface (I) denotes the interface for the Internet application and interface (L) denotes the interface for the legacy system. In this configuration, a client sends a request through World Wide Web. The Internet application gets the request and processes it. When service is needed from the legacy system, the Internet application sends the request to the legacy system through its wrapper. At this point, the wrapper of the Internet application talks to the wrapper of the legacy system. The legacy system gets the request, provides the service and gives the response back. Both the Internet application and the legacy system have their own wrapper. These two wrappers establish, perform, and

commit the communication. The wrapper for the Internet application is not mandatory if the Internet application fully adopts component technology. There are two alternatives to this model. The first one is shown in Figure 4.3 and the second in Figure 4.4. Again the wrappers for Internet applications are not mandatory in these two alternatives.

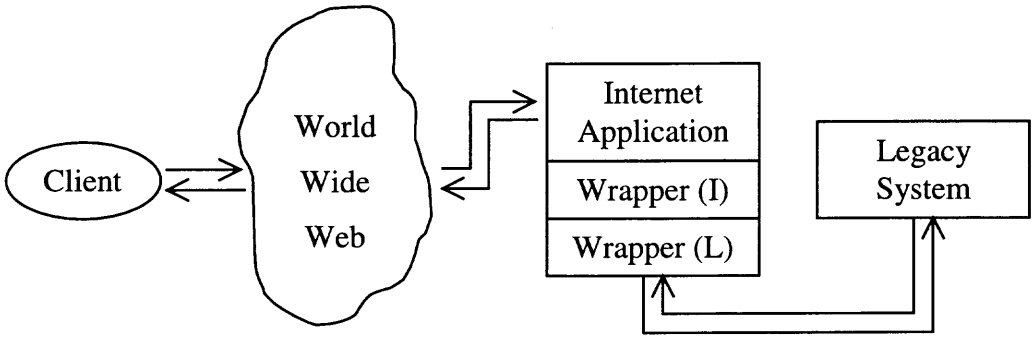

Figure 4.3 Internet application with two wrappers

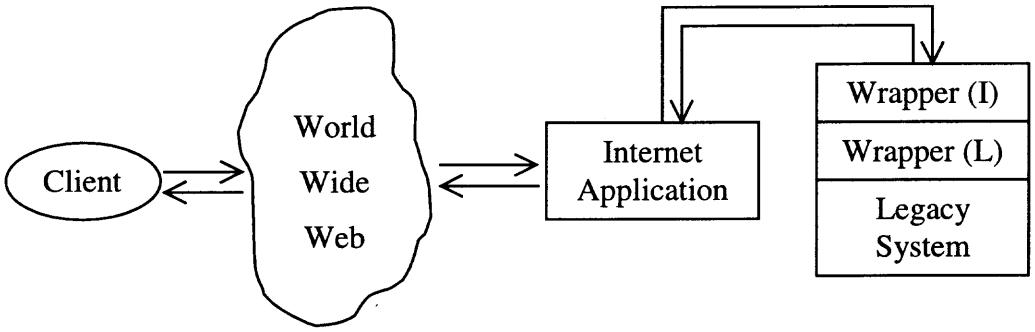

Figure 4.4 Legacy system with two wrappers

In Figure 4.3, the wrappers are both on the Internet application. This is the situation when a quick work is urgently needed. The legacy system may be so complicated that a wrapper of good quality for the system can not be built in time. That

way, we write a specific wrapper for the legacy system. This wrapper only wraps part of the services the legacy system provides and only can be used by the particular Internet application, so it is smaller than a general-purpose wrapper and can be built much faster. This way, we sacrifice generality for the time pressure. This model is also good for the situation when the legacy system may not have many Internet applications to be integrated with. Then a specific wrapper may be a better choice than a general one. However it is obvious that this is a solution far from perfect or even good. Since if the legacy system does not have its own wrapper, each Internet application needs to develop its own wrapper for the legacy system. It may be a shortcut for some applications. But from a long-term point of view, time spent on developing individual wrappers will definitely far exceed that for developing a general wrapper for the legacy system. So this is a model for temporary use.

Figure 4.4 shows both wrappers on top of the legacy system. This can be done by defining the wrapper for the Internet application in such a way that it defines an object for each of the business services provided by the legacy system. This wrapper is more preferable than the separate wrappers in the sense that it provides a higher level and more abstract object-oriented interface. This high level object-oriented feature of the wrapper can be fully utilized by the newest component integration technology like CORBA, EJB, and DCOM. The Internet application does not have to develop its own wrapper for the legacy system's interface. It can directly invoke the services of the legacy systems by calling the corresponding object interface through component middleware, such as CORBA, EJB or DCOM. Internet application developers would definitely like this model. However the demand of resources needed to put into building such a wrapper and

the technical difficulty may be high and deter some organizations. The payoff will be high though, after the wrapper is built.

4.4 Case Study

In this section, we will give an example of how to apply the above framework, methodology and technology. The BPP system is a legacy system in a big telecommunication company T. The BPP system provides the service to connect T1 cables between two offices within the United States. The network of T1s can be considered as a graph. Each office is a node on the graph and each T1 between two offices is an edge between two nodes. Before the BPP system was built, the process of locating T1s were done manually by operators. Sales people got user requests and passed them to the operators. Operators used 3270 terminals to interact with a back end centralized processing system, denoted M to get necessary information. There were several hundreds of screens each of which had a specific purpose and returned a certain set of information. Based on the information the operator got from the terminal, he or she could determine how to construct the T1s between two offices so that it cost the least. Figure 4.5 shows the process flow.

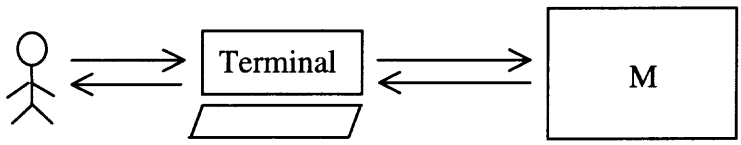

Figure 4.5 Processing with 3270 terminals

The BPP system uses screen scraping to simulate the 3270 terminals. It provides a user interface GUI to the operators. The operator only needs to input the necessary data gathered from users. The BPP system uses screen scraping technology to talk to the back end processing system. Based on the feedback from M and the rules to construct the T1s, BPP can achieve the same result without any user interaction. By doing this, BPP saves a lot of labor cost by automating most of the work. This is the first stage of evolution of the BPP system. Figure 4.6 shows the system configuration.

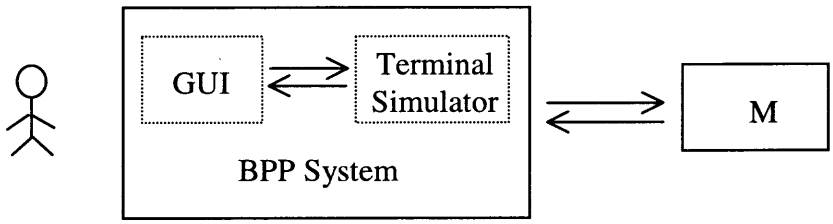

Figure 4.6 Wrapping BPP system by screen scraping

For the second stage, the GUI was separated from the BPP system. This was to make the GUI more portable and make the BPP system more accessible. GUI and the BPP system constitute a client/server architecture. The communication between these two was realized by socket. Figure 4.7 shows the system configuration.

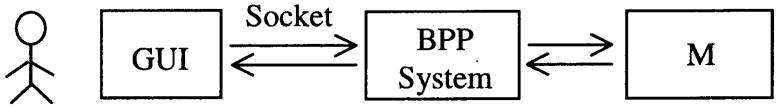

Figure 4.7 Decoupling GUI and BPP system by socket

For the third stage, the GUI needed to become web-based so that users could access the GUI from Internet and local installation and maintenance of the GUI could be avoided. Besides this, potential new subsystems may need the service from BPP system. These new subsystems could be built by component technology and be viewed as Internet application. Due to the time pressure, configuration shown in Figure 4.3 was adopted. Each Internet application built a dedicated legacy system interface in order to talk to the BPP system. Figure 4.8 shows the system configuration.

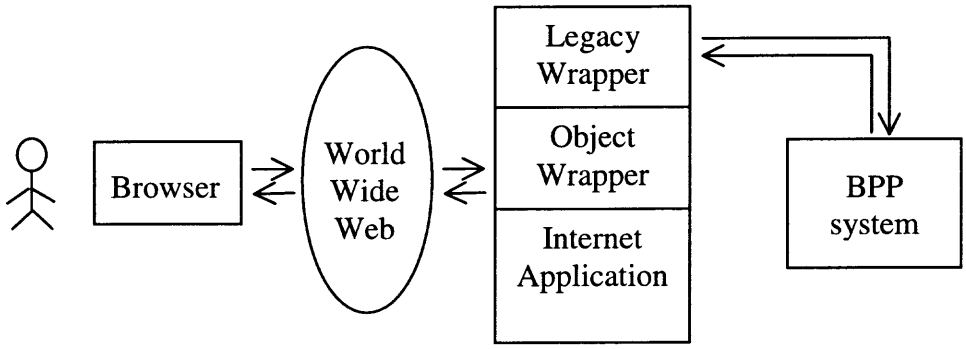

Figure 4.8 BPP system integrated with Internet application

The fourth stage of the evolution of the BPP system is currently undergoing. There are two major objectives for this stage: extracting reusable components from the BPP system and build a component-oriented interface for the BPP system. By decomposing and extracting reusable components, the functionality of the BPP system is easier to be adopted by new systems. By building a component-oriented interface, new Internet applications or other applications do not need to develop specific interfaces for the BPP system anymore. Figure 4.9 shows how the new BPP system will look.

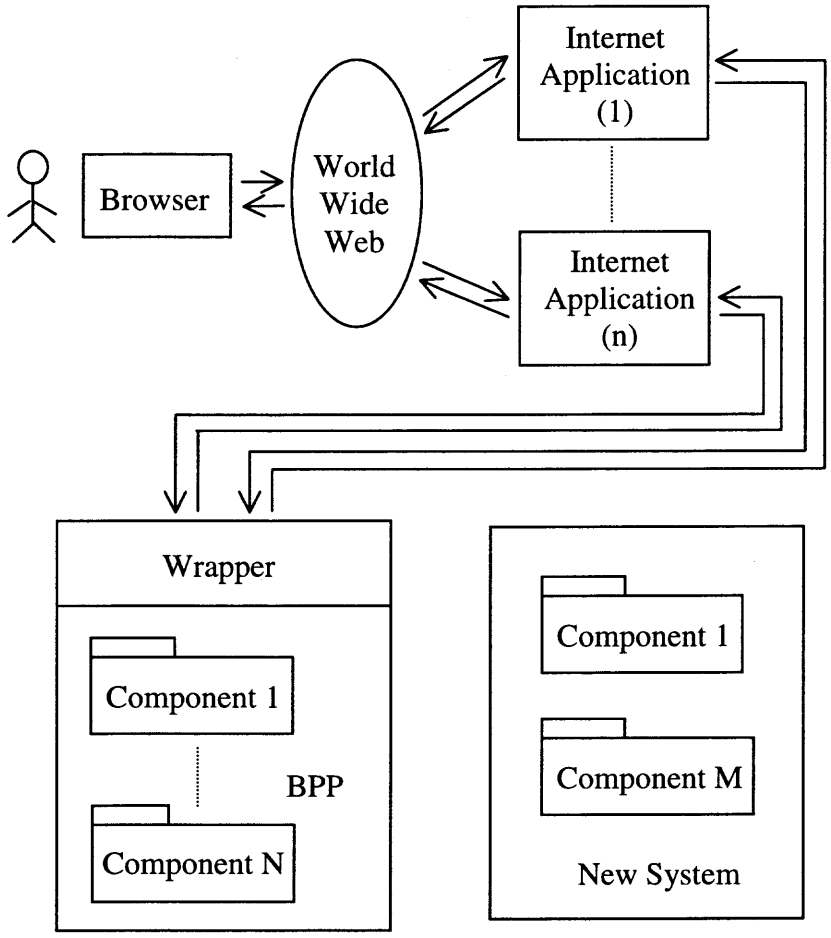

Figure 4.9 The BPP system is decomposed into components 1 to N. These components can be reused by new component-based systems. An object-oriented wrapper is used by all Internet applications.

4.5 Summary

In this chapter, we presented a framework of legacy system componentization. We also presented a specific methodology of componentizing legacy systems and integrating them with Internet applications. Although the latter seems a special case of componentizing legacy system, it is indeed the most prevailing and compelling force to move legacy

systems into component-based systems in today's business. The framework and methodologies proposed in this section are high level systematic guidance for addressing the issue of component building base on existing legacy system. A case study was also provided to illustrate the proposed framework and methodologies.

CHAPTER 5

COMPONENT REPRESENTATION

5.1 Introduction

In "Architectural mismatch: why reuse is so hard" [22], the authors raise a question about why the systematic construction of large-scale software applications from existing parts remains an elusive goal. Their answer is "some of the blame can be rightfully placed on the lack of pieces to build on or the inability to locate the desired pieces when they do exist."

We share the same opinion with them. The definition of component states that components are units of deployment. So one of the reasons for their existence is to allow reusability and integration with other components to produce final products. Imagine we have a mature component-based technology to build software systems by assembling software components, but do not have a component market. What are the consequences? The component-based technology is of no use. We can build systems but we can not find desired components: the building blocks. We end up implementing the components ourselves, which means we are again building our system from scratch not from reusable components. So we strongly believe that the availability of a component market determines whether or not component-based technology will succeed. As a matter of fact, we believe that software component reuse must be supported by an environment that encompasses the following elements:

- a component market that stores, and advertises software components and can be accessed to buy software components

- a specification management system that stores the specification of components and the component reference to the physical location or physical copy of components
- a search tool that locates the component fitting the user's requirement the best
- a retrieving tool that retrieves the desired component from the component market

The above elements imply two open problems for researchers: distribution and retrieval. After a component is developed and fully tested, the manufacturer needs to put it on the market. This activity is distribution. From the mature engineering principles, we know the most important thing in marketing is how to precisely specify the product. These specifications are then organized into catalogs. However, for the software engineering community it is not clear how to specify a software component. Retrieval is the opposite activity of distribution. Suppose we have a software component market already. We have catalogs of components at hand, now the question is how to find the software component that is the most suited for a user's requirements. This is a question of how to match information from a catalog to a user's criteria. Again in this activity, specification of components plays an important role. Let us first examine how CORBA, the currently leading component-based technology, deals with the issue of distribution and retrieval.

In CORBA [54], there are two services, naming and trading. The naming service works in the following way. As soon as a client connects to the ORB, it invokes a standard call to retrieve the object reference for the naming service, because from that it can get references to objects to do anything else that is available on the system. But this retrieval activity is far from sufficient. You can associate a name, or a hierarchy of name,

with an object reference, but you cannot store extra information such as syntax, specification for what the object does, how much it costs to run, etc. A customer needs to know in advance what the objects do and which one he likes before he uses the naming service. In order to store information associated with objects, CORBA provides another service called trader.

A trader is like an electronic combination of a mail order catalog and the yellow pages, where you can look up a service you want, from every provider available. When you find one you like, the trader gives you the object reference. It is like a yellow pages. You can find something you want, but do not have to know the exact information about it. Yellow pages group similar information together. You use common sense or your own criteria to find out which particular item suits you the best. The same thing happens with the trading service. Each object will register various pieces of information about what it does, how and where it does it, how much it costs, where to pick up output, and so on, termed "property list". There is no official property list standardized by ISO. It is reasonable since different trading domains have different properties, and the expertise to draw up the list lives in the domains. However consensus is still needed on the property list within one domain, otherwise programmers won't know what to specify in their query to retrieve the desired component.

The "property list" we believe is dependent on the domain. It holds true for the mature engineering areas. For example the specification for car products is definitely not the same as that for electronic products. Even in the car product industry, the specification for air filter, for instance, is different from that for oil filter. People tend to think software development as one domain, but we believe that with the fast growth of

the usage of software systems in different areas, the software development will be divided into different domains. Different software development strategies will be applied on software systems depending on the domain the software system is in. Possible domains include business applications, scientific applications, operating system applications and so on. The OMG Domain Technology Committee is organizing "domain task forces" to oversee the standardization of domain specification [70]. Currently active task forces are focusing on:

- Business objects: common business object, business object facility
- Manufacturing: high-level requirements, product data management
- Electronic commerce: electronic payment facility, asset and content management
- Telecommunications: control and management of audio/video streams
- Financial: currency, insurance
- Medical: patient identification services, healthcare lexicon service

The future "property list", we believe, relies heavily on the domain of the application and differs from domain to domain. OMG does not specify how the trading service should be implemented. The detail is left to the CORBA vendors. No matter how implemented, the trading services from different vendors should have a similar interface, such as a trading browser to enable user to browse the component repository, to users.

In this chapter, we will study the open problem of component distribution. The core issue behind this problem is how to represent components. We will propose a component representation framework.

5.2 Related Work

Let us go back to the original question: how is a software component to be distributed and retrieved? The core issue behind this question is the component representation schema. According to the component representation schema, components can be abstracted and distributed into persistent storage, and according to it, components can be retrieved from the storage.

As mentioned above, no commonly agreed representation has yet been presented. Ideally, the representation should contain what Tracz [73] has called the 3C model – concept, content, and context. The concept of a software component is "a description of what the component does". This is usually represented by textual description and published interface. The content of a component should be how it is implemented. This would be a highlight of the realization of the software component. Usually casual users do not care about this part, but for software engineers who may be responsible for customizing the component, it should be made accessible. The context places a component within its domain of applicability. This specifies the environment under which the component performs its functionality correctly. To describe a component accurately, concept, content and context have to be translated into a concrete specification schema.

For the past decade, a lot of effort has been put into the representation of components. Methods have proliferated in recent years. Currently there are four major categories of methods to represent components. These are the library science approach [58][59][72], the AI-based approach [47][55][68], the formal method approach [31][50][80], and the hypertext approach [40].

Library science approach

The library science approach adopts methods originally developed for repository information systems. The majority of this kind of methods falls into three categories: enumerated representation, facet representation, and free text indexing.

In enumerative methods, a subject area is broken into mutually exclusive, usually hierarchical, classes. The well-known example is the Dewey Decimal System [15]. Figure 5.1 shows an example of a scheme for describing software tools.

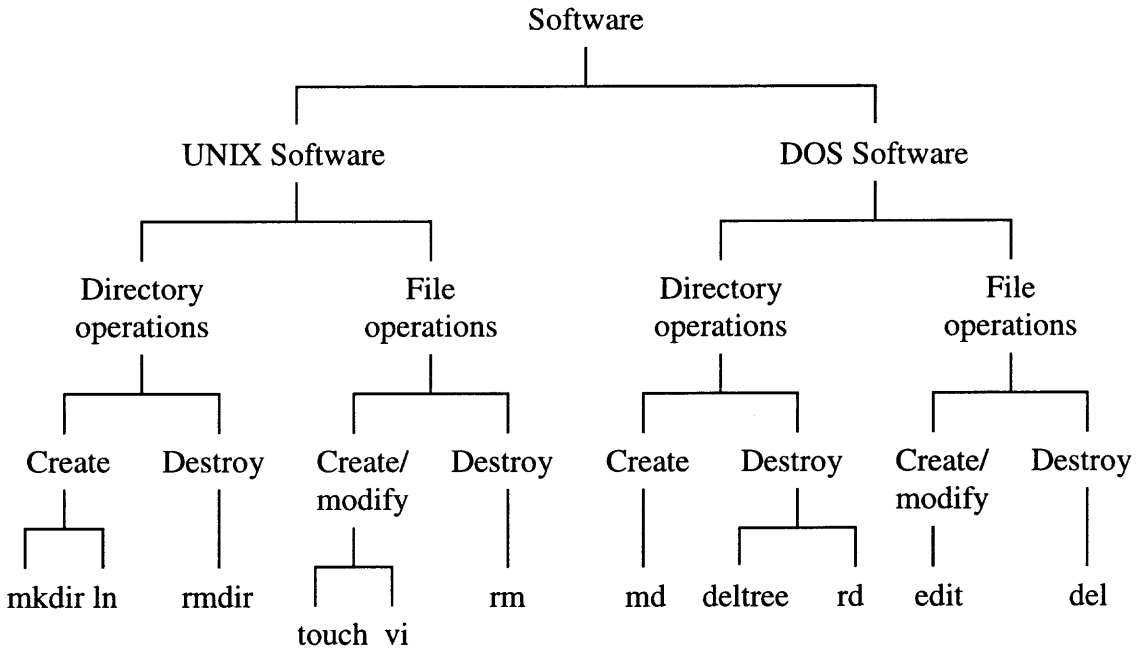

Figure 5.1 Illustration of enumerative method describing software tools

The advantage of the enumerative method is that its highly structured representation makes it easy to understand and use for users. Besides the hierarchical structure helps users to understand the relationship among represented objects. It is also easy for users to browse the structure to find the components they are interested. The

disadvantage is the difficulty to construct the hierarchical structure. Thorough domain knowledge and complete domain analysis is needed to build the structure since it has to be broken into mutually exclusive categories. Even worse, there may be instances that do not fit in only one category, but equally well into several. Another disadvantage of the enumerative method is that it is not easy to maintain. It is hard to add, delete, restructure and store on persistent storage. When a new component is added, if it does not fall into any existing class, then the whole enumerative scheme has to be redefined to accommodate the new component. IBM Share System [58] is one of the first enumerated component repository systems.

In a faceted representation, a subject area is analyzed into basic terms that are organized as facets. A facet list is used to describe a component. Each facet in the list describes one characteristic of the represented component. It may have one or multiple values associated with it. Table 5.1 shows the faceted expression for describing the same software tools in Figure 5.1, plus a new class of software tools that runs on LINUX operating system.

A facet representation does not explicitly state the relationship between components. However, a facet list is relatively easy to maintain by adding, deleting or changing the facets in the facet list and facet values in each facet. As we can see from Table 5.1, the new class of software tools can be added by adding a new facet value "LINUX" to the facet "platform". The facet representation was first studied by Prieto-Diaz when he was at GTE Laboratories [59].

Free text indexing extracts frequently used words to index components. When a description of a component is given, free text indexing tries to find the components that

contain the words appeared in the description with high frequency. In their system [19], Frakes and Nejmeh extracted descriptive terms from comments in C programs to represent components in the C language. The problem with this method is the availability and accuracy of comments in the programs. So the resulting descriptive terms may not characterize the components very well.

Table 5.1 Illustration of faceted method

class	platform	object	operation	name
software	UNIX	directory	create	mkdir
software	UNIX	directory	create	ln
software	UNIX	directory	destroy	rmdir
software	UNIX	file	create/modify	touch
software	UNIX	file	create/modify	vi
software	UNIX	file	destroy	rm
software	DOS	directory	create	md
software	DOS	directory	destroy	deltree
software	DOS	directory	destroy	rd
software	DOS	file	create/modify	edit
software	DOS	file	destroy	del
software	LINUX	directory	create	mkdir
software	LINUX	directory	create	ln
software	LINUX	directory	destroy	rmdir
software	LINUX	file	create/modify	touch
software	LINUX	file	create/modify	vi
software	LINUX	file	destroy	rm

AI-based approach

AI-based approaches [47][55][68] use knowledge representation methods developed in the area of Artificial Intelligence to represent components. The goal is to give semantic meaning to the representation so that it increases the accuracy during the search of components. Among these, the semantic net is the most widely used one. In a semantic net, similar components are grouped together to form a directed graph. The nodes represent components and the edges specify the relationships between components. The advantage of this approach is the retrieval accuracy. The problem with this approach is the difficulty of getting enough knowledge about a domain. The semantic net can also be computationally expensive due to the graph nature of semantic net. Therefore poor response times are one of the major problems with this approach.

Formal method approach

The formal method approach [31][50] [80] is relatively new in this area. Researchers are trying to use formal methods to describe a component. Signature, pre-state and post-state of a function are often used to specify functions. While theoretically sound, the problem with this approach is substantial. It only focuses on the implementation level components. The design level components, such as design patterns, frameworks, can not be represented by formal methods. As we stated in the definition of components, this kind of components is certainly very important to the component-based technology. Another problem with this approach is the difficulty in transforming a retrieval query into a formal specification. Users need to have knowledge of formal methods to be successful in using this approach.

Hypertext approach

The last approach is the hypertext approach [40]. In this approach, a component is described by a set of features. For each feature, there may be a link to other related components. The good side is the ease of browsing the whole storage. The bad side is the difficulty to locate components. If you are looking for a component, you may follow the links to get it. But chances are, unless you are very clear about what you are looking for, you get the wrong links most of the time and it takes substantially longer than you expect to finish the search. This especially holds true when searching in a large amount of components.

5.3 Component Representation Framework

Among the above four component representation categories, the library science approach is the most flexible one and gains most of the research attention these days. We choose the facet representation as the basis for our component representation framework. The reasons are as follows: components are not just program segments in today's component industry. The formal method approach, which is suitable only for implementation level components, then is ruled out; Although AI-based approaches may perform well for a small component repository, the time complexity of them hinders their wide utilization; Difficulties of finding components make the hypertext approach impractical; The success of free indexing depends heavily on the availability of high quality descriptive terms of a component, which is a rare case in the real world. So the free indexing method is out of consideration.

We observe two important requirements of component representation: expandability and easy maintenance. In reality, a component repository could have tens or hundreds of new components added in every day. Outdated components are replaced by newly developed, more full-fledged components. Among the facet representation and enumerative representation, the former works better than the latter with respect to these two characteristics. For the facet representation, expandability and maintenance can be achieved by updating the underlying facet list and terms associated with it. For enumerative representation, a careful study has to be made on the new component and if the new component has functionality crossing the branches of the hierarchical structure, restructuring may be necessary. This operation is expensive and makes enumerative representation less expandable than facet representation. One question with the facet representation, though, is that it seems not to contain enough semantic information. We believe that the facet representation is sufficient for representing components, based on the following observation: in mature engineering disciplines, component information is clearly classified and listed in catalogs. For each type of components, a unique set of information items is used to describe the component. This does not cause any problems in practice. We argue that the future mature component-based software engineering will be going in a similar direction. Like components for other engineering disciplines, software components have common attributes shared by each component and unique attribute to distinguish themselves from others. These attributes make up the facet list for the components.

Based on the above observation, we propose a framework to represent components by adopting double facet lists. One facet list is used to represent the common

attributes and the other facet list to describe the unique attributes. The framework is shown as follows.

Component representation framework

Component

Universal Identifier: string

Local Identifier: string

Originator Organization: string

Domain: string

Type: string

Interface: parameter list

Return Type: string

Exception List: list

Location: string

Description: facet list

This representation framework is a facet list. Each facet represents one common characteristic shared by all components. There are single or multiple values associated with each facet. One facet in the facet list, namely description, is a facet list itself. This facet list has facets that uniquely belong to specific components depending on the domain the component is in. Below is a more detailed explanation of the component representation framework.

Universal Identifier

This facet is used to identify components universally. The value for this facet has to be universally unique, like IP addresses. The scheme to determine universal identifiers should be managed by a centralized organization.

Local Identifier

This facet helps to identify local components. The value is locally unique. It will be assigned by the local organization, typically the developer of the component, following some naming convention. The local identifiers can also be a subset of the global identifiers. If this component will be available externally, a universal identifier is also needed.

Originator Organization

This facet indicates who is the originator of the described component. It has an URL-like format, xxx.yyy.zzz. The last field is the country code, e.g. us, de, jp, cn. The second last field denotes the organization's nature, i.e. a commercial company, an educational institution or a government organization. Values could be edu, org, com. The third last one is the abbreviation of the organization's name, such as att for AT&T, njit for New Jersey Institute of Technology. It is assigned like the way a URL is assigned. Any other field is optional and is controlled by the organization's local authority. Usually it can be used to indicate which group in the organization is responsible for the component. Let us take a look at an example. "Originator organization: bpp.att.com.us" means the component is created by the group named bpp at AT&T that is located in the United States. Note for international companies like AT&T, IBM... the location is not restricted

to one country. So bpp.att.com.cn and bpp.att.com.jp are also possible. It means the bpp group is in the China branch of AT&T and Japan branch, respectively.

Domain

This facet specifies in which application domain the component is located. Possible values could be "finance", "tele-communication", "education", "medical", and so on. This facet is used to distinguish similar components in different domains. For example, brakes can be used on cars and planes. They have the same name and same function, but are not in the same application domain.

Type

Type is used to distinguish a component from other components within the same domain by its functionality. The value could be "design pattern", "framework", "stack", "queue" and so on.

Interface

This facet specifies how to interact with the component. For an implementation level component, this is where the published interface is stored. The parameters passed to the interface are specified here. For a design level component it could be a command used to access the content of the component. Users who know what the interface of a desired component is can use this field to find the component. It is also called signature of a component. Note the term signature is used with different meanings in different contexts. Sometimes it is also a subset of the property list that describes the component sufficiently enough to distinguish it.

Return Type

The return type specifies what the system will get after the component finishes its operation. It is used together with the interface of a component to specify the action of a component. These two facets in the framework are important during the composition of components.

Exception List

The exception list is for implementation level components only. The exception list stores all the possible exceptions a component could encounter during execution. The component may throw an exception to alarm the system when any exception in the exception list occurs. Recall that in the formal methods approach, usually precondition and postcondition must be specified to guarantee the success of an operation. Some researchers claim that precondition and postcondition [50] need to be considered in the representation of a component. We argue that precondition and postcondition are usually only good for describing functional routines. It is useful to ensure the correctness of the execution of a function or method. However a component exists in a much broader context and precondition and postcondition usually is not enough to describe the behavior of a component. Besides, as the complexity of a component grows, the precondition would be very complicated and difficult to be specified in the representation. We believe the component itself should be able to detect any exception to the precondition and throw the exception to alarm the system. So we count more on the exception list to ensure the correctness of execution of a component than on the precondition and postcondition checking.

Location

Location specifies where a component is. When a customer finds a useful component, he or she uses this facet to find the location of the component. Since distributed systems are more and more popular in the computer industry, we believe the component repository does not have to be centralized. It could distribute itself across different locations for better services. The component repository could be a virtually centralized repository composed of several physically separated sub-repositories. In this case, the location of a component needs to be specified so that it can be located within the virtual component repository. An URL like format can be used to uniquely locate the repository and component. In order to distinguish the address of a repository from that of a WWW site, a location-independent naming system is needed. In their paper, Browne *et al.* [10], gives a complete discussion of such a naming system.

Description

This is one of the most important facets in the component representation. It contains a pointer to another facet list in which the characteristics of a component are recorded. The facets in this facet list are determined by the type of the component and the domain the component is in. This facet list is like the property list in the CORBA specification. We believe that when the component technology becomes mature, each component will be classified into one particular class. Components in the same class have certain characteristics to differentiate themselves. In the automobile industry, tires are differentiated by the outer tire diameter, inner tire diameter and the ratio of width to inner diameter. Similar differentiation will develop over time for software components. For

example, queues can be distinguished by size, objects that they operate on, whether or not they are persistent and so on. So for each class of components, a unique facet list can be set up and used to differentiate one component from another. When a user is searching a component, he usually knows the class of the components he is looking for. Like a car mechanic knows what he is looking for in a catalog, a brake pad or an air filter. When the class of the component is determined, the facet of description in the representation is uniquely determined. Then by specifying the specification for this description facet, the appropriate component can be found.

5.4 Summary

In this chapter, we proposed a component representation framework. This framework employs facet lists to represent components. The most important part in this framework is the description facet since it describes the unique characteristic of a component. Most of the retrieval queries compare the desired property of an unknown component against the description facet. In the next chapter, we will see how the representation framework facilitates searching and retrieving activity in component repository.

CHAPTER 6

COMPONENT RETRIEVAL

Software component repositories are persistent storage used to store components and are the key to the reuse of components. As repositories of software components continue to grow the issue of retrieving components from a component repository has become one of the key issues in the component-based technology. The representation of components in a component repository should facilitate the search and retrieval activity. Otherwise, no matter how good the representation method is, no one will use a repository that can not find the requested components. Each of the representation schemes presented in the previous chapter has its own strength and its corresponding method to retrieve components. Based on the proposed software component representation framework, we have developed an efficient method based on neural network technology to retrieve relevant components.

6.1 Neural Associative Memory

Neural network technology is increasingly used in the knowledge representation, reasoning and rule extraction area [38]. A neural network usually consists of processing elements (called neurons) and connections between them with coefficients (weights) bound to the connections, which constitute the neuronal structure, and training and recall algorithms attached to the structure. There are a lot of variants of neural networks for different computing problems. In this dissertation, we apply a variant called neural associative memory [3] [36] for searching components that match or closely match the

user query. Figure 6.1 shows an illustration of a real biological neuron [33] and Figure 6.2 shows the counterpart of an artificial neuron.

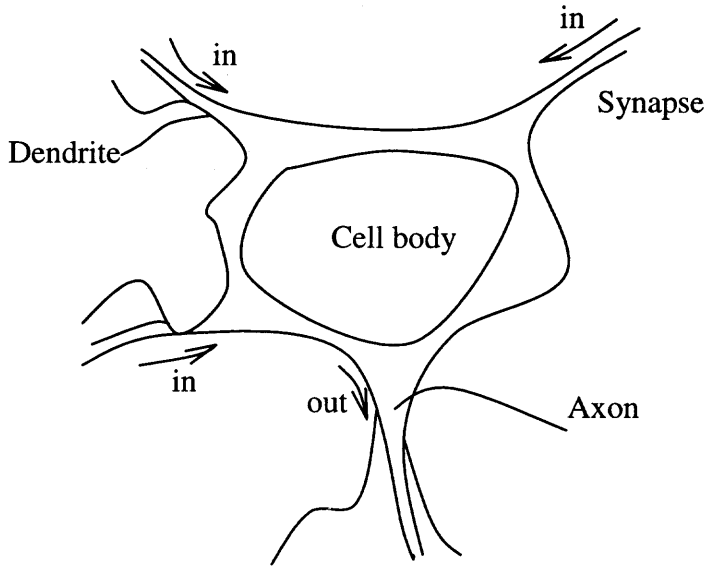

Figure 6.1 A structure of a typical biological neuron. It has multiple inputs (in) and one outputs (out). The connection between neurons is realized in the synapses.

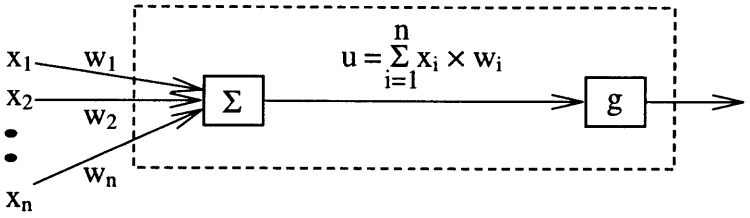

Figure 6.2 A model of an artificial neuron

Neural associative memory [3][36][69] is a single-layer neural network that maps a set of input patterns $X = \{ x^1, \ldots, x^m\}$ into a set of output patterns $Y = \{y^1, \ldots, y^m\}$. Figure 6.3 shows an illustration of a simple neural associative memory. The associative memory remembers a set, $S = \{(x^k, y^k)$: k=1, ..., m$\}$, of mappings. When a new input x is

presented to the network, the corresponding output y is calculated by a mapping $y = xW$. W is called the synaptic connectivity matrix.

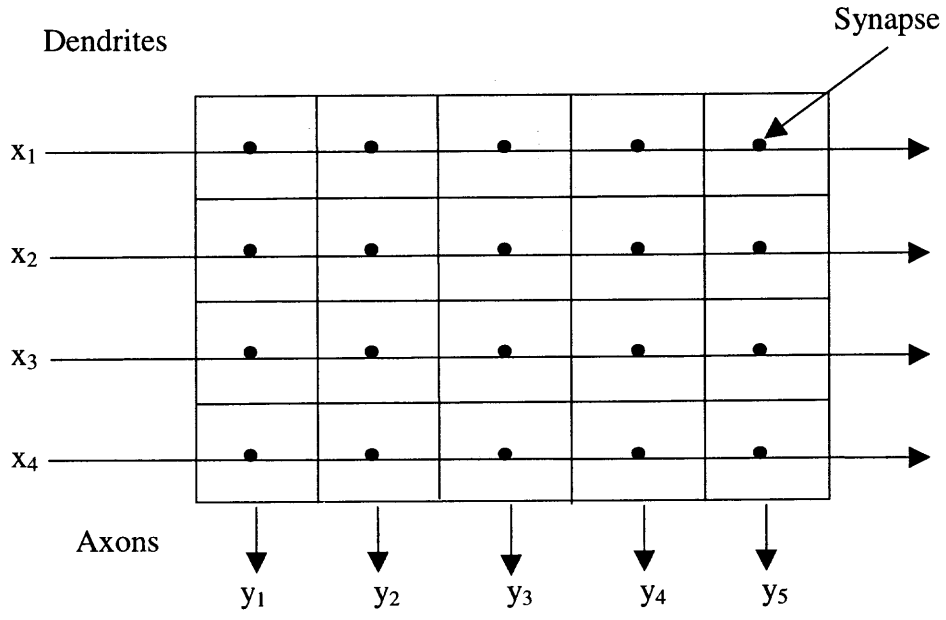

Figure 6.3 A simple neural associative memory

Neural associative memory can be categorized in different ways. One way is feedforward and feedback. In a feedforward network, an input vector x is presented to a single layer of n neurons. The output vector y is calculated in a single processing step. In a feedback model, the output signal is fed back to the input, the network treat the signal as a new input and process it again. The output of a feedback model converges to a stable state that presents the final output of the memory. Another way to categorize neural associative network is hetero-association and auto-association. In hetero-association, the network remembers a mapping from pattern x to y, where x and y are not the same. In auto-association, the network remembers a mapping from one pattern to itself, x to x.

During the learning stage, each pair $(x^k, y^k) \in S$ is presented to the associative memory. This provides a presynaptic and postsynaptic signal at every synapse. According to these two signals, the synaptic weight is changed. We call the method to determine how to change synaptic weight learning rules. Several popular learning rules have been proposed, such as Hebb rule, agreement rule and correlation rule [56]. We use R to denote the learning rules. The synaptic connectivity matrix W with learning rule R is:

$$W = w_{ij} = \sum_k R(x_i^k, y_j^k)$$

For auto-association, it becomes

$$W = w_{ij} = \sum_k R(x_i^k, x_j^k)$$

In the retrieval stage, a new input pattern x is applied to the input of the network. The input signals are propagated through the synaptic connection w_{ij} to all neurons at the same time. Each neuron j transforms the input signals into its dendritic potential d_j, which is the sum of inputs weighted by the corresponding synaptic strength:

$$d_j = \sum_i x_i w_{ij}$$

The new activity of neuron j is determined by a non-linear operation called threshold detection:

$$y_j = f_j(d_j - \theta_j)$$

Function f_j is called activation function and θ_j is called threshold. This equation is used to determine if the neuron j is active or not.

6.2 Retrieval Method

Our goal is to develop a method to facilitate retrieval of desired components in component repositories. Efficiency is the highest priority since we believe that in the future component repositories will grow huge. In an ideal case, when a user query a specific component, he should get it in a very short time. Note that we are concerned about online time that is used to search exactly or approximately matched components. The offline time spent to update and organize the repository is ignored.

In order to be able to find the desired component from the repository, a representation framework has been proposed in the earlier section. For each component, there are ten facets to represent the components. In order not to lose generality we suppose there can be at most n facets to represent a component. Let the facet be denoted $R_1, R_2, \ldots, R_n$. Each facet R_i is a set of finite values, $R_i = \{V_{ij}: j = 1 \text{ to } N_i\}$ where N_i is the number of values for facet R_i. The space for the repository then is:

$$R = R_1 \times R_2 \times R_3 \times \ldots \times R_n, \quad \text{where } R_i = \{V_{ij} : j = 1, \ldots, N_i\}$$

Let φ denote a component and d_φ denote the representation of a component. Then $d_\varphi = (U_1, U_2, U_3, \ldots, U_n)$ where $U_i \subseteq R_i$. d_φ is a facet list. Let L denote the relevancy between two components. Note that the relevancy between two components is the relevancy between two facet lists. So L is also used to denote the relevancy between two facet lists and called facet list relevancy. L is defined as:

$$L(d_\varphi, d_{\varphi'}) = L((U_1, U_2, U_3, \ldots, U_n), (U_1', U_2', U_3', \ldots, U_n'))$$

$$= \sum_{i=1}^{n} L_f(U_i, U_i') / n$$

L_f is called facet relevancy. The facet list relevancy is the sum of all facet relevancies divided by the number of facets. We will quantify the facet relevancy L_f later.

The description facet of a component adopts a facet list to describe a component. This facet list contains several key aspects of the component. We denote this facet list by F and each of facets by F_i, $i = 1, \ldots, n$; n is the number of facets. For each facet F_i, there is a set of values associated with it, $F_i = \{T_{ik} : k = 1, \ldots, M_i\}$ where M_i is the number of values associated with facet f_i. The feature space for F then is:

$$F = F_1 \times F_2 \times F_3 \times \ldots \times F_n$$

The associative memory we use is a one-layer feedforward network. In our model, we use Hebb learning rule to train our associative memory. The Hebb learning rule is stated as follows. If both the neuron on position i of the input pattern X, x_i, and the neuron on position j of output pattern Y, y_j, are active, the weight of the corresponding synapse, w_{ij}, is increased by 1, otherwise the weight remains the same.

$$\Delta w_{ij}(x_i, y_j) = 1, \text{ if } x_i \text{ and } y_j \text{ is active}$$

$$= 0, \text{ otherwise}$$

Figure 6.4 shows the Hebb rule. Then the synaptic connectivity matrix W can be defined as:

$$W(k) = W(k\text{-}1) + \Delta w_{ij}(x_i^k, y_j^k), \quad k > 0$$

$$= 0, \quad k = 0$$

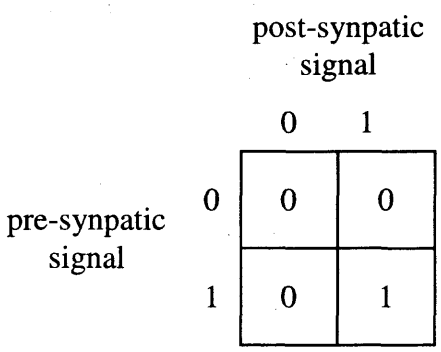

Figure 6.4 Illustration of Hebb rule. The amount of modification depends on the presynaptic and postsynaptic signal.

In the retrieval stage, for each neuron the threshold θ is defined to be 0 and the activation function is defined as $f(x) = x$. The output of each neuron then is :

$$y_j = \sum_i x_i w_{ij}$$

In our method, we assign one neural associative network N_i to one facet R_i in the representation of components. A binary vector is used to represent the feature space of R_i. The dimension of the vector is set to the number of values in R_i. Therefore each bit in the vector represents one value in the feature space. For example, if there are three values for the facet of originator, att.com.us, att.com.cn and ibm.com.us, then the vector to represent the feature space of the facet is $[x_1, x_2, x_3]$. Vector [0,0,1] means the component is produced by ibm.com.us. Vector [1,1,0] means the component is a joint product between att.com.us and att.com.cn.

Another binary vector is used to represent the feature space of components. We set the number of bits in the binary vector the same as the number of components in the component repository. One bit in the vector represents one particular component. For

example if we have 5 components in the repository, we use a binary vector [y_1, y_2, y_3, y_4, y_5] to represent them. Vector [0,0,0,0,1] means the first component is chosen, vector [0,0,1,0,0] means the third one and vector [0,1,1,0,0] means third and fourth components in the component repository are both chosen.

During the training stage, one facet of a component representation is fed into its dedicated neural associative network. For example, in our representation framework, there are 10 facets, so we use 9 neural associative memories to remember the facet values of each component, except for the facet "description". For the description facet, we need more neural associative memories since it itself is a facet list. For each facet, suppose the components have n unique values, then we use an n-dimensional vector to denote the feature space. For one component, suppose the value for the facet occurs on the ith position of the vector, then the bit x_i in the input vector X is set to 1 and others are set to 0. Suppose this component is the jth one in the component repository, then the bit y_j in the output vector Y is set to 1. Based on these two vectors, the increase of the synaptic connectivity matrix Δw_{ij} can be determined. After all the component representations are fed into the associative memory, the synaptic connectivity matrix W is constructed. Note that during the training stage, there can be multiple bits set to 1 in the input vector, if one component has multiple values for one facet (like in the above example, a component is a joint product of two organizations). It is also possible for the output vector to have multiple 1s if multiple components share the same value for one facet. Neural associative memories for different facets can be trained simultaneously to save training time. Let us look at an example of training stage.

Example 1:

Suppose there are 4 components, C_1, C_2, C_3, and C_4. The facet list contains 3 facet values [remove, create, modify]. Component C_1 contains "remove". Component C_2 contains "create, modify". Component C_3 contains "create". Component C_4 contains "modify". The facet values can be represented by a vector X=[remove, create, modify], and the components can be represented by another vector Y=[C_1, C_2, C_3, C_4]. The mapping of facet value vector and component vector for component C_1 is thus [1, 0, 0] -> [1, 0, 0, 0]. After this mapping is fed to the associative memory, the synaptic connectivity matrix is :

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The mappings for C_2, C_3, and C_4 are [0, 1, 1] -> [0, 1, 0, 0], [0, 1, 0] -> [0, 0, 1, 0], and [0, 0, 1] -> [0, 0, 0, 1], respectively. After all the three pairs of mappings are fed to the associative memory, the complete synaptic connectivity matrix is constructed as follows:

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

End of example

During the retrieval stage, the desired component representation is broken down into facets. The values for each facet is fed into its dedicated associative memory to recall

the components that have the same value for this facet. After one processing step, all the components having this value will be recalled. Let us continue the previous example.

Example 2:

Suppose a user is looking for a component that contains facet value "modify". The new input vector thus is [0, 0, 1]. The new output vector can be calculated: $Y=X\times W=[0, 1, 0, 1]$. This output indicates that component C_2 and C_4 contains one of the desired facet values that, in this case, is "modify". If the user is querying for components containing "modify" and "create", the new input then will be [0, 1, 1]. The new output vector will be [0, 2, 1, 1]. This output vector suggests that component C_1 contains 0 desired value, C_2 contains 2, C_3 contains 1, and C_4 contains 1. Notice that all components that contain at least one of the desired facet values are retrieved. The number on each bit in the output vector indicates how many facet values the component, which is represented by the corresponding bit, contains.

End of example

The above example shows if the number of searched facet values is larger than one, the components that contains at least one value in the query are all recalled. So one component may partially match a desired component if it does not contain all but part of the desired facet values. This unique feature of associative memory gives us a way to determine how closely the retrieved components match the query.

We now quantify facet relevancy L_f. Suppose component φ has values U_i for facet i where $U_i \subseteq R_i$, and component φ' has values U_i' for facet i, where $U_i' \subseteq R_i$. The facet relevancy L_f is defined as follows:

$L_f(U_i, U_i') = |U_i \cap U_i'| / |U_i \cup U_i'|$ if U_i and U_i' are not facet lists, where

$|S|$ denotes the number of elements in set S.

For example, a query looked for a component made by a joint team between att.com.us and att.com.cn. Three components were retrieved. One component retrieved was made by att.com.us. The second one was made by the joint team at att.com.us and att.com.cn. The third was made jointly by att.com.us, att.com.cn, and ibm.com.us. Then the facet relevancy on the facet of originator between the query component and the first retrieved component is 1/2, i.e. $|\{att.com.us, att.com.cn\} \cap \{att.com.us\}| / |\{att.com.us, att.com.cn\} \cup \{att.com.us\}| = 1/2$. And the same way, we can derive that the facet relevancy between the query component and the second and the third retrieved components is 2/2 =1, and 2/3, respectively.

The value of $|U_i \cap U_i'|$ and $|U_i \cup U_i'|$ can be derived from the output vector. We showed in example 2 that the value on each bit in the output vector indicates how many facet values that the corresponding component and the query component have in common. Suppose U_i is the facet values in the query and U_i' is the facet values in the component j. V is the output vector which represents all the components. V_j is the jth bit in V and represents component j. When U_i is presented to the associative memory, it is transformed into an input vector. The output vector V is the multiplication of the input

vector and the synaptic connectivity matrix. Then for component j, $|U_i \cap U_i'| = V_j$, and $|U_i \cup U_i'| = |U_i| + |U_i'| - V_j$.

Note that when a facet itself is a facet list, the corresponding facet relevancy is the facet list relevancy between the two underlying facet lists. This applies, in our case, to the description facet in the representation framework. Notice that the "description" facet is actually a facet list that is used to describe the unique characteristics of components.

Now we are able to calculate the relevancy between two components $L(d_\varphi, d_{\varphi'})$. Now given a threshold *m*, $m \in [0,1]$ and a query representation, we can retrieve all the related components whose representation has a relevancy larger than or equal to m to the query representation. *m* is specified by user and is used to indicate how close a user wants the retrieved components to the query. When *m* is equal to 1, the result is an exact match. That means the user gets exactly what he specifies in the query. If *m* is less than 1, the result will be close to the query and is called an approximate match.

Ranking is critical when performing approximate match retrieval. We use relevancy to determine how close the retrieved components are to the query one. When they are returned by an associative memory, the components are ordered by relevancy, components with higher relevancy are displayed earlier.

6.3 Enhancement

The proposed retrieval method based on associative memory is efficient and accurate. However, there are some inherent shortcomings. We identify three issues for the proposed method. The first is the weight between facet values and components, the second is synonym support, and the third is the time complexity of matrix multiplication.

Note that in an associative memory, the weight reflects the relation between a facet value and a component. 1 means a component contains that specific facet value and 0 means it does not. When a user is looking for a component, our method gives the result sorted by relevancy. Relevancy is determined by the number of feature values that are contained in the components, i.e. the sum of weights a component gets. However, the associative memory does not know how unique a facet value is to a component. Let us think of two scenarios.

In the first scenario, there are 100 components, and two facet values. One facet value, V_1, is contained in all 100 components. The other facet value, V_2, is only contained in component A. After the training step, the weight between V_1 and each component is 1 and the weight between V_2 and component A is 1. So to component A, V_1 and V_2 are of the same importance since they have the same weight, 1, to A. However from common sense, we know V_2 is certainly more unique to A than V_1. It better describes the characteristic of A than V_1, since it is only contained in A while V_1 is shared by 100 components. In the second scenario, imagine we have two components A and B. A contains 5 facet values V_1, V_2, ..., V_5 and B contains only one facet value V_1. When the user is looking for a component that contains V_1, B is certainly a better candidate than A, because V_1 is more unique in B than it is in A.

These two scenarios address one question: how to adjust the weight so that it reflects the uniqueness of a facet value to a component. Two factors need to be taken into account: the frequency a facet value appears in the whole component repository and the frequency a facet value appears in a component. We will study this issue in Section 6.3.1.

Another issue to be addressed is synonym support. In reality, it is hard for a user to know which facet value is used to describe components. He can guess a facet value which he thinks is the most likely to describe the component he is looking for. If he chooses a wrong value, he can not get the component even the component is indeed in the repository yet described by another facet value. A thesaurus is a solution to this issue. We will propose a thesaurus model with Bayesian inference and dynamic user feedback adjustment in Section 6.3.2.

The last issue of the proposed method is the time complexity of the matrix multiplication operation. Remember a synapse connectivity matrix is a matrix that could be big. It easily grows to the size of the magnitude of $10 \times 10{,}000$. Multiplication on this size of matrixes usually is time-consuming. However with the help of parallel algorithms and matrix operation algorithms, this issue can be solved. We will discuss it in Section 6.3.3.

6.3.1 Weight Adjustment in Associative Memory

As discussed in the previous section, the binary weight schema, i.e. the weight between components and facet values is either 1 or 0, is not sufficient for accurate component retrieval. The binary weight distribution is based on the false conjecture that all facet values are equally related to a component. In reality, however, some facet values may be more unique to a component than others as we illustrated in the two scenarios in the previous section. In order to reflect that fact, we need to vary the weights in the way that facet values that are more unique to a component have heavier weights.

Our new weight assignment method is based on two observations. First, there is a tendency for less frequent facet values to be more precise to describe components. Second, the more facet values to describe a component, the less weight each facet value should be assigned. These two observations lead to our facet value weighting function:

$$w_{ij}' = \frac{w_{ij} \log^2 (\frac{N}{n_i})}{\sum_{z=1}^{F} w_{zj} \log^2 (\frac{N}{n_z})}$$

This is a variant of classic weight function proposed in [62]. The parameters to the function are defined in Table 6.1.

Note that the sum of weights of all facet values in a component is 1 for normalizing the importance of the facet values. Let us reuse the facet values and components in example 1 to construct the new synaptic connectivity matrix. Following the weight function, the new matrix is constructed as follows:

$$w = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/2 & 1 & 0 \\ 0 & 1/2 & 0 & 1 \end{bmatrix}$$

Table 6.1 Weight function parameters

Symbol	Definition
w_{ij}'	Adjusted weight between facet value i and component j
w_{ij}	Weight in original synaptic connectivity matrix
N	Number of components in the repository
n_i	Number of components exhibiting facet value i
F	Number of distinct facet values

Heuristically, this function takes into account the number of components that exhibit a particular facet value and the number of facet values that is contained by a component. The more components exhibit a facet value *i*, the smaller $\log(N/n_i)$ is. Thus the lighter the weight of the facet value is to the component. Meanwhile the more facet values a component contains, the more items is in the denominator. Therefore the larger the denominator is and the lighter the weight is to the component. This way, the weight function assigns heavier weight to those facet values that better describe the components. As a result when the components are retrieved, their ranking is more accurate. We will test the weight function in our experiment section.

6.3.2 Dynamic Thesaurus with Bayesian Inference

In the real world, a user might specify a facet value that is not memorized by the associative memory. For example, a user may be looking for components that contain "delete" as its facet values. However, delete is not in the associative memory. Instead, "remove" is memorized by the associative memory and the components that contain it are those that the user is interested in. Using the original proposed associative memory, a user is not able to retrieve or has a less chance to retrieve the appropriate component without any synonym support.

A Bellcore study of people choosing terms to describe common objects, cooking recipes, editor commands, and other items revealed that the probability of two people choosing the same word for those objects is between 10% and 20% [21]. Using 15 aliases or synonyms will achieve 60-80% agreement, and 30 aliases can get up to 90% agreement.

Learned from the Bellcore study, a thesaurus can be a help to the shortcoming of our proposed associative memory. A variety of thesaurus models are proposed in different domains [42][44]. A thesaurus in our system memorizes synonyms and when a facet value is not found in the associative memory, it gives the facet values semantically close to the queried facet value. With the help of thesauri, our associative memory can improve its performance by giving more possible matched components.

One problem with a traditional thesaurus is that the weights between synonyms are usually fixed and are set by the experts. In our repository, we do not want to have fixed weights between synonyms. We want to construct our thesaurus in such a way that it discovers and uses the interests of users to adjust the weights between synonyms. So the weights of synonyms are more dynamic and the results of component retrieval are more beneficial to the later users. In this section, we will present a thesaurus model that uses Bayesian inference to adjust the weights between synonyms by taking into account the previous component retrieval results and the users' feedback.

The thesaurus in our system is organized as follows. The thesaurus contains two layers of facet values. The first layer is called primary layer, denoted P, which contains the facet values that are memorized by the associative memory. The facet values in this layer are called primary facet values. The second layer is called secondary layer, denoted S, which contains the synonyms of the facet values of the first layer. These facet values are called secondary facet values. The facet values in the thesaurus are initialized as follows. When a component is put into the repository, the thesaurus updates itself to contain the new primary facet values. Meanwhile, the developer or whoever is responsible of the components can specify the synonyms to the primary facet values and

put them in the secondary layer A traditional thesaurus can also be referenced when constructing the relation between primary layer facet values and secondary layer facet values. Figure 6.5 shows the architecture of the thesaurus. Note some facet values can be contained in both primary layer and secondary layer. This is caused by the fact that people use words interchangeably. Let us look at the following example of the architecture of the thesaurus.

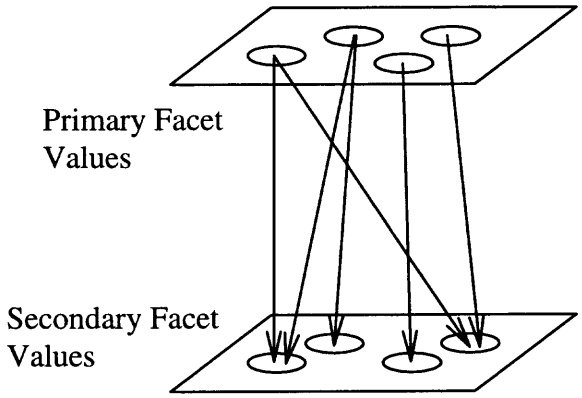

Figure 6.5 Thesaurus architecture. Ellipses are facet values.

Example 3:

Suppose we have three components C_1, C_2, and C_3. Each of them contains "delete", "remove", and "exit". The relation among these three primary facet values are specified by their component developers as follows:

"delete" → {"remove, quit"}

"remove" → {"delete", "exit", "quit"}

"exit" → {"quit", "remove"}

The mapping between primary and secondary facet values thus is constructed as shown in Figure 6.6.

End of example

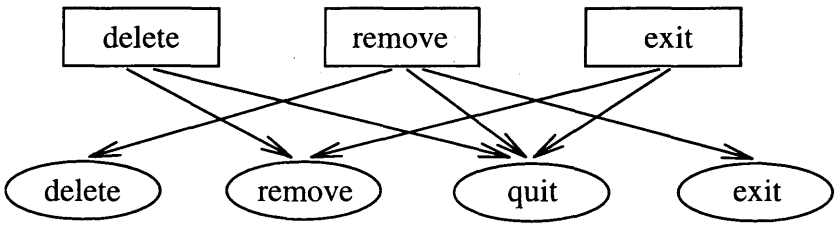

Figure 6.6 Mapping between primary and secondary facet values

Table 6.2 shows the symbols that will be used in this section. We will explain the symbols in more detail when we use them later on.

There are two activities in the thesaurus. When a facet value queried by a user is found in an associative memory, i.e. a hit in the primary layer, all components that contain the facet value will be retrieved from the associative memory. Besides those, the secondary facet values in the secondary layer that are pointed to by the primary facet value will be also presented to the associative memory to retrieve components. This procedure is defined as "forward propagation".

When a facet value in a query is not found in the primary layer but found in the secondary layer, a backward procedure defined as "backward propagation" takes place. All the primary facet values that points to the found secondary facet value will be presented to the associative memory to retrieve components. This is a reverse action of the forward propagation.

Table 6.2 Symbols of variables

Symbol	Definition
P	Primary layer
S	Secondary layer
P_i	Facet value *i* in primary layer
S_j	Facet value *j* in secondary layer
N_{ij}	Number of secondary facet value *j* referred by primary facet value *i*
ΔN_{ij}^{k}	Increase of the number of secondary facet value *j* referred by primary facet value *i* for retrieval process *k*
FR_{ij}	Relevancy between primary facet value *i* and secondary facet value *j* in forward propagations
BR_{ij}	Relevancy between primary facet value *i* and secondary facet value *j* in backward propagations
T_{req}	Total number of queries on the primary facet values
$T_{req}(i)$	Total number of queries on the primary facet value P_i

In the original associative memory, a query input is composed as a binary vector, i.e. the values in the vector is either 0 or 1 dependent on if the query facet value is found in the associative or not. When a thesaurus is in use, the facet values that are achieved by either forward propagation or backward propagation will set values on corresponding bits in the input vector. These values will be between 0 and 1 and be the same as FR_{ij} for forward propagation or BR_{ij} for backward propagation. Now let us look at the variables defined in Table 6.2 and how to use them in our thesaurus.

N_{ij} is the number of secondary facet value S_j being referred by primary facet value P_i. In the thesaurus, these two facet values are connected by a directed edge from P_i to S_j.

If five users are looking for components that contains P_i, three of them finally decide that components containing S_j are also interesting and two of them find component containing S_m interesting. Then we say ΔN_{ij}^{k} is 3 and ΔN_{im}^{k} is 2 for retrieval process k. N_{ij} is defined as the sum of ΔN_{ij} over all retrieval processes:

$$N_{ij} = \sum_k \Delta N_{ij}^{k}$$

We can quantify FR_{ij}, relevancy between P_i and S_j in a forward propagation, based on N_{ij}. FR_{ij} is defined as follows:

$$FR_{ij} = \frac{N_{ij}}{\sum_z N_{iz}}$$

Note that N_{ij} changes with the retrieval processes. Heuristically, the more retrieval processes have been conducted, the more users have put their opinions to the relation between P_i and S_j and the more accurate FR_{ij} is. Here is an example of calculating FR_{ij} and composing input vector.

Example 4:

Suppose P_1 is "delete" and corresponding S_1, S_2, and S_3 are "remove", "quit", and "exit". Three users conducted retrieval process by using delete as their query facet value. The first user thought "remove" and "quit" were also interesting. For him, ΔN_{11}^{1} is 1 and ΔN_{12}^{1} is 1. The second one thought "remove" and "exit" are interesting, so ΔN_{11}^{2} is 1 and ΔN_{13}^{2} is 1. For the third one, who thinks "remove" is interesting, ΔN_{11}^{3} is 1. Then FR_{11}=3/5, FR_{12}=1/5, and FR_{13}=1/5 after the thesaurus is used by three users.

When the forth user is looking for components containing "delete", then the value for facet value "delete" in the input vector is 1. If "remove", "quit", and "exit" are also in the primary layer, then the values for them are 3/5, 1/5, and 1/5, respectively, in the input vector. The result of the retrieval activity of the forth user will then again be counted to calculate the FR_{ij}.

End of example

The above example is an illustration of forward propagation. It is less complex than backward propagation since we only need to find the synonyms pointed to by the primary facet values. The relevancy is well established by previous processes. If the query facet value S_j is not found in the primary layer P, but found in the secondary layer S, and it could be pointed by multiple primary facet values $\{P_i | P_i \subset P\}$, what is the BR_{ij} between P_i and S_j? We will use Bayesian inference to calculate it.

Bayesian inference is based on Bayes' theorem which was discovered by Thomas Bayes in 1763 in his unpublished work "An Essay Towards Solving a Problem in the Doctrine of Chances" [41]. The basic idea of Bayes' theorem is to predict events under conditions of uncertainty. The equation of Bayes' theorem takes into account knowledge of past events and new observations to infer the probable occurrence of an event. Mathematically Bayes' theorem is presented as follows:

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A \mid B)P(B) + P(A \mid \overline{B})P(\overline{B})}$$

P(B|A) means the possibility of A given that B has occurred. P(B|A) is called posterior probability in this equation. It represents the result of the inference. P(A|B) and

$P(A|\overline{B})$ are called likelihood, which is the likelihood that A happens given that B and $\overline{B}$ happens. P(B) and $P(\overline{B})$ are called prior probability which is the prior knowledge we observe from past events. By expanding the denominator of the equation, we can derive the following equation:

$$P(B_i \mid A) = \frac{P(A \mid B_i)P(B_i)}{\sum_z P(A \mid B_z)P(B_z)}, \text{where } \sum_z P(B_z) = 1$$

In our thesaurus, A is the event that secondary facet value is of a user's interest and B is the event that primary facet value is of a user's interest. So P(B|A) is interpreted as if a secondary facet value S_j is presented to the thesaurus by a user, what is the probability that a primary facet value P_i will be interesting to the user. The Bayes' theorem can be rewritten as follows:

$$P(P_i \mid S_j) = \frac{P(S_j \mid P_i)P(P_i)}{\sum_z P(S_j \mid P_z)P(P_z)}, \text{where } P = \bigcup_z P_z$$

P_z denotes the primary facet values in the thesaurus's primary layer. P is the primary facet layer that contains all the primary facet values. $P(P_z)$ denotes the possibility that P_z is queried by a user. $P(P_z)$ is the prior knowledge the thesaurus knows about the primary facet values. $P(P_z)$ is defined as follows:

$$P(P_z) = \frac{T_{req}(z)}{T_{req}}$$

$T_{req}(z)$ is the total number of user requests on primary facet value P_z and T_{req} is the total number of requests performed on the thesaurus. $P(S_j|P_z)$ is defined as $P(S_j|P_z) = FR_{zj}$.

Plug in those items in the formula, and $P(P_i|S_j)$ can be calculated. The relevancy between P_i and S_j in a backward propagation, BR_{ij}, then is defined as $BR_{ij}=P(P_i|S_j)$.

Example 5:

Suppose there are 2 primary facet values, "delete" and "remove". Both facet values point to second facet values, "quit" and "exit". So P_1 is "delete", P_2 is "remove", S_1 is "quit", and S_2 is "exit". Suppose there were 3 users using the repository. User number 1 looked up facet value P_1 "delete" and determined he also interested in facet value S_1 "quit". $\Delta N_{11}^{1} = 1$. The second user looked up facet value P_1 "delete" and was also interested in both S_1 "quit" and S_2 "exit". So $\Delta N_{11}^{2} = 1$. $\Delta N_{12}^{2} = 1$. The third user looked up P_2 and was interested in S_1, $\Delta N_{21}^{3} = 1$. Then $FR_{11} = \sum_k \Delta N_{11}^{k} / \sum_k \sum_z \Delta N_{1z}^{k} = 2/3$, $FR_{12} = 1/3$, $FR_{21} = 1$, and $FR_{22} = 0$. If the forth user is looking up facet value "quit" and "exit". Since both facet values are not in the primary layer, the thesaurus needs to calculate $P(P_i|S_j)$. From the assumption, we know T_{req} is 3. $T_{req}(1)$ is 2 and $T_{req}(2)$ is 1. So $BR_{11}=P(P_1|S_1) = 4/7$, $BR_{12}=P(P_1|S_2) = 1$. $BR_{21}=P(P_2|S_1) = 3/7$, and $BR_{22}=P(P_2|S_2) = 0$. Note when backward propagation takes place, the values for T_{req}, $T_{req}(i)$, and FR_{ij} do not change after the retrieval process.

End of example

The power of this thesaurus model is that it takes users' feedback into account. Note that N_{ij}, FR_{ij}, BR_{ij}, T_{req}, and $T_{req}(i)$ change each time after the user finishes one process of component retrieval. We believe each user is an expert in terms of judging how close synonyms relate to each other, and users tend to have a consensus on the

relation between synonyms. So the more users use the thesaurus, the closer the thesaurus is close to the consensus.

6.3.3 Sparse Matrix Multiplication

The associative memory we use is a powerful tool in terms of memorizing facet values. However, as a reader might notice, the calculation on the associative memory is either vector multiplying vector or vector multiplying matrix. The synaptic connectivity matrix is a matrix that could have a large size. As we all know, matrix multiplication, especially with large matrices, is a relative expensive operation. In order to improve the efficiency, we need to improve our original associative memory.

Fortunately, the synaptic connectivity matrix is a sparse matrix. There are some research results on sparse matrices that we can take advantage of to lower the time complexity on matrix multiplication. So the computational complexity of our matrix multiplication is proportional to the sparse matrix multiplication algorithms that are used to implement the associative memory. The detail of these algorithms is beyond the scope of our paper. We will utilize the result of the algorithms to illustrate the feasibility of our associative memory.

We identify two issues that, we think, are important to the performance of a sparse matrix multiplication algorithm. First the size of the matrix the algorithms can cope with. This is important because the size for synaptic connectivity matrix could be as big as at the magnitude of $10 \times 100{,}000$. So we are looking for algorithms that are feasible at least for that size. Secondly, we believe the scalability of the algorithms to

parallel processing is very important. A fast response can be achieved by using parallel machines to do matrix multiplication.

The following research papers give us some idea what algorithms can be used. Papers [1][76] proposed efficient algorithms and both of them work well for matrices of size of 28924×28924. Paper [24] presented an algorithm to utilize the power of parallel processing. Besides the work on algorithms, faster hardware and bigger computer memories also make a big contribution to the improvement of the performance of sparse matrix multiplication. In our prototype, we will still use regular matrix multiplication since there are not many currently available components that can be store in a repository. Actually there are very few publicly available components at the time this paper is being prepared. Traditional matrix multiplication is not a problem so far for the prototype. When more and more components are available, sparse matrix algorithms then can be adopted to implement a component retrieval system.

6.4 Experiment

We have developed a prototype of a component repository retrieval system based on our proposed methods. We need to point out that this system is intended as a rapid prototype aimed at testing our proposed approach and not as a complete classification and retrieval system. It is not aimed at testing the system usability, nor the user friendliness of the interface of a potential prototype based on our method. So the system is relatively simple and not very user friendly. The general architecture of our prototype is shown in Figure 6.7.

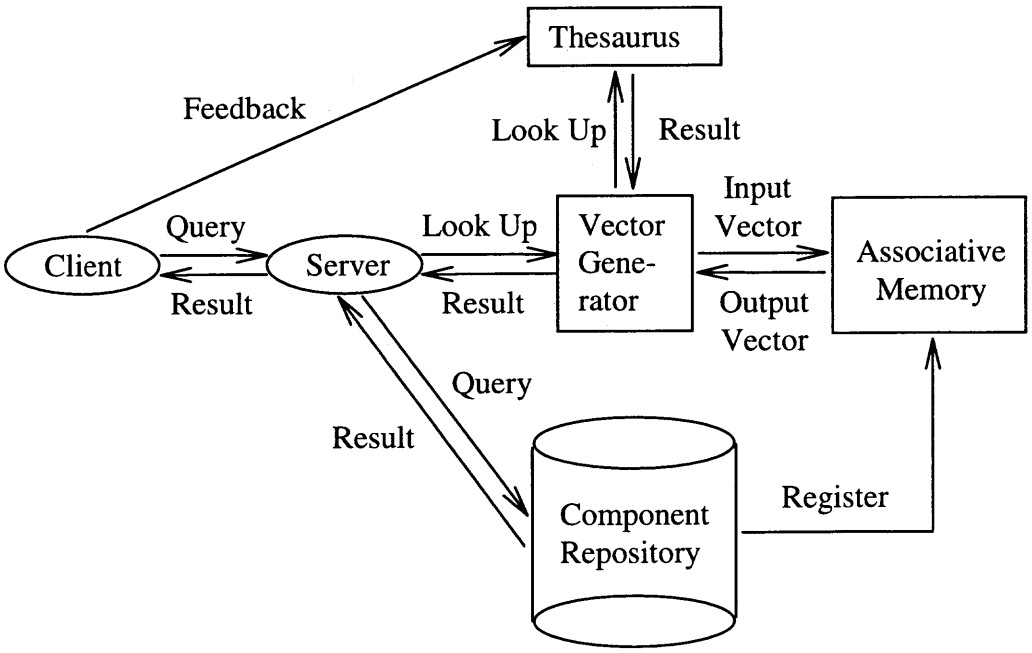

Figure 6.7 Component retrieval system architecture

Implementation

The system is implemented in C++ on a Sun Sparc 20 workstation running Sun's Solaris 2.5 operating system. It works as follows:

The client initializes a request to looking for components in the component repository. The server gets the request, parses it, and sends it to the vector generator. The vector generator contains the knowledge of the primary facet values. If the facet values are found in the primary facet values, a new input vector is generated by setting values on the bits that represents the facet values that the user is interested. The vector generator also forwards the facet values to the thesaurus whether it finds them in the primary layer or not.

The thesaurus looks for the facet values in the primary layer. If it finds them, the facet values in the secondary layer that are pointed to by the primary facet values are also retrieved to set the input vector. The relevancy between primary facet values and secondary facet values FR is calculated. If the facet values are not found in the primary layer, but in the secondary layer, the thesaurus retrieves the facet values in the primary layer that points to the found secondary facet values. The relevancy BR is calculated. Then the synonyms found by the thesaurus are returned to the vector generator. The vector generator generates new input vectors by using the synonyms and BR or FR.

The input vector is then multiplied by the synaptic connectivity matrix. New output is generated to indicate which components in the component repository contain the facet values or their synonyms. The result then is sent back to the server. The server sends the representation of the component back to the user. If the user is satisfied with the search result, he sends the request again to the server to retrieve the component by presenting the unique identifier to the server. The thesaurus takes the response of a user and uses it to update the variables in Table 6.2 for future usage. The server then queries the component repository to get the component.

The component repository performs routine updates, such as adding new components, deleting old components and replacing old versions of components by new versions. It also reports such changes to the associative memory so that the associative memory keeps the most up-to-date representation of the components. Every time the state of component repository changes, the neural associative memory adjusts its memory accordingly. This work may be done during a less busy time of a day as in the middle of the night.

The following problems came up during our experimentation while putting the system in a prototypical implementation.

When we first designed the system, we used the tradition client server model to handle the request. When a request comes in, the server spawns a child process to handle the request. However, one problem with the associative memory is its relatively expensive usage of computer memory.

Recall that each associative memory remembers all the possible values for one facet, say n, which means we need a vector of n bits to represent the values. We need another m-bit vector to represent the m components in the component repository. Suppose each component has at most C unique values for each facet. Then the number of possible values n for one facet is $C \times m$. The number of bits of the input vector therefore is $n = C \times m$. Then the size of a neural associative network is $n \times m = C \times m \times m = O(m^2)$. For a mid size component repository, say it contains thousands of components, it would be a limited concern of memory space. But when we have hundreds of users sending queries to the same server at the same time, it would be a problem, because there are hundreds of child processes running and each of them tries to allocate a memory space for the associative memories.

Therefore we changed our classic client server model to the persistent server model. In this model, when the system first starts up, the server spawns a certain number of child processes. When a request comes in, it is put in a queue by the server. Any idle child can read a query from the queue, and process the query. When no idle child is available, the queries are stored in the queue and wait for the first available child to pick

up. This model also saves CPU time by eliminating overhead to start and shutdown child processes.

Furthermore, instead of using child process, we use threads to handle the processing. When spawning a child thread, the child thread is in the same address space of the main thread. This makes it possible to pass in the address of the synaptic connectivity matrix to the child threads for them to process. So child threads do not need to load the whole matrix as opposed to what the child processes do. This thread model further saves memory space for our prototype. The number of child threads is tunable. In the real application, the application administrator can do an estimation of the pattern of user activity, like what is the peak time of a day, what is the possible maximum number of users who access the server at the same time, etc. Based on this information, the system can be tuned to best fit users' needs.

Experiment Results

In addition to the implementation of the prototype, we also used it to verify the sufficiency and accuracy of our proposed methods. Research approaches that have been done on component repositories usually use different ways to represent and retrieve components (refer to Chapter 5). There is no available benchmark system for us to test against. Following the experiments done in [13], we design the experiments to show the feasibility and correctness of our proposed methods instead of testing against other available systems.

The component base is the most difficult part to standardize in this experiment. Again following [13], we use a family of 30 Unix commands that are used to create,

modify, and delete files, pipes and sockets as our component base. Instead of using 10 facets proposed in the component representation, we use 2 facet lists to represent the components. This is feasible due to the fact that our representation is highly scalable, so 2 facet lists is enough for the purpose of demonstrating the feasibility of the proposed methods.

One important issue in the experiment is to determine the threshold value. As addressed in [13], "determining a way to compute thresholds other than by trial and error is still an unsolved problem in text retrieval research", we believe it is the same situation for component retrieval research. In our experiments, we will change the value of the threshold to see the impact of it on the overall system performance.

The precision rate and recall rate are two concepts that have traditionally been used to evaluate retrieval methods. Following the definition in [46][62], let Q be a set of components that should be returned as a response to a query, and let R be a set of components that are actually returned to a query. Then the precision rate and recall rate are defined as:

$$\text{Precision}(R) = |R \cap Q|/|R|$$

$$\text{Recall}(R) = |R \cap Q|/|Q|$$

The precision rate is the proportion of retrieved items that are relevant, measured by the ratio of the number of relevant retrieved items to the total number of retrieved items. The recall rate is the proportion of relevant items retrieved, measured by the ratio of the number of relevant retrieved items to the total number of relevant items in the collection. For a retrieval system, a high precision rate and a high recall rate are

favorable, which means the retrieval system is able to retrieve relevant components with a small number of irrelevant components.

A group of Unix C, C++ programmers and Unix, NT system administrators was asked to search the Unix commands in the component repository. The experiment participants were taught the syntax for the query and how to use the component retrieval system first. Then they were free to use words and phrases to compose their queries based on their knowledge toward Unix commands. They recorded the number of queries they tested, the number of retrieved components and the number of components they actually wanted and were in the repository. When testing the ranking of retrieved components, the experiment participants used a scale of 1-10 to denote how satisfied they were with the ranking with 10 being the best and 1 the worst.

The experiments are organized in the following order:

1. Testing Precision(R) with threshold changing
2. Testing Recall(R) with threshold changing
3. Testing Precision(R) with thesaurus and threshold changing
4. Testing Recall(R) with thesaurus and threshold changing
5. Testing ranking without weight function
6. Testing ranking with weight function

The results of experiment 1 to 4 are shown in Figure 6.8 and Figure 6.9. An obvious trend for the change of recall rate with respect to the change of threshold is observed. The result suggests that when a threshold value is lower, more components will be retrieved. Thus high recall rate is more likely since the queried components are more

likely to be returned in the candidate components. When the threshold is high, low recall rate is more likely because less candidate components are retrieved from the component repository. However the trend for the precision rate is not apparent. When the threshold is small, the number of components retrieved is large. Thus the precision rate is relatively low. When the threshold becomes higher, the precision rate should become higher because the number of component retrieved becomes smaller. However the number of desired components is also becoming smaller which offsets the change of the number of retrieved components to some degree. Thus the change of precision rate is not so obvious as that of recall rate.

When a thesaurus is applied, the recall rate is relatively higher, and precision rate is relatively lower because more candidate components are retrieved, provided the experiments are conducted using the same threshold value. The result confirms that it is hard to get both high precision and recall rate for retrieval systems as proved in information retrieval system. Our system shows that for our component repository, a threshold of 0.3 or 0.4 is favorable because it achieves relatively high precision rate while it is also acceptable for the recall rate. Usually for a component retrieval system with a small component repository, the recall rate is more important than precision rate since the user likes to see as many relevant components as possible in one retrieval process. When a component repository is big, however, the precision rate also has to be taken into account to limit the number of components returned by the retrieval system. Otherwise the users will quickly lose interest using the retrieval system if they have to filter hundreds of returned components to find the ones they are interested in. A balanced choice between precision rate and recall rate needs to be given.

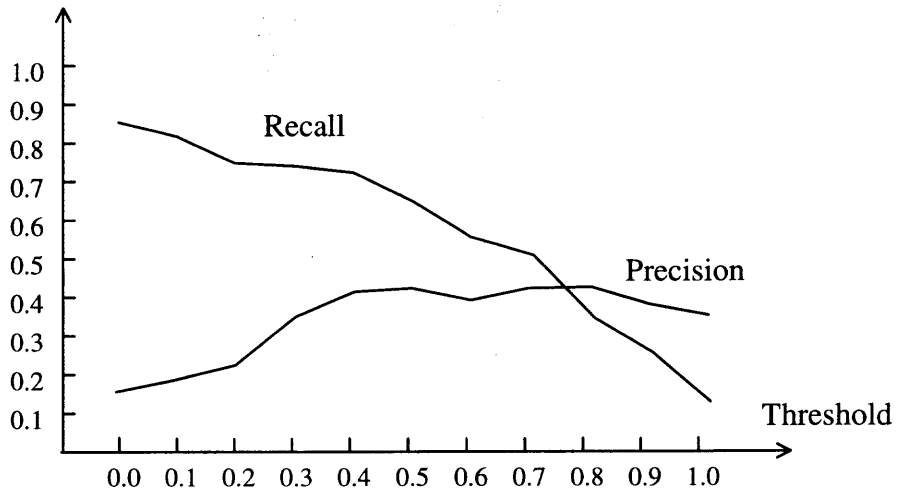

Figure 6.8 Precision and recall rate without thesaurus

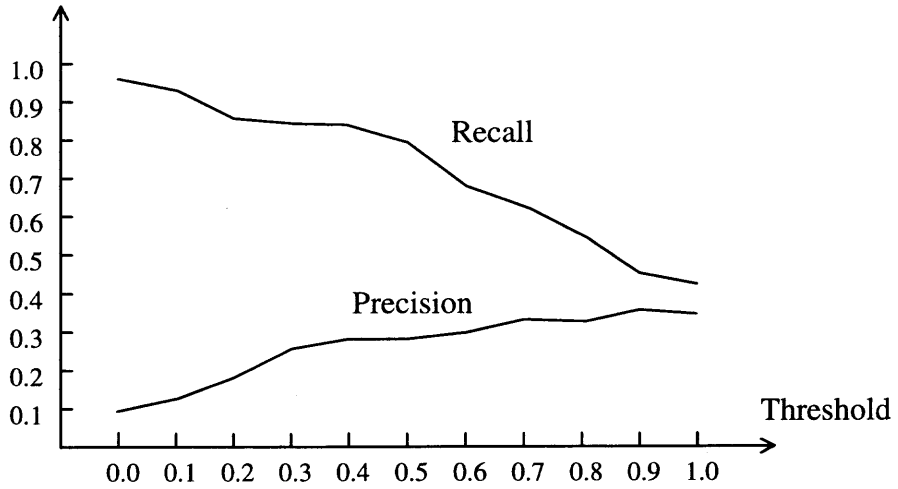

Figure 6.9 Precision and recall rate with thesaurus

For the experiments of ranking of components, i.e. experiment 5 and 6, we set the threshold to 0.3 which enables the retrieval system to retrieve most of the desired components. The average score for retrieval without weight adjustment in the associative memory is 6.3 while it is 7.4 for retrieval with weight adjustment. These two scores,

although as expected, are less convincing than the precision rate and recall rate in our experiment because of the possible bias of experiment participants and the relatively small size of component repository. However, all of our participants feel the difference among the experiments conducted with and without the use of weight function.

6.5 Summary

In this chapter, we have proposed a method, which is combined of neural associative memory, information retrieval, thesaurus, and Bayesian inference technologies, to find components in a component repository. The neural associative memory is able to memorize the relationship between components and facet values when the weight function is able to specify the closeness of the relationships. Bayesian inference enables the thesaurus to change its weights between synonyms so that the thesaurus is able to take into account users' feedback and adapt to users' preference.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this dissertation, we established an infrastructure for component-based software development. The topics we studied include software planning, software building from legacy systems, component represenation, and software component retrieval.

We proposed a software development process model and a definition of component-based software development team roles. They are different from their counterpart in the traditional software engineering in the sense that they facilitate the development of software systems based on reusable software components. The proposed software development process model achieves maximum software component reuse by conducting domain engineering and application engineering simultaneously to map a software system to a set of existing components. The definition of team roles adds unique team roles that exist only in the component-based software engineering. By giving a clear definition of team roles, it helps to achieve the maximum usage of team resources.

Unlike most studies that focus on how to build new components by using the most advanced component technology, we studied component building in an opposite direction: how to build components from legacy systems. This is more difficult than building components from scratch since there are many technical restrains on transforming legacy systems to component-based systems. We proposed a framework to decompose legacy systems and build components by extracting component candidates and wrapping them. A specific methodology is also proposed to facilitate a special case of component building: the integration of legacy systems with Internet applications. By

using this methodology, it is possible to wrap a legacy system in a timely manner and make it available for newly emerging and demanding Internet services.

Another issue that was studied is how to retrieve a software component in a component repository. We believe the success of component technology depends on the availability of a mature component market that stores and distributes components. If such a market is at hand, then how do people find the wanted components? To solve this question, we first proposed a framework to represent components. This component representation framework uses double facet lists to represent components. The relatively simple yet powerful scheme of facet representation is able to accommodate the big, up-coming component market. More importantly, the inherent structure of the representation facilitates efficient component retrieval.

Based on the representation framework, an efficient retrieval method adopting the neural associative memory technology was developed to find and retrieve components from component repositories. The neural associative memory memorizes facet values in a facet list that is used to represent components. In one processing step, it can accurately and quickly find all the components that contain the query facet values. The strength of this method is that by training an associative memory offline, it achieves online efficiency when it interacts with users.

Using the associative memory as a starting point, a weight function derived from information retrieval technology was proposed to refine the associative memory. The weight function makes the associative memory memorize not only the relation between facet values and components, but also the strength of the relation. Experiments show that the weight function improves the retrieval precision rate. Bayesian inference is used to

improve the performance of the associative memory. In order to provide synonym support, a two-layer, dynamically adjusted thesaurus using Bayesian inference was proposed. The thesaurus is able to adjust the weights between synonyms by taking into account user preference during the retrieval activities. This thesaurus model improves both the precision rate and recall rate of the associative memory. In order to lower the computational expenses of matrix multiplication, we also looked into available sparse matrix multiplication algorithms and parallel algorithms to help the associative memory further cut its response lag.

A prototype of a component repository is built to demonstrate the power of the proposed retrieval method combined of neural associative memory, information retrieval, thesaurus, and Bayesian inference technologies. The experiments showed the consistency between the theoretical results and the practical results. In addition to the further study and experimentation, the component retrieval mechanism is also under investigation for a component-based approach to knowledge management [37]. In this case, components contain knowledge instead, but the overall system design and construction principles are quite similar.

REFERENCES

1. R. C. Agarwal, F. G. Gustavson, M. Zubair, "A High Performance Algorithm Using Pre-Processing for the Sparse Matrix-Vector Multiplication," in *Proceedings of Supercomputing*, pp. 32-41, 1992.

2. P. Allen, S. Frost, *Component Based Development for Enterprise Systems: Applying the Select Approach*, Cambridge University Press, New York, 1997.

3. J. A. Anderson, "A Simple Neural Network Generating an Interactive Memory," *Mathematical Biosciences*, No. 14, pp.197-220, 1972.

4. D. Batory, S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transaction on Software Engineering and Methodology*, Vol. 1, No. 4, pp. 355-398, 1992.

5. R. Bellinzona, M. G. Fugini, B. Pernici, "Reusing Specifications in OO Applications," *IEEE Software*, Vol. 12, No. 2, pp. 65-75, 1995.

6. B. Boehm, "A Spiral Model for Software Development and Enhancement," *Computer*, Vol. 21, No. 5, pp. 61-72, 1988.

7. G. Booch, *Software Components with Ada: Structures, Tools, and Subsystems*, Addison-Wesley, Reading, Massachusetts, Harlow, England, Menlo Park, California, Berkeley, California, Don Mills, Ontario, Sydney, Bonn, Amsterdam, Tokyo, Mexico City, 1990.

8. F. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Massachusetts, 1975.

9. A. W. Brown, K. C. Wallnau, "The Current State of CBSE," *IEEE Software*, Vol. 15, No. 5. pp. 37-46, 1998.

10. S. Browne, J. Dongarra, S. Green, K. Moore, "Location-Independent Naming for Virtual Distributed Software Repositories," in *Proceedings of the 17th International Conference on Software Engineering on Symposium on Software Reusability*, pp. 179-185, 1995.

11. A. Cimitile, A. D. Lucia, G.A. Lucca, A.R. Fasolino, "Identifying Objects in Legacy Systems," in *Proceedings of IEEE 5th International Workshop on Program Comprehension*, pp. 138-147, March, 1997.

12. G. Coulouris, J. Dollimore, T. Kindberg, *Distributed System: Concepts and Design*, Addison-Wesley, Harlow, England, Reading, Massachusetts, Menlo Park, California, New York, Don Mills, Ontario, Amsterdam, Bonn, Singapore, Tokyo, Madrid, San Juan, Milan, Mexico City, Seoul, Taipei, 1994.

13. E. Damiani, M. G. Fugini, C. Bellettini, "A Hierarchy-Aware Approach to Faceted Classification of Object-Oriented Components," *ACM Transaction on Software Engineering and Methodology*, Vol. 8, No. 3, pp. 215-262, 1999.

14. A. Davis, *Software Requirements: Objects, Functions, and States*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

15. M. Dewey, *Decimal Classification and Relative Index*, 19th ed., Forest Press Inc., Albany, New York, 1979.

16. W. Dietrich, I. Nachkman, L. Gracer, "Saving a Legacy with Objects," in *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pp. 77-88, New Orleans, LA, 1989.

17. D. D'Souza, A. C. Wills, *Objects, Components, and Frameworks with UML: The CataLysis Approach*, Addison Wesley Longman, Reading, Massachusetts, 1998.

18. D. G. Firesmith, *Object-Oriented Requirements Analysis and Logical Design*, Wiley, New York, 1993.

19. W. B. Frakes, B. A. Nejmeh, "An Information System for Software Reuse," in W. Tracz (editor), *Software Reuse: Emerging Technology*, IEEE Computer Society Press, Monterey, California, 1988.

20. W. B. Frakes, C. Terry, "Software Reuse: Metrics and Models," *ACM Computer Survey*, Vol. 28, Issue. 2, pp. 415-435, 1996.

21. G. W. Furnas, T. K. Landauer, L. M. Gomez, S. T. Dumais, "The Vocabulary Problem in Human-system Communication," *ACM Transaction on Communication*, Vol. 30, No. 11, pp. 964-971, 1987.

22. D. Garlan, A. Robert, and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," *IEEE Software*, Vol. 12, No. 6, pp. 17-26, 1995.

23. L. Gilman, R. Schreiber, L. Gilman, *Distributed Computing With IBM MQSeries*, John Wiley, 1996.

24. S. Gupta, E. Rothberg, "DME: A Distributed Matrix Environment," in *Proceedings of the Scalable High-Performance Computing Conference*, pp. 629-636, 1994.

25. C. Hall, *Building Client/Server Applications Using Tuxedo*, John Wiley, New York, 1996.

26. D. O. Hebb, *The Organization of Behavior*, Wiley, New York, 1949.

27. W. S. Humphrey, M. Lovelace, Ryan Hoppes, *Introduction to the Team Software Process*, Addison-Wesley, Reading, Massachusetts, Harlow, England, Menlo Park, California, Berkeley, California, Don Mills, Ontario, Sydney, Donn, Amsterdam, Tokyo, Mexico City, 1999.

28. I. Jacobson, F. Lindstrom, "Re-engineering of Old Systems to an Object Oriented Architecture," in *Proceedings of OOPSLA91*, pp. 340-350, 1991.

29. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, Harlow, England, Menlo Park, California, Berkeley, California, Don Mills, Ontario, Sydney, Donn, Amsterdam, Tokyo, Mexico City, 1994.

30. *Java Message Service FAQ*, http://java.sun.com/products/jms/faq.html, Feb. 2000.

31. J. Jeng, B. Cheng, "A Formal Approach to Reusing More General Components," in *Proceedings of IEEE 9th Knowledge-Based Software Engineering Conference*, pp. 90-97, Monterey, California, Sept. 1994.

32. L. Jololian, "A Meta-Semantic Language for Smart Component Adapters," *Doctoral Dissertation,* Department of Computer and Information Science, New Jersey Institute of Technology, Newark, New Jersey, 2000.

33. N. K. Kasabov, *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering*, The MIT Press, Cambridge, Massachusetts, London, England, 1996.

34. B. Keith, "Legacy Systems: Coping With Success," *IEEE Software,* Vol. 12, No. 1, pp. 19-23, 1995.

35. E. H. Khan, M. R. Girgis, "The Effect of OO Life Cycle on Software Project Management," in *Proceedings of International Conference on Engineering and Technology Management*, pp. 233-240, 1996.

36. T. Kohonen, "Correlation Matrix Memory", *IEEE Transaction on Computer*, Vol. 21, No. 4, pp. 353-359, 1972.

37. F. J. Kurfess, "Component-Based Knowledge Management," submitted for publication.

38. F. J. Kurfess, "Neural Networks and Structured Knowledge," *Special Issue of Journal of Applied Intelligence*, Vol. 11, No. 1, No. 2, 1999.

39. C. Landauer, K. L. Bellman, "Lessons Learned from Wrapping Systems," in *Proceedings of 5th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 132-142, 1999.

40. L. Latour, E. Johnson, "SEER: A graphical Retrieval System for Reusable Ada Software Modules," in *Proceedings of 3rd International IEEE Conference of Ada Applications and Environments*, pp. 105-113, 1988.

41. P. M. Lee, *Bayesian Statistics: An Introduction*, Oxford University Press, New York, 1997.

42. H. Liao, M. Chen, F. Wang, J. Dai, "Using a Hierarchical Thesaurus for Classifying and Searching Software Libraries," in *Proceedings of the 21st Annual International Computer Software and Application Conference*, pp. 210-216, 1997.

43. S. Liu, N. Wilde, "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 266-271, 1990.

44. J. Llorens, A. Amescua, M. Velasco, "Software Thesaurus: a Tool for Reusing Software Objects," in *Proceedings of 4th International Symposium on Assessments of Software Tools*, pp. 99-103, 1996.

45. A. D, Lucia, G. A. Lucca, A. R. Fasolino, P. Guerra, S. Petruzzelli, "Migrating Legacy Systems towards Object-Oriented Platforms," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 122-129, 1997.

46. Luqi, Jiang Guo, "Toward Automated Retrieval for a Component Repository," in *Proceedings of IEEE Conference and Workshop on Engineering of Computer-Based System*, pp. 99-105, 1999.

47. Y. Maarek, D. Berry, G. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transaction on Software Engineering*, pp. 800-813, Vol. 17, No. 8, 1991.

48. J. McDermid, P. Rook, "Software development Process Models," in *Software Engineer's Reference Book*, pp. 15/26-15/28, CRC Press, Florida, 1993.

49. D. McIlroy, "Mass-produced Software Components," in *Proceedings of the 1968 and 1969 NATO Conferences*, pp. 88-98, 1969.

50. R. Mili, A. Mili, R. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System," *IEEE Transaction on Software Engineering*, Vol. 23, No. 7, pp. 445-460, 1997.

51. R. Monson-Haefel, *Enterprise JavaBeans*, O'Reilly & Associates, Beijing, Cambridge, Paris, Sebastopol, Taipei, Tokyo, 1999.

52. H. A. Muller, K. Wong, M. A. Storey, "Wrapping Coarse-Grained Objects Using Standard Infrastructure Technology," in *Proceedings of IEEE International Conference on Software Maintenance*, pp 301, 1997.

53. J. Q. Ning, A. Engberts, W. Kozaczynski, "Recovering Reusable Components from Legacy Systems by Program Segmentation," in *Proceedings of IEEE 4th Working Conference on Reverse Engineering*, pp. 64-72, 1993.

54. R. Orfali, D. Harkey, J. Edwards, R. Crfali, *Instant CORBA*, John Wiley & Sons, New York, 1997.

55. E. Ostertag, J. Hendler, R. Prieto-Diaz, C. Braun, "Computing Similarity in a Reuse Library System: An AI-Based Approach," *ACM Transaction on Software Engineering and Methodology*, Vol. 1, No. 3, pp. 205-228, 1992.

56. G. Palm, F. Schwenker, A. Strey, F. J. Kurfess, "Neural Associative Memories," Technical Report, University of Elm, 1994.

57. R. Penteado, P. C. Masiero, A. F. Prado, R. T. V. Braga, "Reengineering of Legacy Systems Based on Transformation Using the Object Oriented Paradigm," in *Proceedings of IEEE 5th Working Conference on Reverse Engineering*, pp. 144-153, 1998.

58. R. Prieto-Diaz, "A software Classification Scheme," *Doctoral Dissertation*, Department of Computer Science, University of California, Irvine, California, 1985.

59. R. Prieto-Diaz, "Implementing Faceted Classification For Software Reuse," *Communication of ACM*, Vol. 35, No. 5, pp. 89-97, 1991.

60. R. Prieto-Diaz, G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, Los Alamitos, California, 1991.

61. W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," in *Proceedings of the 9th International Conference on Software Engineering*, pp. 328-338, 1987.

62. G. Salton, C. Buckley, "Term-weighting Approaches in Automatic Text Retrieval," *Information Processing and Management*, Vol. 5, No. 24, pp. 513-523, 1988.

63. J. Sametinger, *Software Engineering with Reusable Components*, Springer Verlag, New York, 1997.

64. R. Sessions, *COM and DCOM: Microsoft's Vision for Distributed Objects*, John Wiley & Sons, New York, 1997.

65. H. Sneed, "Encapsulation Legacy Software for Use in Client/Server Systems," in *Proceedings of 3rd IEEE Working Conference on Reverse Engineering*, pp. 104-119, IEEE Computer Society Press, Monterey, California, 1996.

66. H. Sneed, E. Nvary, "Extracting Object-Oriented Specification from Procedurally Oriented Programs", in *Proceedings of IEEE Working Conference on Reverse Engineering*, pp. 217-226, Toronto, Canada, 1995.

67. H. Sneed, "Planning The Reengineering of Legacy System," *IEEE Software,* Vol. 12, No. 1, pp. 24-34, 1995.

68. J. Solderitsch, K. Wallnau, J. Thalhamer, "Constructing Domain-specific Ada Reuse Libraries", in *Proceedings of 7th Annual National Conference on Ada Technology*, pp. 125-134, 1989.

69. K. Steinbuch, *Mensch und Automat.* Springer, Heidelberg, 1961.

70. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ACM Press, New York, Addison-Welsley, Harlow, England, Reading, Massachusetts, Menlo Park, California, New York, Don Mills, Ontario, Amsterdam, Bonn, Sydney, Singapore, Tokyo, Milan, Madrid, San Juan, Mexico City, Seoul Taipei, 1997.

71. A. Taivalsaari, R. Trauter, and E. Casais, "Workshop on Object-Oriented Legacy Systems and Software Evolution," in *Proceedings of 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 180-185, 1995.

72. Y. Tang, "A methodology for Component Based System Integration," *Doctoral Dissertation,* Department of Computer and Information Science, New Jersey Institute of Technology, Newark, New Jersey, 1998.

73. W. Tracz, "Where Does Reuse Start?" in *Proceedings of Realties of Reuse Workshop*, Syracuse University CASE Center, pp. 14-20, 1990.

74. T. C. Valesky, *Enterprise JavaBeans: Developing Component-based Distributed Applications*, Addison Wesley, Harlow, England, Reading, Massachusetts, Menlo Park, New York, Don Mills, Amsterdam, Bonn, Sydney, Milan, Madrid, Mexico City, 1999.

75. I. Warren, L. W. Chang, L. Xu, *The Renaissance of Legacy Systems: Method Support for Software-System Evolution*, Springer Verlag, New York, 1999.

76. J. B. White, "On Improving the Performance of Sparse Matrix-Vector Multiplication," in *Proceedings of IEEE 4th International Conference on High-Performance Computing*, pp. 66-71, 1997.

77. S. White, M. Edwards, "Domain Engineering: The Challenge, Status, and Trends, " in *Proceedings of IEEE Symposium and Workshop on Engineering of Computer-Based Systems*, pp. 96-101, 1996.

78. P. Winsbery, "Legacy Code: Don't Buy It, Wrap It," *Datamation*, pp. 36-41, May 1995.

79. R. Wirfs-Brock, B. Wilkerson, L. Weiner, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

80. A. Zaremski, J. Wing, "Signature Matching: A Tool for Using Software Libraries," *ACM Transaction on Software Engineering and Methodology*, Vol. 4, No. 2, pp. 146-170, April 1995.