

# Automating Design Intent Capture For Component Based Software Reusability

by

Siva Kumar Dirisala

B.Tech, Civil Engineering  
Indian Institute of Technology, Madras (1995)

Submitted to the Department of Civil and Environmental  
Engineering

in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

© Massachusetts Institute of Technology 1998. All rights reserved.

Author .....

Department of Civil and Environmental Engineering

May 19, 1998

Certified by .....

Feniosky Peña-Mora

Assistant Professor of Civil and Environmental Engineering

Thesis Supervisor

Accepted by .....

Joseph M. Sussman

Chairman, Department Committee on Graduate Studies

JUN 02 1998

LIBRARIES

# **Automating Design Intent Capture For Component Based Software Reusability**

by

Siva Kumar Dirisala

Submitted to the Department of Civil and Environmental Engineering  
on May 19, 1998, in partial fulfillment of the  
requirements for the degree of  
Master of Science

## **Abstract**

With the growing interest for open architectures and standards, it is likely that reuse methods would be part of the software development life cycles. Reuse of software components requires the technology to build easily reusable components and support systems for cataloging and retrieving the right components. Object oriented programming and component object models have made it possible to build easily reusable components. Component technologies like COM and JavaBeans make it possible to build large systems by seamlessly integrating several components based mainly on their interface instead of implementation details. However, it is still necessary to identify the right components that offer the required functional features. Technology for classifying and retrieving the correct components is still far from expectation because earlier methods have used mainly either semantic information or textual information. Systems based on semantic information often required learning new specification languages while those based on textual information resulted in poor classification and retrieval. This research uses both types of information to make use of the complementary advantages of each and provides a better reuse support system. Performance from using textual information can be improved by making use of semantic knowledge derived from simple analysis on existing semantic information. Capturing design intent, which gives the functional features of a component, can be automated by using both types of information. A cost effective reuse technology is provided by mainly using the existing project information that is produced without reuse concern. While cost is an issue, it is also necessary to make the end user interface simple without imposing the additional burden of learning new specification languages or theorem proving techniques to retrieve components. This research presents a cost effective, easily usable and efficient reuse technology for automating the capture of design intent and retrieving multiple components to build large systems.

Thesis Supervisor: Feniosky Peña-Mora

Title: Assistant Professor of Civil and Environmental Engineering

## Acknowledgments

First and foremost I would like to thank Feniosky Peña-Mora for his support, guidance and constant encouragement. The attitude I have experienced from him is *even a drop of creativity in an ocean of research is significant* and this very thought was a motivating force in my research efforts.

I would like to acknowledge the support received from the School of Engineering, the Department of Civil Engineering and NSF-CONACyT grant (Project Number IRI-9630021).

I would like to thank the Head of the Department, Prof. Rafael Brass and the administrative staff of the department who work behind the screens and make sure that the students' academic life runs smoothly.

Pursuing higher studies requires a stronger foundation of fundamentals and the credit in shaping my *technical personality* goes to my Alma mater Indian Institute of Technology, Madras (IIT-M).

My special thanks to my undergraduate classmate Niranjan Krishnan (Ninja) who helped me in coming to MIT and adjusting to US life. All those memorable weekend dinners with Battu, Ghost, KK, Ninja and Rugby made me feel like still studying in IIT-M instead of MIT.

Several friends at MIT helped in having a *fun life* abridge the academic and research pressures. Chandra B, Chandra P, Deepak, Gangs, JP, Neeraj, Pulp, Sri and Subbu are some of them.

Apart from academic and research advisors (who happened to be the same for me right from the beginning), it is nice to have advice from others and I always had OP, Ninja and Auroop for that.

Being in a friendly research group is very important and the DaVinci group has provided such an environment with people like Lucio, Kareem, Sanjeev, Chun-Yi and Yee-Sue. Thanks to Thomas, Nina and Gordon for tolerating my computer occupancy, being friendly and sharing the *den*, 1-270. Special thanks to Joan McCusker who takes care of all the administrative work of the IT group. I thank Petros, another IESL

labmate, for giving company several nights for doing assignments and doing our term project in Graphics course. He is one of the very nice guys I met in MIT.

I express my deepest love to my parents for helping me in being what I am today and always letting me do whatever I wanted to do. I would also like to express my deepest love to my sister and brother for their affection and source of encouragement.



# Contents

<b>1</b>	<b>Introduction</b>	<b>12</b>
1.1	Software Reuse and the Advantages . . . . .	13
1.1.1	What can be Reused . . . . .	13
1.1.2	Advantages of Software Reuse . . . . .	14
1.2	Technology to Develop Reusable Software Components . . . . .	15
1.2.1	Object Oriented Paradigm . . . . .	15
1.2.2	Component Object Model . . . . .	16
1.3	Software Development by Integrating Components . . . . .	16
1.4	Scope of the Research . . . . .	18
1.5	Roadmap to the rest of the Thesis . . . . .	20
<b>2</b>	<b>Background</b>	<b>22</b>
2.1	Semantic Approach . . . . .	23
2.1.1	Parameterized Programming, OBJ Language . . . . .	23
2.1.2	Structured Algebraic Specification, Clear Language . . . . .	24
2.1.3	Partially Interpreted Schemas, PARIS . . . . .	25
2.1.4	Object Oriented Module Interconnection Language, MIL . . . . .	27
2.1.5	Summary of Semantic Approach . . . . .	27
2.2	Statistical Approach . . . . .	28
2.2.1	Faceted Classification Scheme . . . . .	29
2.2.2	Conceptual Schemas . . . . .	30

<i>CONTENTS</i>	6
2.2.3 Summary of Statistical Approach . . . . .	32
2.3 Hybrid Approach . . . . .	32
2.4 Design Recommendation and Intent Model Extended for Reusability (DRIMER) . . . . .	33
<b>3 Requirements Analysis</b>	<b>37</b>
3.1 Information in Software Projects . . . . .	38
3.1.1 Classification of Information used in Software Projects . . . . .	39
3.1.2 Observations . . . . .	42
3.2 Information Search - A Case Study . . . . .	42
3.2.1 Search for a TextBook/Journal in a Library . . . . .	43
3.2.2 Observations . . . . .	44
3.3 Retrieving Multiple Components . . . . .	46
3.3.1 Retrieving and Ranking . . . . .	46
3.4 Framework and Customization . . . . .	47
3.5 Summary of Requirements . . . . .	48
<b>4 Conceptual Model</b>	<b>50</b>
4.1 Existing Reuse Models . . . . .	51
4.1.1 Reuse based on Available Textual Information . . . . .	51
4.1.2 Reuse based on Available Semantic Information . . . . .	52
4.2 New Reuse Model . . . . .	54
4.2.1 Reuse based on both Textual and Semantic Information . . . . .	54
4.3 Class Tree Provides Better Weights for Statistical Analysis . . . . .	55
4.3.1 Assumptions . . . . .	56
4.3.2 Class Tree . . . . .	56
4.4 Creational Patterns Suggest Abstractions of Functional Features . . . . .	58
4.5 Software Component Representation . . . . .	59
4.5.1 Simple External Representation . . . . .	60

<i>CONTENTS</i>	7
4.5.2 Complex Internal Representation . . . . .	61
4.6 Multiple Software Component Retrieval . . . . .	62
4.6.1 Reggia's Generalized Set Cover Algorithm . . . . .	65
4.6.2 Multiple Software Component Retrieval . . . . .	66
4.6.3 Ranking the Retrieved Solutions . . . . .	68
4.7 Summary of the Conceptual Model . . . . .	69
<b>5 Prototype Framework and Tools</b>	<b>71</b>
5.1 Framework . . . . .	72
5.1.1 Client . . . . .	73
5.1.2 Server . . . . .	75
5.2 Tools . . . . .	78
<b>6 Illustrative Examples</b>	<b>81</b>
6.1 Client . . . . .	81
6.1.1 Specifying Constraints and Features . . . . .	82
6.1.2 Selecting for Composibility, Relevancy or Reusability . . . . .	86
6.1.3 Online Software Component Browsing and Report Generation	90
6.2 Server . . . . .	91
6.2.1 Adding New Software Component . . . . .	92
6.2.2 Browsing and Modifying Existing Components . . . . .	93
6.2.3 Browsing Existing Software Features . . . . .	94
6.3 Tools . . . . .	95
6.3.1 Chat Application . . . . .	95
6.3.2 Analysis using Textual Information . . . . .	95
6.3.3 Class Tree, a semantic analysis . . . . .	97
6.3.4 Analysis using Textual Information and Semantic Information	97
<b>7 Conclusions</b>	<b>101</b>
7.1 Conclusions . . . . .	101

<i>CONTENTS</i>	8
7.2 Future Research Directions . . . . .	103
<b>A OOP and Design Patterns</b>	<b>105</b>
A.1 Object Oriented Programming (OOP) . . . . .	105
A.2 Design Patterns . . . . .	107
A.2.1 Creational Design Patterns . . . . .	107
<b>B COM and JavaBeans</b>	<b>109</b>
B.1 Component Object Model, COM . . . . .	109
B.2 JavaBeans . . . . .	111
<b>C CORBA</b>	<b>113</b>
<b>D Generalized Set Cover Algorithm</b>	<b>115</b>
D.1 Generalized Set Cover (GSC) . . . . .	115
D.1.1 GSC Pseudo code . . . . .	115
D.1.2 Minimum Cardinality Set Solution . . . . .	117
D.1.3 Light Weight Set Solution . . . . .	118

# List of Figures

1-1	Generic framework customization for specific domain applications . . .	13
2-1	Parameterized List object representation using OBJ Language . . . . .	24
2-2	Pseudo-Clear code for Group with Equivalence Relation . . . . .	25
2-3	Schema Representation of Linked List iteration using PARIS . . . . .	26
2-4	Home-heating System module specified using MIL . . . . .	28
2-5	Representation of Software Components using Faceted Classifying Scheme	29
2-6	Search generalization using wildcards . . . . .	30
2-7	A sample Conceptual Schema Card . . . . .	31
2-8	DRIM model <i>adapted from [FPn94]</i> . . . . .	34
2-9	DRIMER web based system <i>adapted from [Vad96]</i> . . . . .	35
3-1	Classification of Project Information . . . . .	39
4-1	Reuse from Textual Information . . . . .	52
4-2	Reuse from Semantic Information . . . . .	53
4-3	Reuse from both Semantic and Textual Information . . . . .	54
4-4	Class tree provides relative weights for the words . . . . .	57
4-5	Software Component Representation . . . . .	60
4-6	Customization of Component Representation . . . . .	62
4-7	Software components and features represented as bipartite graph . . .	64
4-8	Example of Set Covering . . . . .	66

<i>LIST OF FIGURES</i>	10
5-1 Web Client . . . . .	74
5-2 Reusable Software Component Server . . . . .	78
6-1 Specifying constraints for the required software component . . . . .	83
6-2 Specifying software features and relaxing the constraints . . . . .	84
6-3 Results and unavailable features . . . . .	85
6-4 Searching for unavailable features by relaxing constraints . . . . .	86
6-5 Search for better Composibility . . . . .	87
6-6 Search for better Relevancy . . . . .	88
6-7 Search for better Reusability . . . . .	89
6-8 Online browsing of individual components . . . . .	90
6-9 Report of selected components generated on the browser . . . . .	91
6-10 Adding a New Software Component . . . . .	92
6-11 Browsing Software Components . . . . .	93
6-12 Browsing Software Features . . . . .	94
6-13 Class Tree of the Chat Server . . . . .	97
6-14 Class Tree of the Chat Client . . . . .	97
A-1 Structure of Factory Method <i>adapted from [GHJV95]</i> . . . . .	108
B-1 VTable to implement COM binary standard . . . . .	110
C-1 Common Object Request Broker Architecture, CORBA <i>adapted from [ORB97]</i> . . . . .	114
D-1 Pseudo code for Reggia's GSC Algorithm . . . . .	116
D-2 Pseudo code for Genset used in GSC Algorithm . . . . .	117
D-3 Selection of disease/component for Minimum Cardinality Set Solution	117
D-4 Selection of disease/component for Light Weight Set Solution . . . . .	118

# List of Tables

5.1	Software Component File Fields . . . . .	76
5.2	Datafile . . . . .	77
6.1	Constraint Attributes and valid values . . . . .	82
6.2	Results of Analysis using only Textual Information . . . . .	96
6.3	Relative Weights from the Class Tree of Chat Application . . . . .	98
6.4	Results of Analysis using Textual and Semantic Information . . . . .	99
D.1	Notation for Reggia's Generalized Set Cover Algorithm . . . . .	116

# Chapter 1

## Introduction

*Legal Regulations Enforce Recycling efforts, Open Standards Reinforce Reuse efforts*

---

---

Presently, at the turn of this century there is an implicit industrial revolution for open architectures. This is especially true in the IT industry. Both the software vendors and the consumers are preferring open architectures, the former to compete against monopoly and the later to have flexibility in seeking services from various vendors. Organizations like Object Management Group, OMG [OMG], are working towards these open architectures and standardizing the software components in terms of well defined business objects, services and facilities [BOB97, COS97, CFA97].

As a result of this industrial revolution, software reuse is no more an expensive task and it will soon be a part of the software development lifecycles. This chapter reviews the benefits of software reuse, the existing technology for developing reusable software components, presents a couple of scenarios which illustrates the development of software systems by integrating well defined software components, defines the scope of this research and provides a road map to the rest of the thesis.



## 1.1 Software Reuse and the Advantages

Many software applications share certain common functionality like data manipulation, data presentation, and use certain common algorithms for implementing these functionalities. For example, in case of systems software, one could use the same back end of a compiler with several front end compilers for different languages. Applications software is also being developed these days (by companies like 10Fold [TNF]) with a similar architecture by having a single horizontal framework (back end) and several vertical domain applications (front ends) built on top of it. The horizontal framework is reused in several domain applications as shown in Figure 1-1. With standards for the domain applications like the business objects from OMG [OMG], even the domain applications could be tailored from components of several vendors.

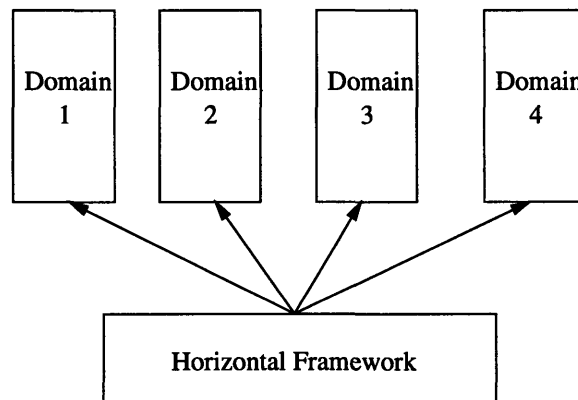


Figure 1-1: Generic framework customization for specific domain applications

### 1.1.1 What can be Reused

The final software artifact is not the only part of a software project that could be reused. It is possible to reuse all design documents, test plans along with the final source code. Infact, reusing the existing designs in the early phases of the software development process leads to reusing all the corresponding artifacts in the subsequent phases. This offers a significant reuse benefit.

However, in order to be able to reuse the design along with the final artifact design rationale and intent of use of the artifact are required. Capturing design rationale and intent is described in [PM94].

### **1.1.2 Advantages of Software Reuse**

Reusing the existing software components has the following advantages

#### **Reduction in the cost of product development**

Similar to the reduction of price in hardware with technology and open architectures, the cost of developing software systems is also decreasing. Open architectures will improve the component reuse and accelerate the reduction of developmental cost.

#### **Reduction in the duration of project**

While it took years to complete projects earlier, the demand is in months. And with the horizontal framework and vertical domain applications many companies are promising to meet this demand. The change in business policies demands for constant upgrading and enhancing of applications software and with less time to deliver the product, the business strategies and processes will no more be limited by the software systems. It would be possible to buy the best components from multiple vendors and integrate them and depend on the integrated system for the business solution. Since the system is developed using already existing components, the time to develop it would be lesser.

#### **Increase in the robustness of the product**

Using a production tested and proven component will make the new system more robust and this makes the policy makers implement several business strategies more dynamically but with minimal risk.

## 1.2 Technology to Develop Reusable Software Components

The above mentioned advantages of software reuse can be obtained only if it is possible to develop complex software systems by integrating the reusable components with a very little effort. This becomes possible if components could be integrated based on the interface they provide and not their implementation details.

If not for reuse, software has been developed from the beginning in a structured manner by using top down, bottom up and modular approaches mainly to make the design, implementation and maintenance of complex software systems tractable. With better understanding of software engineering and advent of object oriented languages the focus has broadened to software reuse as well. Microsoft's Component Object Model, COM [COM95] (Appendix B) and Sun's JavaBeans [JBn97] (Appendix B) provide a model to develop software as components which can be integrated with one another very easily. Software reuse due to both object oriented paradigm and component object models is discussed below. While object oriented paradigm provides abstractions and reuse at object level, component object models provide abstractions and reuse at component level.

### 1.2.1 Object Oriented Paradigm

Object Oriented Paradigm, OOP, provides *abstraction*, *encapsulation*, *inheritance* and *polymorphism* (all these terms are explained in detail in Appendix A). *Abstraction* and *encapsulation* helps in analyzing and organizing a complex system into manageable modules. Though these features do not directly contribute to the software reuse, they act as a means of identifying the isolated functionality and publishing the interface and hiding the implementation of the components. *Inheritance* makes it possible to design systems from general abstractions to specific instances. This provides for some software code reuse. A derived class can reuse the code from its parent class.

However, this is more like a glass box <sup>1</sup> type reuse because one has to understand the implementation details of the parent class before designing the derived class. *Poly-morphism* is another very important feature of OOP that makes the behavior of the system dependent on the objects at the runtime. This helps in writing generic flexible frameworks and customizing them as required by having a different implementation for the required interface and creating these instances.

### 1.2.2 Component Object Model

In Component Object Model the software component is viewed as a functional unit with a well defined interface. The components are integrated based on the interface they provide and without the need to understand the implementation details of the individual components. Hence, this is more like a black box type reuse. There are development environments like the JavaBeans' BeanBox tester [JBn97] that help in integrating and testing the Bean compliant software components.

While JavaBeans provide reuse of components developed in the same languages, distributed object environments like CORBA (Appendix C) and DCOM [DCO98] makes it possible to integrate software components implemented in different languages and on different platforms.

## 1.3 Software Development by Integrating Components

With the emerging open architectures in all industrial domains and reuse enabling technologies like COM, more and more companies will meet their IT needs by purchasing components from various vendors and seamlessly integrating them.

---

<sup>1</sup>Black box testing and Glass box testing are two types of software testing. Black box testing is not concerned with the implementation details. Whether or not input maps to the expected output according to the test plans is verified in this type of testing. Glass box testing on the other hand is done by going through the code covering all possible paths of execution.

Two examples are given below to illustrate the situations where software systems are developed by integrating existing components. The first example is in an industrial setting, the second is for developing a web page for personal use.

### **Example 1**

Suppose Motawala, a chip manufacturing company, decides to automate their supply chain management, they have the option of developing the entire software themselves or buying the necessary software components from leading companies in supply chain management like I2 technology or Manugistics. If these companies develop their components in compliance with OMG's Manufacturing Business Objects, then Motawala could integrate any third party component along with the software from these companies. They need not wait for either of these companies to offer the lacking features in their existing products. However, after the integration if Motawala finds that the third party component is not offering good performance and finds yet another vendor offering a high performance component, they could just buy the component from this new vendor and integrate with their existing software. It is like plug and play for hardware.

### **Example 2**

More and more homepages are coming up on the web and everyone wants to be as creative as possible. But not everyone is a java [CH96] programmer or a Javascript [JS97] writer to make their own web pages attractive. However, there are several applets and script functions already written and available freely on the web [GAM, FRC]. These could offer functionality like menus, fancy buttons, toolbars or some sophisticated applets like .dvi (DeVice Independent format used to display  $\text{\LaTeX}$ [Lam86] files) formatters. Hence, one could collect several such applet components on the web and prepare their web pages without really knowing the implementation details of these applets. All they need to know is what functionality the applets provide, what

parameters are passed to these applets and what their values should be.

The above examples shows that software development using existing components is by itself can be a new form of software development cycle with a component search phase in the initial stages of design and there will be a large growing interest for this kind of software development with open standards and enabling reuse technology.

## 1.4 Scope of the Research

Everyone in the industry acknowledge the above mentioned advantages offered by software reuse. However, most of the companies do not invest money in reusing software. They mainly follow the traditional software development life cycles. The management in companies consider reuse effort as an additional investment without realizing that they offer long term benefits there by amortizing the overall cost over several projects. This is mainly a culture and attitude issue [dJ96]. On the other hand technology is another issue. From the above description on the enabling technology for developing reusable components and the growing interest for open architectures and standards, the industry has the substantial knowledge in building reusable components and overcoming the cultural and attitude barrier towards software reuse.

However, there is a limited research in how to represent and store the reusable components in a library of reusable components and retrieve them when needed. As mentioned in the above example of creating an attractive web page, there are thousands of freely available applets on the web and in order to choose the right ones, there should be a systematic way of storing and retrieving these applets.

Software Engineers have used several reuse techniques. They have used statistical and schematic techniques on informal textual documents (described in Chapter 2). They also have looked into defining formal means of representing the components using specification languages and performing semantic analysis for searching the components matching the given requirements. While one method offers ease of use the

other offers better classification and relevancy of search.

Also, earlier the main focus was on code reuse. However, over the years it has been realized that the experience gained in the projects during various stages of software development could also be reused. Capturing design rationale and intent is key to the success of reuse of higher level design concepts [PM94]. And with the component technology, once the right functional units are identified, it is just a matter of integrating them to build the required system.

Glass box type reuse requires understanding the design rationale of the existing components in order to be able to adapt the existing design. Design patterns (Appendix A) which are conceptual design abstractions can act as building blocks to capture design rationale [Vad96]. On the other hand, black box type reuse requires no such understanding of the underlying design decisions and implementation details. Only knowing the functionality offered by the component may be required. The developer expresses the functionality as the intent for developing. However, from the view of end user searching for reusable components, the intent is expressed as a set of functional features required by the system. Black box reuse is suitable for building large systems that could be developed by integrating existing reusable software components built using component object models. This is because each of the well defined individual component abstracts certain functional features and in order to use these components it is only required to know the functionality that they offer. The implementation details are not necessary to integrate these components.

Automating the means of capturing the intent of use of a software component and using the appropriate components when needed has been the focus of the research. A hybrid approach of using information from both the informal specification documents and the structured design documents to extract the component features is proposed in this research. The drawbacks of using only either kind of information and how they are complementary are described. The proposed hybrid approach exploits this complementary nature.

The above examples in Section 1.3 suggests that large software systems could be built by integrating well defined software components. This requires identifying minimal set of components that could provide as much functionality as possible. Hence, instead of retrieving individual components for reuse, set cover algorithm is used to retrieve multiple components for composition based software development.

## 1.5 Roadmap to the rest of the Thesis

This chapter has introduced the growing interest for reuse, it's advantages, existing enabling technology to develop reusable components and scenarios for developing large software systems by composing several software components. It also mentioned that design rationale and intent are required for making use of existing designs and described the scope of the research which is mainly to automate the capture of design intent from the documents and retrieving multiple components for composition based software development. The remaining chapters review some of the research in this area, requirements analysis for the developed reuse model, conceptual reuse model, details of the system developed, couple of examples, conclusions and future research. The appendices complement the thesis by providing some technical details which help as a quick reference.

The chapters are organized as follows

Chapter 2 presents the related research. Models using informal design documents as well as formal documents are presented and their advantages and disadvantages are studied.

Chapter 3 gives the requirements analysis for the suggested framework and the developed software system.

Chapter 4 presents the hybrid approach of information reuse for capturing design intent and the use of setcover algorithm for component retrieval for composition based software development



Chapter 5 explains the software system developed in the research, the underlying design decisions and the organization of the framework.

Chapter 6 explains the intended use of the software system with a couple of examples.

Chapter 7 concludes the thesis by summarizing some of the ideas developed and understood in the research. Also, direction for the future research is presented.

Appendix A is a technical note on Object Oriented Paradigm and Design Patterns.

Appendix B is a technical note on COM and JavaBeans.

Appendix C is a technical note on CORBA.

Appendix D is a technical note on Reggia's Generalized Set Cover Algorithm.

# Chapter 2

## Background

*Learn from the Past, plan for the Future*

---

---

It has long been realized that reusability of software could be achieved by building systems from composing the existing software components. Various means of achieving software reusability and the experiences and quantitative measures are present in [BP89, PDF93, Sar96]. Efforts have been made to provide better programming and specification languages that can support high levels of abstraction, interface between modules and templates. These methods help in writing reusable components. However, the ability to reuse software components does not stop with the ability to implement them. Selecting the appropriate software components for reusing them in a given context and automating this process still remains as a challenging problem. The main issues involved are pertaining to the representation of a reusable software component in a catalog, the user interface for identifying the required components and the retrieval mechanisms.

Components have been represented from simple forms like a set of keyword tuples to complex forms like parameterized modules as described in Section 2.1 and 2.2. In this case, the representation details were obtained from informal textual specification

documents or from structured, detailed design documents. At the time of searching for a component, the required functionality is specified as a simple text expressed in natural language or by representing it in a specification language.

This chapter reviews several earlier research attempts which adopted some of the above mentioned methods for representing components and obtaining information from the user. These approaches are classified broadly as *Semantic approach* and *Statistical Approach*. The definition of each of these approaches and their advantages and disadvantages are presented.

## 2.1 Semantic Approach

Use of specification languages, module interface languages, logical programming languages and compiler technologies is considered under this approach. A couple of examples are presented below and how they are difficult to use are mentioned.

### 2.1.1 Parameterized Programming, OBJ Language

OBJ is a language designed for Parameterized Programming [Gog89]. OBJ has four kinds of entity at its top level: objects, theories, views and reductions [Gog89]. A *theory* defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter for meaningful instantiation. A *view* expresses that a certain module satisfies a certain theory in a certain way; that is a *view* describes a binding of an actual parameter to a requirement theory. Instantiation of a parameterized module with an actual parameter, using a particular *view*, yields a new module. Figure 2-1 is an example code written using OBJ [Gog89].

LILEANNA is another parameterized programming language which has similar concepts of theories and views [Tra93]. This is a good example of a mechanism for writing reusable software since it does not commit the list implementation to a particular object type. However, this does not address the issue of identifying the

```

obj LIST[X :: TRIV] is
  sorts List NeList .
  subsorts Elt < NeList < List .
  op _ _ : List List -> List [assoc id: nil] .
  op _ _ : NeList List -> List [assoc] .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  op empty?_ : List -> Bool .
  var X : Elt .
  var L : List .
  eq head X L = X .
  eq tail X L = L .
  eq empty? L = L == nil .
endo

```

Figure 2-1: Parameterized List object representation using OBJ Language

right parameterized module suitable to implement the required functionality.

### 2.1.2 Structured Algebraic Specification, Clear Language

Clear is a language is designed for Structured Algebraic Specification [Tra93]. From their experience on Raytheon project, the others have identified the problem of designing an Ada-based software system, for maximum reusability of its component modules within different systems has been identified [LM89]. The goal of Raytheon project was to design a modeling system having a catalog of reusable Ada modeling components and a means of connecting them into complete models. The authors realized that a central part of the problem of designing the catalog was the problem of rigorously specifying the allowable uses of each component, that is, specifying the class of contexts into which a given component could meaningfully fit, and the kinds of components that could fit within a given context.

Clear is a language for formal, well-structured specification of software components [LM89]. It provides formalisms for expression of *algebraic theories*, operators

for building new theories from combination of old theories, and the definition of *theory morphisms*, which implement mappings between algebraic theories defined in terms of theory-building expressions. The semantics of Clear are formally defined using category theory. The authors believe "that much of this mathematics can be hidden by a user-friendly interface to the library manager and by automated assistance in mathematical reasoning". Figure 2-2 shows an example of how to specify a package that represents a group with equivalence relation on the data in a pseudo-Clear notation.

```

procedure E_CLASS_PACKAGE (Element : Group_with_Equiv_Rel) =
  Element enriched by
    data stores E_Class:
      opns *_ : E_Class, E_Class = > E_Class;
      axioms axioms for equivalence classes ;
  enden

```

Figure 2-2: Pseudo-Clear code for Group with Equivalence Relation

### 2.1.3 Partially Interpreted Schemas, PARIS

A *partially interpreted schema* is a program in which some parts remain abstract or undefined [KRT89]. These abstract entities can include both program sections and non-program entities such as functions, domains, or variables. For different interpretations of abstract entities in the schema, the results will be different programs performing different functions. PARIS has *manual matching* mode (also called shopping list mode) and *automated matching* mode. It uses Boyer-Moore theorem prover to carry out the verification. Figure 2-3 shows the schema for Linked List Insertion.

The following research directions that these authors mentioned are interesting

- Adding more schemas into the library, for a variety of computational models.
- Classifying all schemas within the library to increase searching efficiency.

```

Entity List    function f
function leq
function equal
domain D1, D1
variable data : D1
variable newdata : D1
structure node : D1 × pointer(node)
programsec S1
Applicability Conditions
f : -> D2
leq : D2 × D2 -> boolean
equal: D1 × D1 -> boolean
For all n : n in node : n -> next ≠ NULL =>
    leq(f(n -> data), f(n -> next) -> data))
Result Assertions ...
Section Conditions ...
Schema Body
struct node { struct node *next; D1 data; }
insert(listhead, newdata)
struct node *listhead;
D1 newdata;
{
struct node *p, *q, *new;
new = malloc(sizeof (struct node));
/* assign newdata to new->data */
S1
if(listhead == NULL)
{
listhead = new;
:
}

```

Figure 2-3: Schema Representation of Linked List iteration using PARIS

- Defining additional keywords in the system's vocabulary.
- Developing a user-friendly interface to add more convenience for the user's formulation of a problem statement and to provide more interactive communication during the process of matching and verification. In particular, when the theorem prover fails to prove one of the two implications needed for a successful matching, the user should not have to be an expert in automatic theorem proving in order to understand the reason for the failure.
- Augmenting the theorem prover with facts in the problem domain of the schemas, especially with facts about temporal logic assertions.

#### 2.1.4 Object Oriented Module Interconnection Language, MIL

Module interconnection languages (MILs) were introduced in 1976 by DeRemer and Kron for "programming-in-the-large" [HW93]. Subsequently MILs have found importance in software reuse, as the means of interconnecting components. MILs permit the description of components (or more generally "resources"), and the independent description of their interconnection [HW93]. The specification explicitly identifies not just the methods it *provides* but also the methods it *requires*. Object specification is a template which has *provides* and *requires* while Object implementation is a template which has *contains*, *external connections* and *internal connections*. There are other usual constructs like variables and methods. Figure 2-4 gives an example of describing a home-heating system.

#### 2.1.5 Summary of Semantic Approach

From the above mentioned cases, for adopting semantic approaches to reusability the user has to be an expert in using the system, have to learn a new language and map

```

object implementation home-heating
contains
  control-clock; clock;
  temp-controller; controller;
  temp-guage; thermometer;
  space-heater; heater;
external connections
  home-heating.read-temperature-setting =
    temp-controller.read-control-value;
  home-heating.set-temperature-setting =
    temp-controller.set-control-value;
  home-heating.set-beat = control-clock.set-beat;
internal connections
  control-clock.trigger = temp-controller.clock;
  temp-controller.read = temp-guage.read-temp;
  temp-controller.on = space-heater.on;
  temp-controller.off = space-heater.off;
end home-heating

```

Figure 2-4: Home-heating System module specified using MIL

the functional requirements into these language constructs. While the final retrieved components are more closer to what the user wants, there is a major burden on the user to learn a new specification language and represent the requirements using the specification language. This type of systems could not be useful especially for preliminary design. However, it should be noted that MIL languages which emphasize on the interconnection of modules rather than the specification of individual modules are still very useful once the user identifies the appropriate reusable components from the preliminary search.

## 2.2 Statistical Approach

Use of classification schemes based on syntactic clustering, statistical correlations is considered under this approach. It should be noted that semantic relations are



also used in this approach to representing a component. However, these relations are derived from the informal textual description while it is expressed using formal languages in the above approval. A couple of examples are presented below and how they provide poor performance is mentioned.

### 2.2.1 Faceted Classification Scheme

A *classification scheme* is a tool for the production of systematic order based on a controlled and structured index vocabulary [PD89]. This index vocabulary is called the *classification schedule*. Classification schemes can be either enumerative or faceted. Faceted schemes were used in describing the software components. Software components can be described by (1) the function they perform, (2) the way they perform it, and (3) their implementation details [PD89]. Under this assumption, the components were represented as tuples describing the functionality and environment. Functionality is represented as a  $\langle \textit{function}, \textit{object}, \textit{medium} \rangle$ . Classifying a component consists of selecting the sextuple that best describes the component. Some examples are shown in Figure 2-5

$\langle \textit{add}, \textit{integers}, \textit{array}, \textit{matrix} - \textit{inverter}, \textit{modeling}, \textit{aircraft} - \textit{manufacture} \rangle$ $\langle \textit{compress}, \textit{files}, \textit{disk}, \textit{file} - \textit{handler}, \textit{DB} - \textit{management}, \textit{catalog} - \textit{sales} \rangle$ $\langle \textit{compare}, \textit{descriptors}, \textit{stack}, \textit{assembler}, \textit{programming}, \textit{software} - \textit{stop} \rangle$
--

Figure 2-5: Representation of Software Components using Faceted Classifying Scheme

The end user interface provided by this system is interesting. The user has to enter a sextuple as above. Generalization can be achieved by making any of the six attributes as wildcards. An example is shown in Figure 2-6

While the user could use the generalization technique to browse and refine his/her query, query expansion technique uses conceptual distances from the conceptual graph to provide queries of closely related terms. However, this could be automated by synonyms substitution. More importantly, the expressive power of the user to describe

```
substitute/backspaces/file/text-formatter/program-development/*  
substitute/backspaces/file/text-formatter/*/*  
substitute/*/*/*/*/*
```

Figure 2-6: Search generalization using wildcards

the required system is limited to six attributes that could only describe smaller software components.

However, providing the user interface in terms of informally describing the functional requirements of the systems and not as mappings using specification languages makes the system easier to use. But the relevancy of the retrieved components might be lower.

## 2.2.2 Conceptual Schemas

Reusable conceptual components are defined as Generic Conceptual Units with associated Meta-Conceptual Units, which provide guidelines for reuse in a given application [CA93]. Conceptual schemas are properly defined according to a selected model such as Entity-Relationship (E-R) model, or Object-Oriented (O-O) models. The authors used a meta-model whose meta-constructs allow the definition of the constructs of both the E-R and O-O models. The main meta-constructs used were *Conceptual Unit* (CU), *Structural Property* (SP), *Behavioral Property* (BP) and *Dependency* (D) [CA93]. In [CA93] these meta-constructs are defined as

**Conceptual Unit (CU)** it allows the definition of constructs used to describe objects of the real world within the schema (e.g., entity, object class).

**Structural Property (SP)** it allows the definition of constructs used to describe a static feature of an object in a schema. SPs are expressed by means of attributes in both E-R and O-O models.

**Behavioral Property (BP)** it allows the definition of constructs used to describe the behavior of an object in a schema. SPs are expressed by means of attributes in both E-R and O-O models.

**Dependency (D)** it allows the definition of constructs used to describe relationships between two or more CUs in a schema.

A schema appears to be a kind of document suitable for automatic indexing: it is structured at both syntactic and semantic levels, and even belongs to a well identifiable domain. Schemas in the Library are grouped and classified with respect to the domain they belong to. For schema indexing, each schema  $S$  is associated to a set of Schema Descriptors (SDs), extracted from the schema itself, according to the criteria that privilege their capability of representing the schema subject. Only the labels of the Conceptual Units (CUs) are used as schema descriptors. The procedure for extracting SDs from a given schema  $S$  is composed of the following steps:

- assignment of a weight  $W$  to each CU of  $S$ ;
- definition of a threshold for selecting SDs from the weighted CUs.

A sample Schema-card [CA93] is shown in Figure 2-7

Schema name: $\langle string \rangle$ Application name: $\langle string \rangle$ Domain name: $\langle string \rangle$ Weighted Schema Descriptors: $\langle listof(SDs, weight) \rangle$ Total No. of terms : $\langle number \rangle$ Model : $\langle typeofmodel \rangle$ Schema code: $\langle schemaidentifier \rangle$ References: $\langle textualdescription \rangle$
---

Figure 2-7: A sample Conceptual Schema Card

Similarity Coefficient, Structural and Behavioral Affinity and Hierarchy Affinity [CA93] are calculated by examining the SPs and BPs of CUs and used in the retrieval

process. The success of the above approach depends on assigning the appropriate weights to the conceptual units.

### 2.2.3 Summary of Statistical Approach

From the above mentioned cases, for using Statistical Approaches, the user need not be an expert in a new language and not much work is required for searching for appropriate components. However, this approach results in poor classification and retrieval. The performance depends on the appropriate statistical weights used. Hence, this approach could be used mainly during the preliminary design stage to select limit the search space. Since, the search results may not be highly relevant, the user than has to go through them manually.

## 2.3 Hybrid Approach

Both the Semantic and Statistical approaches have their own advantages and disadvantages. While one provides more relevant solutions the other has some noise. While one requires substantial manual effort and technical expertise the other requires lesser manual effort and technical expertise. However, the interesting fact is that these approaches are complementary in nature.

During the preliminary design stage it is better to use the Statistical classification and retrieval approach to limit the search space. Also, with the use of component models which provide easy integration, the need for module interconnection languages and algebraic specification languages reduces. Once the user identifies the right components offering the required functionality, they could be easily integrated using suitable editors like the JavaBean's BeanBox tester [JBn97]. Also, as mentioned in the Chapter 1, creating a fancy webpage using several individual applets does not require integrating them at code level. As long as the applets with required functionality are identified it is possible to make the webpage. Hence, using open architectures

and component object models, the focus shifts from spending time to integrate the individual components to identifying the right functional components. Coming up with better ways of using statistical classification approaches there by reducing the noise is the primary focus of the research.

Capturing and utilizing the Design Rationale and Intent is the means of reusing design at conceptual level [PM94]. The intent of the design could be best viewed from the end users as the features offered by the artifacts. How to extract these features from the software design documents is the primary concern. While it is possible to use either the semantic or the statistical method for this, for the above mentioned complementary nature of both these methods, a hybrid approach can be used which analyses both the informal textual specification documents and formal structured design documents. The user interface is similar to those provided in statistical approaches since they are easier to use and less effort is needed from the end user. However, the noise in retrieval is reduced by using the information from the analysis on the formal structured design documents. All these details are explained in the next three chapters.

## **2.4 Design Recommendation and Intent Model Extended for Reusability (DRIMER)**

DRIM is a model developed for capturing the design rationale and intent of the product development [PM94]. Figure 2-8 shows the DRIM model. DRIMER (DRIM Extended for Reusability) is an extended DRIM model for reuse of design concepts augmenting design patterns with design rationale [Vad96]. DRIMER model assumed the use of Case Based Reasoning (CBR). However, as mentions in [MS93], "empirical studies indicate that these reuse tasks are difficult, even for experienced software engineers". Also, it mentioned "Most software reuse research has ignored the role of the software engineer. However, software engineers tend to be better reasoners

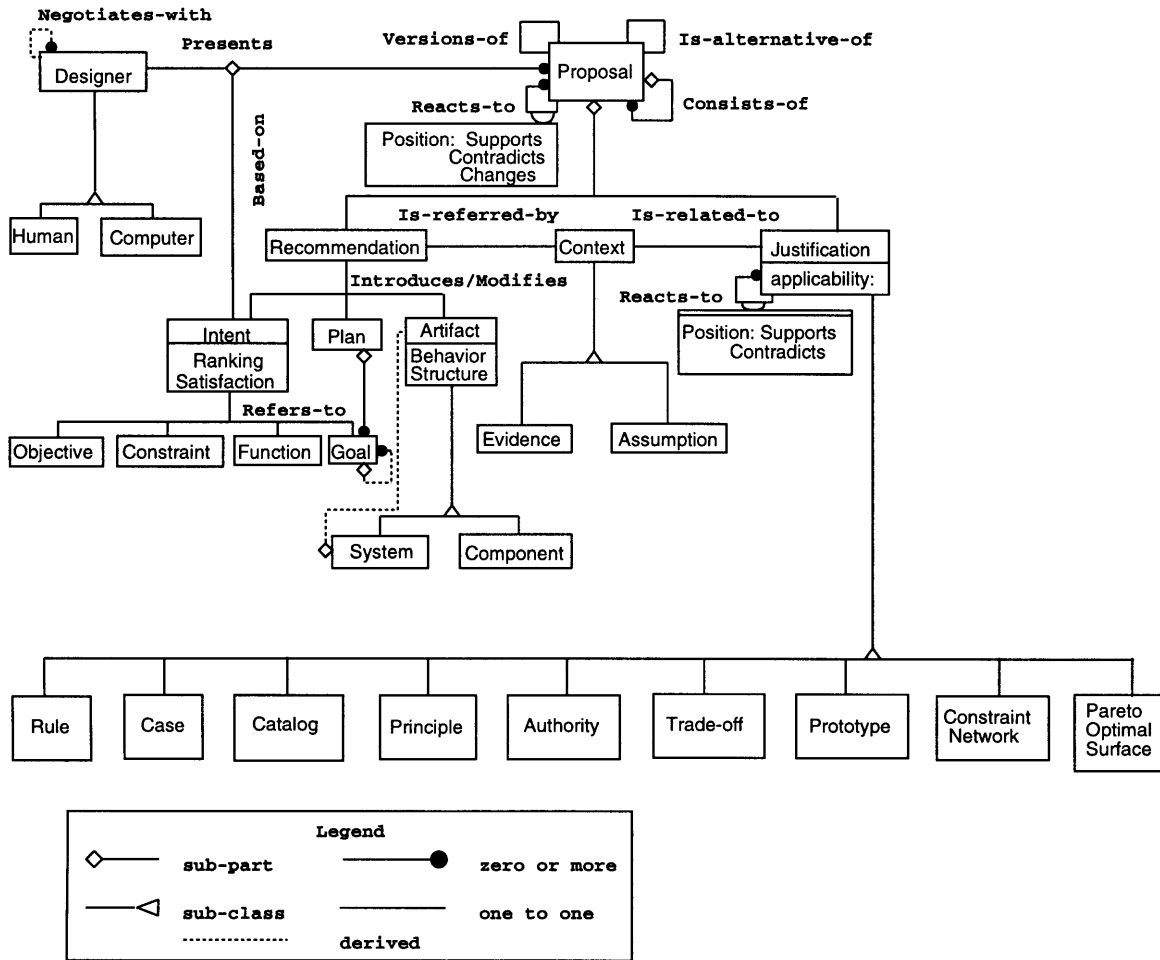


Figure 2-8: DRIM model adapted from [FPn94]

and have more experiences to recall than tool-based reuse mechanisms”. Hence, with the present technology the ability for tools to automatically and efficiently adapt the software components to provide the required functionality is difficult. Also, most of the CBR Systems like Caspian [Pri97] require a set of indices so that they could retrieve partial matches or similar cases based on these indices. In Caspian, if these indices are numbers then the degree of matching is based on interpolation of the numerical values. Non numeric indices are usually a set of values belonging to a class (e.g., FileInputStream and DataInputStream defined in the class InputStream) and the rank of matching varies based on perfect matching to belonging to the same

class. A software component cannot be however described using a few numeric and non-numeric attributes.

Figure 2-9 shows the web based system of DRIMER. DRIMER has a web based

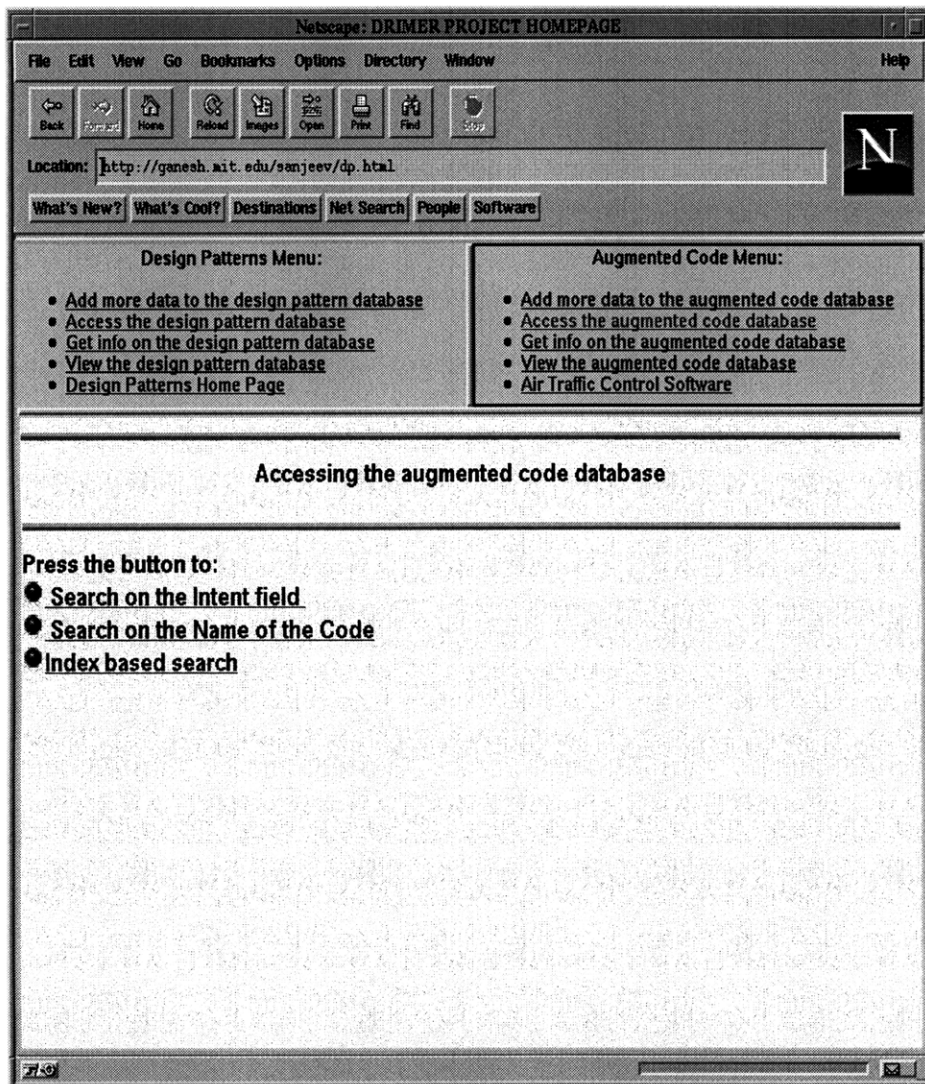


Figure 2-9: DRIMER web based system *adapted from [Vad96]*

client interface which provides facility to query for components based on intent field, name of the code and also keywords serving as indices. It is also possible to retrieve design patterns using queries based on similar fields. There are a few shortcomings with this approach and also the software framework. They are

- While components could be retrieved based on keywords describing their functionality similar retrieval mechanisms for design patterns may have limited applicability. This is because, from the requirements of a new product it is possible to identify the features that are required but it is not easy to find out whether or not a design pattern could be used unless one knows the pattern in the first place. Hence, searching for design patterns based on names and even intent is of limited applicability, the system provides as a database for components. Tools that could recognize the need for a specific design pattern based on the functional description of the software are needed.
- It is only possible to retrieve individual components based on the search fields. However, if the required functionality of the system is large it requires composition of several smaller components. This composition is not considered.
- CGI scripts and HTML forms are used for processing and user interface respectively. Using forms limits the user interaction capabilities. Even if sophisticated user interface is built using just the forms, the interface loses the intuition and ease of use.

This research focuses on capturing the design intent of software component and also addresses the last two shortcomings mentioned above.



# Chapter 3

## Requirements Analysis

*Besides other factors, incomplete understanding of the System results in Versions*

---

---

Two most important factors considered in designing the reuse model are the cost of implementing an effective reuse technology and the ease of use of the technology. It is necessary to utilize as much of the already generated information during software projects as possible to minimize the cost and effort of reuse. To design a reuse technology that minimizes the cost of implementation it is necessary to understand and classify the kind of information generated during the software design projects. Classification of information based on availability and formality are proposed in the research presented by this thesis. A case study of obtaining required information from a library maintaining a catalog is presented. This is useful in coming up with the representation for a reusable software component. This case study also helps in understanding the kind of support one would expect from a tool during the preliminary design stage. Retrieving multiple components whose composition gives as much of the required functionality as possible is required to increase the search relevancy.

As mentioned in Chapter 1, most of the application software in the commercial world is being built by developing a customized domain software on top of a hor-

izational framework. This results in reusing the base framework for several domain applications. The same could be done even for the software reuse tools. Instead of developing a specific reuse framework for each individual domain like finance, manufacturing and medicine, a general framework based on the above requirements analysis is developed and this could be customized to support reuse of software components of each individual domain.

### 3.1 Information in Software Projects

The following are some of the various kinds of documents generated during the software development life cycle

- User's Requirement Specification (URS)
- System's Requirement Specification (SRS)
- High Level Design documents (HLD)
- Low Level Design documents (LLD)
- Test Plans
- Programs
- Programmer's/Reference Manual
- User's Manual

The research presented in this thesis proposes a classification of the above information in order to be able to develop a cost effective and efficient reuse model. This information is classified as *available* and *inherent* information and, *semantic* and *textual* information. This classification helps in studying the means of capturing the design rationale and intent of software projects.

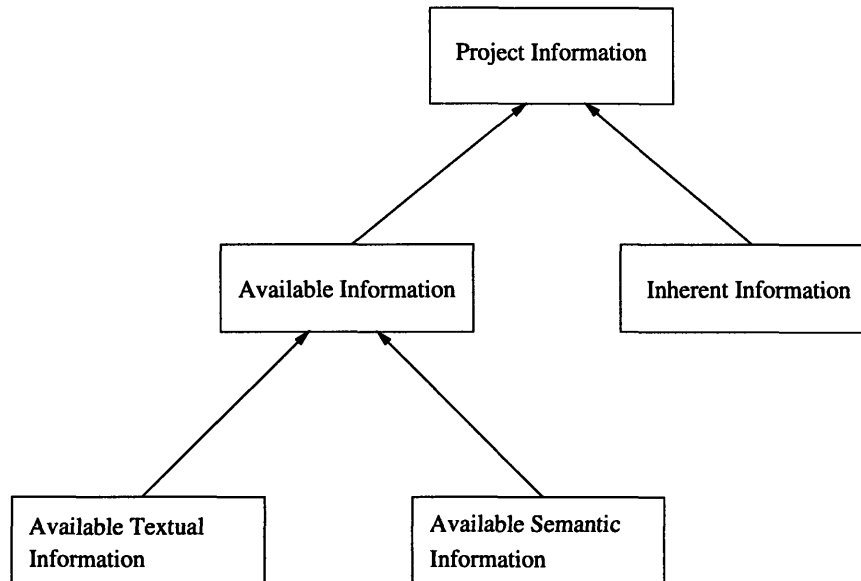


Figure 3-1: Classification of Project Information

### 3.1.1 Classification of Information used in Software Projects

All the above mentioned documents contain various kinds of information related to the software product. The entire information that is used in the software development process is classified in two different ways which is presented below. One is based on the availability of the information at the end of the project and the other is based on the formality of the information. Figure 3-1 shows the information classification.

#### Classification based on Availability

Information used in a project may or may not be documented and this classification is based on documentation of information.

**Available Information:** This is the information that is available in some document or the other at the end of the project. This information is produced as a part of the need for the project, i.e., this information is required to be documented to implement the project without concerning its reuse. Making use of available information for preparing reusable information reduces the cost and effort. One of the issues of

software reuse is the extra cost required to introduce it in the existing software life cycle models. Hence, if it is possible to design the reuse techniques based on the available information itself then such a technique could easily be integrated into the existing software development practices.

**Inherent Information:** This is the information that is not available in any of the documents at the end of the projects. This is because this information is not documented as a part of the project and remains in the minds of the people involved in the project. This type of information is mainly related to the justification of the design decisions (the results of the decision are usually available in the *available information*) and design rationale. It is the information used in obtaining the *available information*. Inherent information is very useful because of its *meta* information nature which helps in choosing one design over the other. But often this just remains in the minds of the people who participate in the project and seldom gets recorded. Making inherent information reusable requires additional cost of documenting such information. At the end of the project each person participated in the project (irrespective of his/her designation) could be asked to spend a couple days on documenting such inherent information. This could add to the overall cost of the present project. However, over a period of time this will reduce the cost of any new project.

Available information alone is not enough to define design rationale. Inherent information is also required. For example, inspite of identifying the existing design patterns in the design, it is also necessary to have information related to these patterns which helps in understanding the reason for choosing the particular design possibly among several alternatives. However, most of the design intent could be obtained from the available information. This information will be mostly available in the textual specification documents and user's manuals. Black box type reuse requires mainly classifying and retrieving components based on the functionality they offer (Chapter 1) and hence design intent which could be obtained from the existing documents could

be used for classification and retrieval. Glass box reuse on the other hand requires design rationale (Chapter 1) and hence the need for the inherent information. This type of reuse is possible only if there is an initial investment towards reuse.

### **Classification based on Formality**

Information in the software projects might be represented using a formal or natural language. This another classification can be based on formality of information.

**Semantic Information:** Information that is represented using formal specification language, mathematical formulae or design notations (like OMT or UML) falls under this category. Programs contain only semantic information. Also, HLD, LLD, Test Plans which are documents generated during the design contain mostly semantic information. Programmer's Manuals also contain some semantic information. This kind of information is mainly useful for semantic analysis described in chapter 2. However, most of the semantic analysis techniques propose a separate specification language making it necessary to express the reusable components in this language. Also, expressing the required functionality in the specification language has to be manually performed at the time of search for the components. Hence, both maintaining and searching for component results in additional activities apart from the normal activities in the software life cycles. It would be useful to develop tools that could use the existing semantic information itself eliminating the need to express the components in a new specification language.

**Textual Information:** Information that is represented using natural language falls under this category. URS, SRS and User's Manuals which are design documents generated during the the software development contain textual information. This kind of information is mainly useful for statistical analysis described in chapter 2. As mentioned in Chapter 2, since reuse tools developed using statistical analysis are easy to use, the aim of the research presented in this thesis is to represent reusable com-

ponents by extracting information from these documents and semantic documents. Semantic analysis on the semantic information is used to increase the classification accuracy and retrieval relevancy.

### 3.1.2 Observations

The following observations could be made from the above classification

- Minimizing the cost of implementation requires using only the *available information* as opposed to using inherent information which requires additional documentation specifically for reuse.
- Maximizing ease of use requires statistical analysis on *textual information* as opposed using semantic approach which requires learning new specification language.
- Maximizing the relevancy of search requires semantic analysis on the *available semantic information* as opposed to using statistical approach whose accuracy depends on assigning the right weights.

Hence, the reuse technology should be based on *available textual* and *available semantic* information and use both the statistical and semantic analysis. Semantic analysis on the *available semantic* information is performed only to increase the classification accuracy and search relevancy of the statistical analysis approach.

## 3.2 Information Search - A Case Study

This case study is mainly based on personal experience in searching for relevant research material. The user interface for the proposed reuse model is mainly motivated from the observations from this case study <sup>1</sup>.

---

<sup>1</sup>It should be noted that only the conventional online cataloging and search facilities are studied and recent technologies like digital imaging and related search technologies are not considered

### 3.2.1 Search for a TextBook/Journal in a Library

Searching for a reusable component has similarities with searching for a text book on a particular topic in a library. In a big university library like in MIT, with several floors and books covering different disciplines providing a facility to search for books is very important. Organization of the books and journals is crucial. Books are separated from Journals. Books belonging to a particular discipline are all placed at the same place. With in this structural layout, books related to the same specialization are placed together. Each book is given a call number. An online catalog is maintained to make it easier for people to find out where the books are located.

When a person wants to search for a book he/she might have different levels of information about the book he/she wants. If the call number of the book or the ISBN is known it is very easy to locate the book. Even if the author name or the exact name of the book is known it is possible to precisely locate the book. This is same with journals. If the volume number and the publication of the journal is known it is easy to locate the journal. But several times a student wants to find out all the related books and journals that are of interest to his/her research. In such a situation he/she will not have a particular book/journal in mind but he/she is only interested in books with his/her research topic.

Initially the user might search for books with a few keywords. This gives the location of a couple of books having these keywords in their title. Then the user can go to the appropriate stacks and look into the listed books and select those books that are most appropriate to him/her. Also, many times some of these selected books gives references to new books relevant to the interested topic. While this keyword search does not retrieve only the relevant books, it does help the research student to start with something that eventually leads to the required books/journals. However, ways of improving search results should be provided to the cataloging system. Once the user browses through these books and picks up a few then the user goes through them and further limit his/her selection. This shows that while in the initial stage

the automatic retrieval can ease the effort of selecting books, human judgment is necessary for the final books necessary for the research.

While search based on keywords in the title of the book alone might be sufficient the search can be improved if every book is represented with a small description of what the book contains, similar to the abstract of a paper. Using this description while searching will improve the relevancy of the retrieved books. However, proper search mechanisms that retrieve relevant information should be provided because keywords are likely to match for more books because of using larger text for keyword matching. Displaying the description along with the title and other details of the book will let the user to decide whether or not to read the book without actually going to the stacks and going through the book.

Sometimes the user might find only a single chapter in the book relevant to him/her. The system has the option of retrieving each relevant chapter from a book (or a paper from a journal) or just the book (or journal). The granularity of information units that are retrieved is, hence, another important factor. In a big university with too many books and journals it might not be appropriate to index on individual chapters and papers. Not only does this require a lot of effort for the catalog maintainers but also for the end users since they have to browse through several search results before they actually find what they require. The time required to search also becomes a limiting factor. This could defeat the purpose of the search unless the search is structured and guided semantically.

### 3.2.2 Observations

Following observations can be made from the above case study

- After filtering the totally irrelevant books/journals users would browse the retrieved list of books to find out precisely the books that are useful to them. Similarly *a designer can first filter irrelevant software components through automated search and then study the retrieved components to decide on using them.*



- From the preliminary stage of searching for books users find other relevant books/journals from the initial set of books they choose. The new books are referred in the selected books. This observation suggests that *a software component should not only be represented with its own design documents and code fragments but it should also contain references to other similar or related reusable components, if possible.*
- Giving a small description of the book along with the title and author will often save the user the effort of locating the book and going through it to decide on using it. Similarly, *reusable software components should have descriptions and comments by people who have reused them before.* This will be very helpful at the time of manual search which the user does by browsing through the retrieved components.
- *Granularity of the information unit directly relates to the human effort.* Indexing every chapter of the book is a major effort for the catalog maintainer. Also, the end user has to go through several retrieved results and the degree of relevancy will be poor. Granularity plays an important role for software components as well. Reusing smaller components offers lesser benefits because the cost of integrating the components could be more than the benefit of reusing them. However, smaller components have higher relevancy.
- Describing the book/journal and keyword search based on the description will increase the search relevancy. Search based on just the title might miss some useful books/journals. Similarly *a software component should be represented with a set of features that best describe its functionality.* Search will be based on these features.
- *Search should also be possible on fixed attributes like Call Number and ISBN.* People who have been reusing a set of components often or are referred to by

other members in the organization can access them easily by using these unique ids.

- *Search for relevant information is a process in which the search effort shifts from the computer to the human.* Automating the search for the initial part of the search saves significant amount of time and effort to the user. Then the user has to use his/her discretion to further limit the search. Only the degree of computer vs human effort changes with intelligent retrieval techniques but ultimately human judgment is necessary.

All the above observations should be taken into account while designing the representation of software component and also the search user interface.

### 3.3 Retrieving Multiple Components

A reusable software component could have a more than a single functional feature as explained in Section 3.2. The larger the component the more features it is likely to contain. Since there should be no restriction on the granularity of the reusable software component, it is assumed that the same component could have several functional features. Similarly, there could be several functional features required for a new software system. The same component could offer more than one of these features. As a result there could be several possible solutions each having multiple components and there should be a means of ranking these solutions. The next section describes the requirements for retrieval and ranking.

#### 3.3.1 Retrieving and Ranking

To begin the search, the user would have a set of functional features required by the system to be developed. The user could specify each of these required features separately and get a number of components satisfying that feature or specify all the

features and get sets of components each of which contain components that offer as much of the required functionality as possible. It is preferable to reuse the set of fewer components that could provide maximum functionality since it minimizes the effort of integrating the components. Hence, it is necessary to provide a search mechanism in which it is possible to specify all the required functional features and obtain a minimal set of components that offers as much of the functionality as possible. Other alternatives should also be provided because some of the components in the minimal set could be very large with several irrelevant features while the user is interested in developing a light weight component. Also, if there are no components that could offer some of the required features, then the components should be retrieved for some of the required features and the uncovered features should be presented back to the user. This helps the user in estimating which components need to be modified or what additional development effort is required.

If there are multiple solutions possible, the user should be presented with all the solutions and the solutions should be ranked on some criteria in helping the user to make decisions. One of the criteria could be ease of reuse. Composibility and Relevancy are some other criteria mentioned in Chapter 4.

### **3.4 Framework and Customization**

First the details of CORBA [ORB97] framework and how commercial applications are being built is described. Similar model is used in designing the reuse framework and hence it is useful to know how the CORBA applications are being developed.

Apart from specifying CORBA, a distributed computing architecture, OMG [OMG] also standardizes Object Services, Common Facilities and Application Objects. Services are necessary to construct any distributed application and are always independent of application domains [COS97]. A collection of services that many applications may share are developed as Common Facilities. These facilities are again divided into

two major categories; Horizontal Common Facilities, which are used by most systems, and Vertical Market Facilities, which are domain-specific. Application Objects are specific to particular commercial products or end user systems. Application Objects correspond to the traditional notion of applications and so they are not standardized by the OMG. Both Object Services and Horizontal Common Facilities could be used in building the Application Objects. Vertical Market Facilities help in providing the interoperability between various application objects of the same domain [CFA97]. As mentioned in Chapter 1, many consulting firms are also having a general framework which could be used in several applications built for different domains. Applications are built on top of the framework by customizing it for the specific domains.

Part of the component representation and search techniques can be common and not depending on the application domain. Also, black box type reuse which mainly depends on the design intent could be modeled as much as possible independent of the domain. However, using design rationale for glass box type reuse requires substantial domain knowledge and hence customization for individual domains.

Hence, it is preferable to design a reuse technology with a general framework and customizing it for individual domains using domain specific knowledge. The main focus of the research in this thesis is building the general framework for capturing and reusing the design intent.

### 3.5 Summary of Requirements

From the above observations, the following are the requirements identified in developing the model for design intent capture.

- The reuse technology should be based on the *available* information to minimize the cost of implementing the technology so that it could be seamlessly integrated with the existing software life cycle practices.

- To provide an easy end user interface and minimize the effort of preparing input. Understanding the required functionality is essential for all software development life cycles. However, expressing this functionality in a specification language is an additional burden on the designers. One need not be an expert in understanding theorem proving and logical reasoning. This will only hinder people from using tools based on such methods. It is, hence, necessary to have a simple user interface for identifying the reusable components in the preliminary design stages.
- To increase the accuracy of classification and relevancy of retrieval by using a hybrid approach mentioned in Section 2.3.
- Retrieving multiple components to suggest the possible composition to increase the search relevancy. Suggesting the least number of components offering the same functionality helps in reducing the effort of integrating the components.
- To provide a horizontal reuse framework. Reuse is fundamentally a concept independent of the domain. Hence, several reuse facilities could be common for all the applications and hence the reuse technology should be based on a general horizontal reuse framework. This framework can be further customized by providing a domain specific knowledge.

# Chapter 4

## Conceptual Model

*Perhaps, even the Universe is based on a Model*

---

---

This chapter presents the conceptual model on which the design intent capture framework and tools are developed. First the conceptual models related to reuse techniques based on either only *available textual* or *available formal* information are reviewed. Then the model using both kinds of *available* information is suggested. This is mainly based on *statistical analysis* on *available textual information* together with the semantic knowledge from *semantic analysis* on *available semantic information*. This has the benefits of using both types of available information; accuracy of classification and high relevancy of retrieved information and, ease of use. Two types of knowledge from *semantic analysis* that are used along with the *textual information* for *statistical analysis* are presented. One increases the classification accuracy while the other helps in identifying those features of a component that could easily be customized while using the rest of the framework.

The representation of software component is broken into two parts. One is part of the general framework which is presented in this chapter. The other is specific to the individual domain and could be customized. The component representation is pre-

sented from the end user's point of view. More sophisticated internal representations could be used and customized by providing the appropriate mapping.

Reggia's Generalized Set Cover [RNW85, RNWP85] algorithm and how it could be used to retrieve multiple components for composition based component reuse is presented.

## 4.1 Existing Reuse Models

As mentioned in Chapter 2, reuse techniques are mostly based either on textual information and statistical analysis or semantic information and semantic analysis. Several examples are presented in Chapter 2. These two types of information usage can be conceptually modeled as follows.

### 4.1.1 Reuse based on Available Textual Information

User's requirement specification (URS) document contains mainly the functionality needed by the final product. Some of the functionality may not be finally available in the product because it may not be feasible or not required. System's requirement specification (SRS) document gives any additional assumptions that need to be made in building the product. Since all this information is in textual form extracting the right component features requires natural language processing. One could also consider extracting features based on the statistical relevancy of the words used in these documents by using techniques like WAISindexing and clustering algorithms. However, this could still lead to a lot of redundancy and poor relevancy. Figure 4-1 gives the conceptual model of using textual information and statistical analysis for automating component reuse.

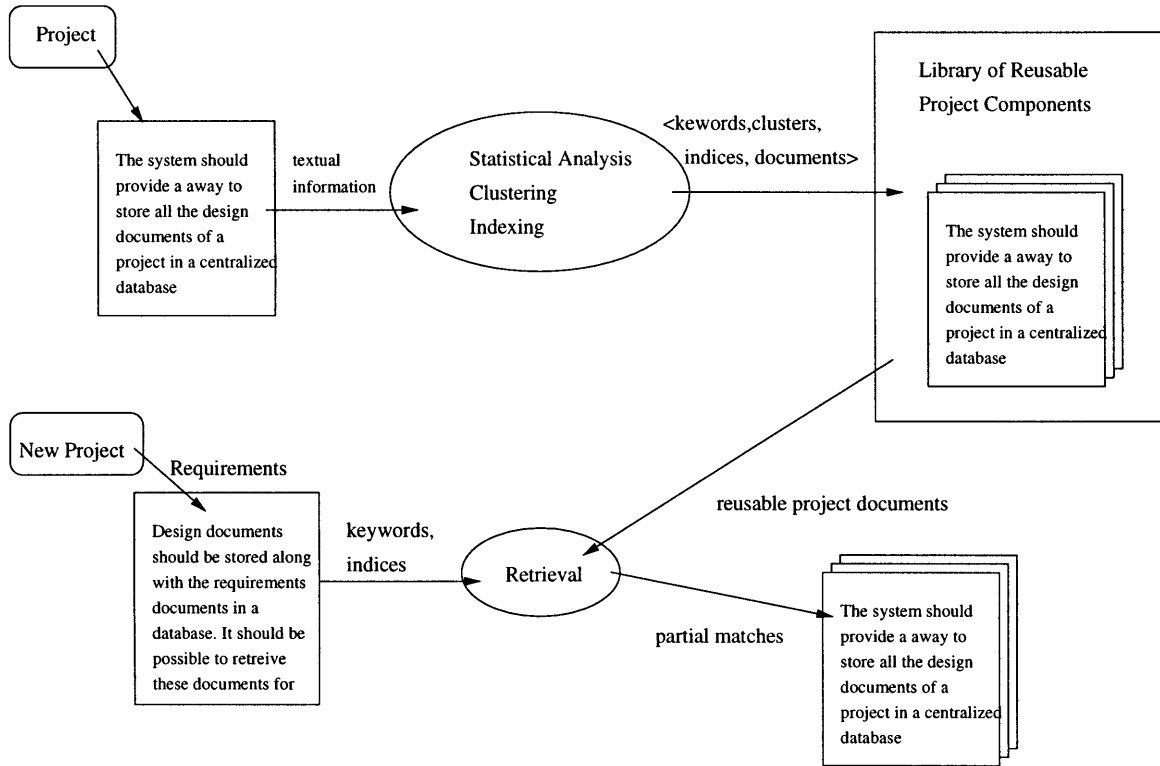


Figure 4-1: Reuse from Textual Information

### 4.1.2 Reuse based on Available Semantic Information

Reusable information can mainly be obtained from the high level design diagrams, low level algorithms, flow charts and to some extent from the programs themselves. Obtaining reusable information from these documents requires semantic analyses like data flow and control flow analysis. Design patterns provide a level of abstraction to work with when retrieving information from high level design diagrams. Obtaining reusable information from algorithms and programs may be very difficult if not impossible. Also, matching the description of a problem with the reusable information from these documents is very difficult and requires a lot of expertise in topics like theorem proving and logic reasoning as mentioned in Chapter 2. It also requires complex parametric type representation and matching, pre and post condition representation and satisfying as well as data and control flow analysis. Hence, in general for any



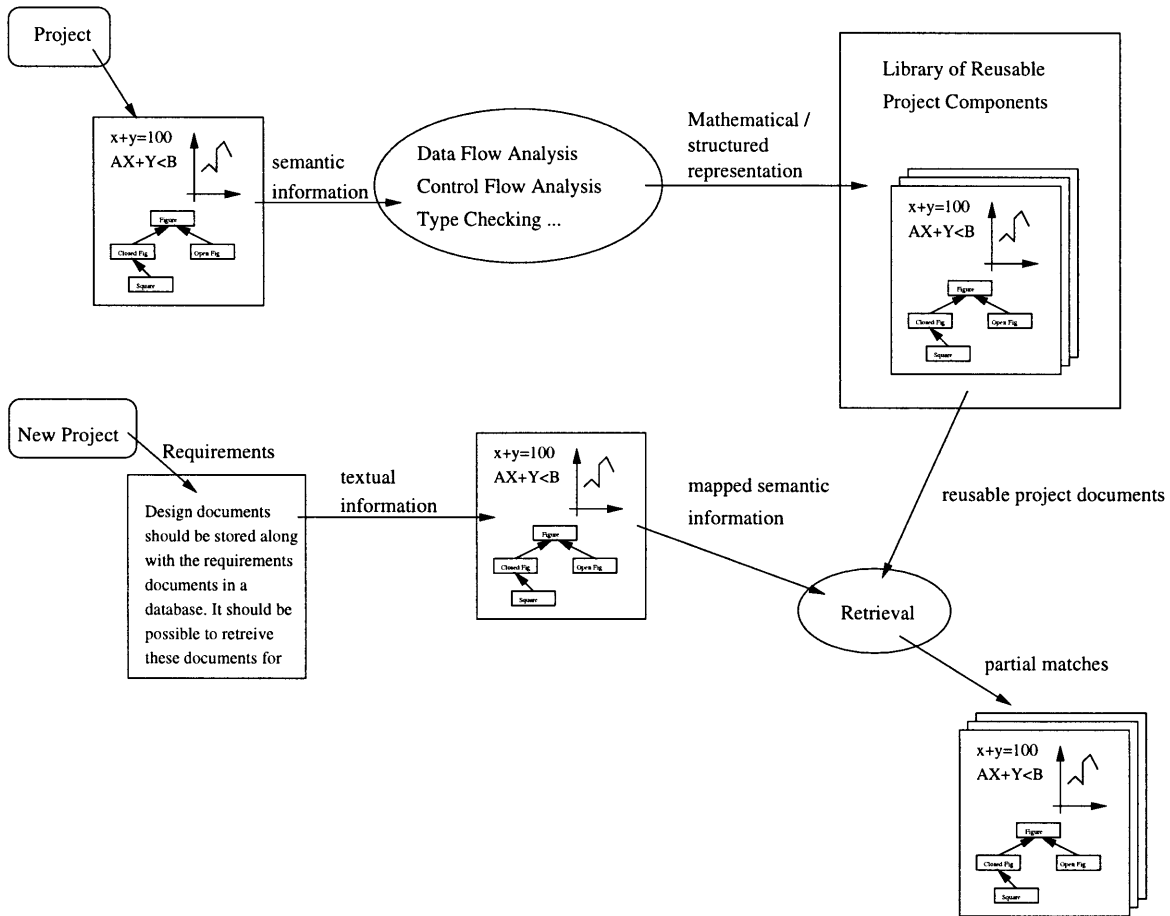


Figure 4-2: Reuse from Semantic Information

type of product, extracting and using the reusable information from semantic information is very difficult and some times may not be practical. Also, mapping a given problem from the informal description to a formal representation suitable as input to the retrieving agent has to be done manually and this might itself be a difficult and time consuming task defeating the purpose of reuse. Figure 4-2 gives the model of using semantic information for automating component reuse.

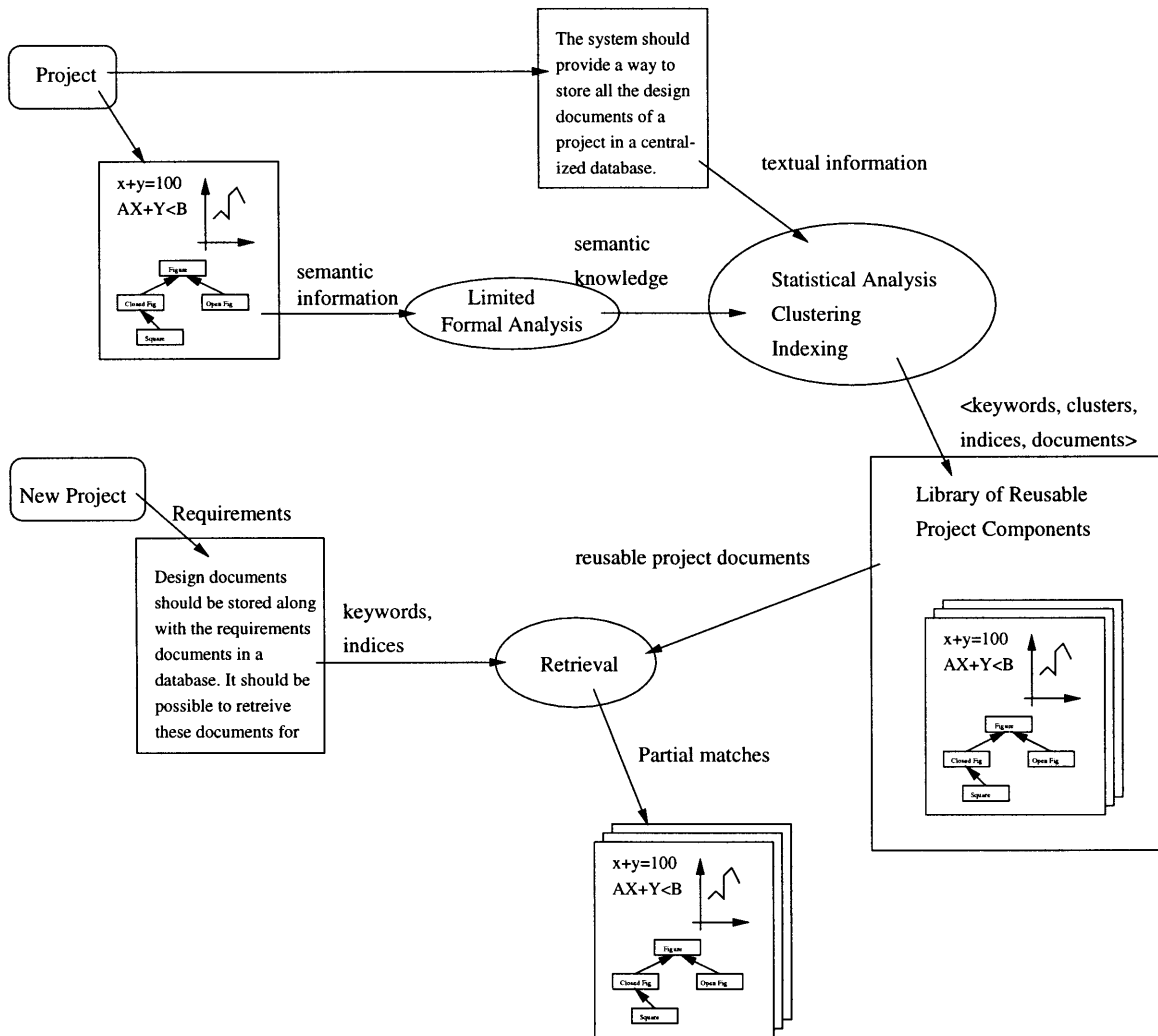


Figure 4-3: Reuse from both Semantic and Textual Information

## 4.2 New Reuse Model

### 4.2.1 Reuse based on both Textual and Semantic Information

From the above description it is clear that extracting and using reusable information from either available textual information or available semantic information requires various degree of effort and each offer various degree of relevancy and accuracy. However, these are complementary in nature. Hence, one could do limited semantic analy-

sis on available semantic information and use the derived semantic knowledge together with the available textual information for statistical analysis and get reusable information whose accuracy and relevancy is increased, if not complete. Figure 4-3 gives the model of using both semantic and textual information for automating component reuse.

The next two sections presents the means of improving the efficacy of statistical analysis on textual information by using the semantic knowledge derived from semantic analysis on semantic information.

### **4.3 Class Tree Provides Better Weights for Statistical Analysis**

As mentioned in Chapter 2, most of the reuse techniques based on textual information use some kind of statistical analysis. And these empirical methods use suitable weights based on certain assumptions. In [DA96], the authors propose a method of constructing design representations from Text analysis. Given any textual document of a design, the document is first parsed to get all the words omitting the most common words like the, is and are. Once a set of words are retrieved, statistical analysis is done on the words. The words are clustered and Bayesian networks are built. This representation is used in later retrieval of the relevant information.

It is assumed that the more frequent a word is, the more general it is in the domain of the problem and hence given lesser weightage because they do not serve as good discriminators of concepts. At the same time words with lower frequency should not be given too much of weightage because some long, infrequently used sophisticated words could be given higher weights even though they do not actually represent the details of the component. Coming up with the right weights for the words to serve as good discriminators of concepts is the main challenge in this approach.

Since a software artifact also has semantic information associated with it, it is

better to use some knowledge from this information to come up with appropriate weights for the words. One possible way of doing that is suggested in this section. Before explaining the semantic analysis and its use in textual analysis, it is necessary to understand the underlying assumptions. These assumptions are described below.

### 4.3.1 Assumptions

Suppose an application is built using a pure object oriented programming language like Java. The applications are designed such that higher level abstractions which represent the software system are built making use of lower level abstractions which represent the implementation details. It is assumed that class names are not arbitrary and are chosen based on the vocabulary of the application domain. Hence the words in the names of classes representing higher level abstractions are more likely to contain information of what the application is all about than those representing some lower level implementation details. Then it remains to identify automatically the higher level classes from the application. A formal means of achieving this is presented below.

### 4.3.2 Class Tree

An object oriented software component has a set of classes and these classes have various relations like *inheritance*, *uses* and *contains* among themselves. Assume that a Graph (V,E) where V, the vertices represent the classes in the design and E, the edges represent the *uses* or *contains* relation between any two classes. In Java, any application has a single class from which the execution of the program starts. Let this be the root class. Similarly, it is assumed that a component with a well defined interface has such a root class, say X. Starting from X and traversing the graph using breadth first search will give a tree with X as the root. For any component, it can be observed that the classes in the top few levels of this class tree represent high level abstractions directly related to the functionality of the component. On the

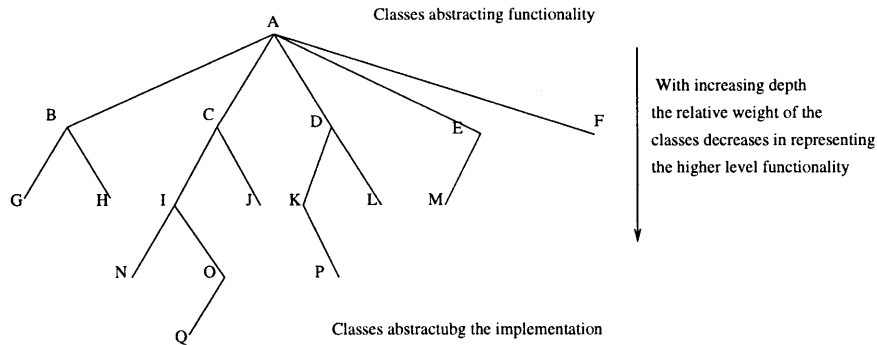


Figure 4-4: Class tree provides relative weights for the words

otherhand, classes down the tree are more related to the implementation details and serve as building blocks of the highlevel classes. Figure 4-4 shows such a Class Tree and an example is given in Chapter 6.

From the class tree the weights are derived such that the words in the class name at the root level has maximum weight and decrease for class names with increasing depth. Let the depth of a class in the Class Tree be  $d$ . Equation 4.1 is used to derive the relative weight,  $rw$ .

$$rw = 1/(1 + d) \tag{4.1}$$

Hence, the weights,  $W$ , of the words are modified as  $rw * W$ . It should be noted that the names of the classes present at the bottom of the Class Tree might not have the words that are present in the specification documents. This is because these classes might be more related to the implementation details related to algorithms and data structures. Even though these words have relative weights calculated from the above formula, their absolute weight will still remain zero. Suppose  $h$  be the maximum depth of a class in the Class Tree whose name has words present in the specification documents. The relative weight of all words present in the documents but not in the Class Tree is

$$rw = 1/(2 + h) \tag{4.2}$$

The relative weights computed using the above formulae are used to modify the weights obtained from statistical analysis.

## 4.4 Creational Patterns Suggest Abstractions of Functional Features

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. Creational patterns become important as systems evolve to depend more on object composition than class inheritance. There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together [GHJV95].

Generic components are reusable in more situations than the specific components. Good generic components are usually built using Design Patterns. Creational patterns are used to provide the flexibility of creating objects that could potentially change the behavior of the system based on the class they belong to, yet having the same interface. Objects that are created using Creational Patterns are good candidates for identifying the functional features of the components. Hence, for example, if *Factory Method* design pattern (Appendix A) is used to create objects, then the *base class* of these objects abstracts a functionality that could be modified by plugging in objects of a newly *derived class*.

While the program as a whole has a single class that has the *main* entry point, and hence the root class in the Class Tree, classes whose instances are created using Creational Patterns are by themselves likely to have high level abstractions. In this case, even though these classes occur at some depth,  $d$ , in the Class Tree, their relative weight,  $rw$ , should not be calculated directly using the formula given in the previous section. The relative weights of these classes and all other classes which are

in the tree rooted by these should be higher than other classes at the same depth. For example in Figure 4-4 if the instances of class D are created using Creational Design Patterns, then the relative weights of D, K, L and P should be more than the values calculated using the above formula. Suppose, the depth of the class of objects created using Creational Patterns is  $d'$ , then  $rw$  is increased by a factor of  $(1 + 1/d')$ . This means that with increase in depth, again, the factor decreases indicating that the flexibility in creating implementation objects does not provide much information about the functional features.

## 4.5 Software Component Representation

A reusable software component should be stored with all the related documents and the code fragments. While this entire information is presented to the user when a component is selected, a component should have a representation for the process of searching. Various models have been suggested varying from simple representation of attributed tuples and schemas to specifications using some high level languages. Each has its own advantages and disadvantages as mentioned Chapter 2. While the search agent can use any degree of formal representation, the user should be given a simple way of describing the required functionality. This is even more essential during the preliminary stages of the design. However, a suitable mapping should be adopted by the search agent to convert from the simple description provided by the end user to its complex internal representation. The designed reuse framework uses a suitable simple software component representation from the end user perspective to provide all the required functional features. The simple representation is presented and possible extensions for complex internal representations are suggested.

### 4.5.1 Simple External Representation

A fixed number of predetermined attributes cannot cover all the necessary features to classify software components from various domains. Hence a list of attribute-value pair alone cannot classify software components. Only certain properties like the language of implementation, the Operating Systems in which the component can be used, application domain and version number may be used as attributes. These attributes limit the use of the components based on factors external to the functionality of the components and are applicable for classifying all software components. One should be able to describe a software component by a set of features provided by the component along with a set of attribute-value pairs. The advantage with this approach is that it will not limit describing the reusable components only in terms of the predetermined attributes because they will not be usually sufficient to describe all the components.

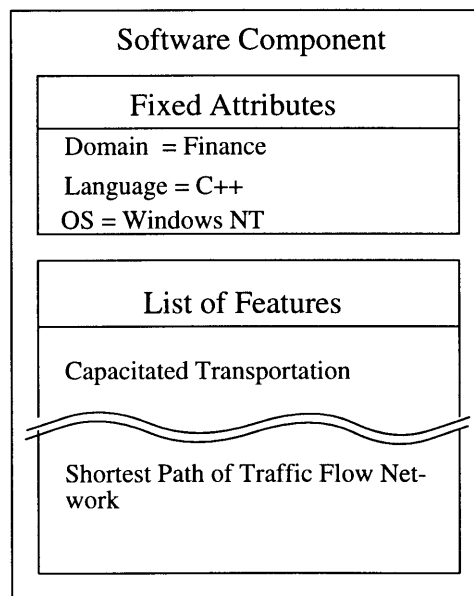


Figure 4-5: Software Component Representation

A given software component is represented as a fixed set of attribute-value pairs and a variable length set of features describing the software component, Figure 4-5. The attributes chosen are, as mentioned above, those that limit the use of a component



based on factors such as implementation language and Operating System. However, if a person wants to search for components irrespective of language, he/she will be able to do it by specifying it as a wild card. If the user is more interested in the design rather than the implementation, he might not be concerned about the language of implementation. Also, if the user is looking for CORBA/DCOM based components, language is not a concern. Similarly, if the user is interested in components written in pure Java, he/she may not need to specify the Operating System. In all these cases, search based on wild cards will be very useful.

Before putting the software component into the component library, all the relevant features of the components are to be automatically identified. This is done using the above mentioned analysis on both semantic and textual documents. Identifying the values of the attributes like language of implementation and version does not need any analysis. The component manager can easily obtain this information.

This representation of a software component makes it easier to get all the relevant components for reuse. The user can retrieve the relevant components by specifying the features required by the new component. This will be very useful at the early stages of analysis and design. If appropriate components are found, various documents stored in the library about that component can be used at different stages of the project. The user only has to have an understanding of the application to be developed and no mapping of this information into a semantic representation for retrieval of reusable components is necessary.

### **4.5.2 Complex Internal Representation**

Component features described in simple phrases will not match exactly with the existing feature descriptions. Several natural language processing techniques could be used for a better match. An *equals* operator is polymorphic which provides the degree of equality between two component features. A threshold could be used on this value and various mapping techniques could be used to improve the search. Synonym

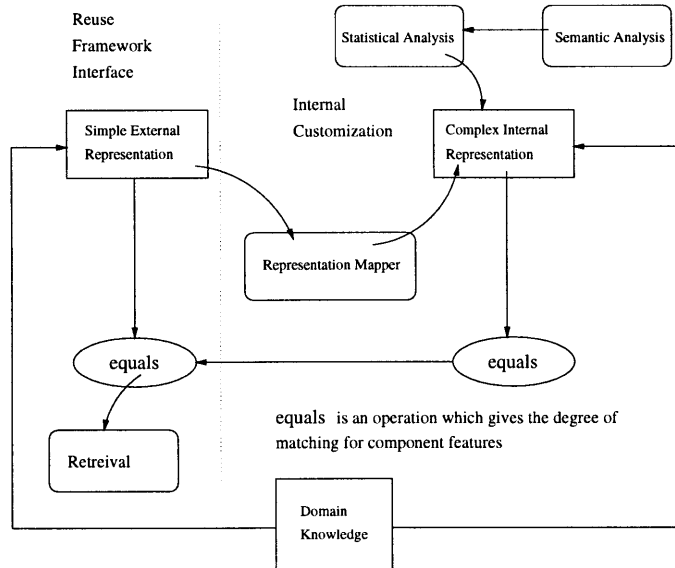


Figure 4-6: Customization of Component Representation

based extension is a simple technique that could be used to increase the matching [MV97].

Apart from the natural language processing techniques, some of the representations like the Conceptual Schemas [CA93] and Bayesian networks [DA96] could be used. Accuracy of classification for representing components using complex internal representations could be achieved by using the above mentioned hybrid approach of performing both semantic analysis and statistical analysis. Figure 4-6 shows how to customize the software component representation both for complexity and domain dependency.

## 4.6 Multiple Software Component Retrieval

Medium to large size applications will have several functionalities. However, reusable components are usually small with well defined interface and limited functionality. Queries with multiple functionalities without considering composition will result in the retrieval of several individual components each offering a part of the required

functionality. This results in poor degree of relevancy for each individual retrieved component because composition of the smaller components to get higher functionality is not considered. The user has to look at the various combinations of these reusable components. Integrating fewer components is easier and less error prone. Hence, one should try to find the minimum set of components in the library that can provide as much functionality as possible.

For example, suppose the user wants features  $f_1$ ,  $f_3$  and  $f_5$  ( Figure 4-7). If the composition of the components is not considered, the user will be presented with the components  $c_1$ ,  $c_2$ ,  $c_4$  and  $c_5$  and the degree of relevancy will be lower because the ratio of the number of features a component can cover to the total required features is low. However, if the composition of the components is considered to cover all the given features, the user will be presented with the solutions  $(c_1 c_4)$ ,  $(c_1 c_5)$ ,  $(c_2 c_4)$  and  $(c_2 c_5)$ . This will also give higher degree of relevancy. This gives the user an immediate idea of which components can be composed for the required features.

Sometimes, the user might be more interested in building a light weight system. Minimal set solution will not necessarily produce a light weight solution. This is because the components in the minimal set solution might have several features that are not required. In this case, a solution with more components but with fewer unrequired features would produce a light weight solution <sup>1</sup>. For example, suppose the user wants features  $f_2$ ,  $f_7$  and  $f_8$  ( Figure 4-7). While the minimal set solution would give  $(c_1 c_6)$  and  $(c_4 c_6)$ , the light weight solution would provide  $(c_1 c_7 c_8)$  and  $(c_4 c_7 c_8)$ . Again the solution  $(c_1 c_7 c_8)$  is lighter than  $(c_4 c_7 c_8)$  because the first solution has one unrequired feature while the other has two.

The efficiency of retrieval of software components can be increased by taking care of the composition of the individual components. This is mainly motivated from a well known problem in AI; identifying the set of all diseases which can cover a given set of

---

<sup>1</sup>The user can always remove the extra functionality from the components but that requires understanding the implementation breaking the black box reuse model.

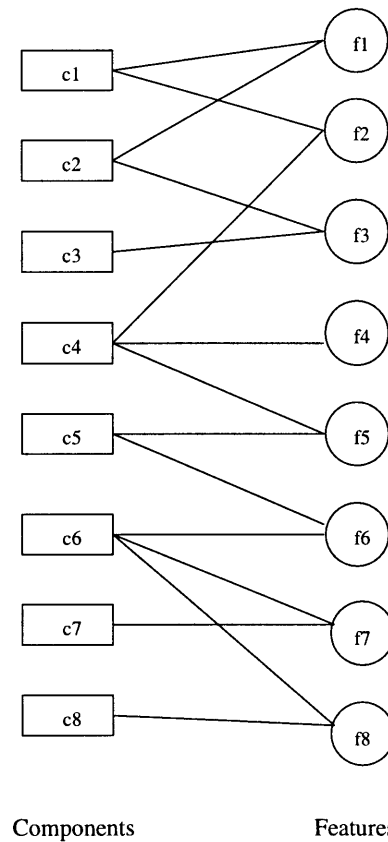


Figure 4-7: Software components and features represented as bipartite graph

symptoms [RNW83, NR86]. Software component retrieval based on a set of features is similar to this problem because, there is a many-to-many correspondence between the domain (diseases/software components) and the range (symptoms/features) in both the problems. While diseases can be considered as the software components, symptoms can be considered to be features provided by these software components. So the problem can be restated as identifying the set of all the software components that can cover a given set of features. This can be achieved by using the Reggia's Generalized Set Cover algorithm [RNW85, RNWP85].

### 4.6.1 Reggia's Generalized Set Cover Algorithm

Let  $S$  be a set. Let  $S_1, S_2, \dots, S_n$  be the subsets of  $S$ . Let  $R$  be the set of sets  $S_1, S_2, \dots, S_n$ . Let  $S'$  also be a subset of  $S$ . Reggia's Generalized Set Cover is one of the several Set Cover algorithms used to find most of the minimal cardinal sets  $G$  which contains those sets  $G_i$  of  $S_1, S_2, \dots, S_n$  the union of which covers the set  $S'$  (or is superset of  $S'$ ). Other solutions are also possible with higher cardinality. Let all these solutions be  $G_1, G_2, \dots, G_l$ .

In the above description, if  $S'$  represents the set of patient's symptoms (required features) and  $S_1, S_2, \dots, S_n$  represent the set of symptoms of individual diseases (features of individual components), then Reggia's Generalized Set Cover algorithm gives the minimum number of diseases (components) required to cover all the symptoms (diseases).  $S$  is the total set of symptoms (features) that are known to the system. Figure 4-8 shows an example with  $S, S'$  and  $S_1, S_2, \dots, S_{11}$ .  $S'$  is covered by the sets  $S_1, S_3, S_5, S_6, S_7$  and  $S_9$ .

However, a software component having a set of features may have additional constraints like the language in which the component is implemented and the operating system in which it can be used. Hence, even though a software component has a particular feature, it may be implemented in a language different from what the user wants. This software component should not be considered to cover the corresponding features. In the disease and symptoms analogy, it is like not considering the possibility of certain disease because of the patient's gender even though it can account for some of the symptoms. Here, gender of the patient acts as a constraint which prevents the possibility of the disease.

As an example, suppose components  $c_2$  and  $c_5$  are implemented in C++ and  $c_1$  and  $c_4$  are implemented in Java. Then for the same query as above, i.e., components with features  $f_1, f_3$  and  $f_5$ , together with the constraint that Java should be the implementation language will give only  $(c_1, c_4)$  as the possibility.

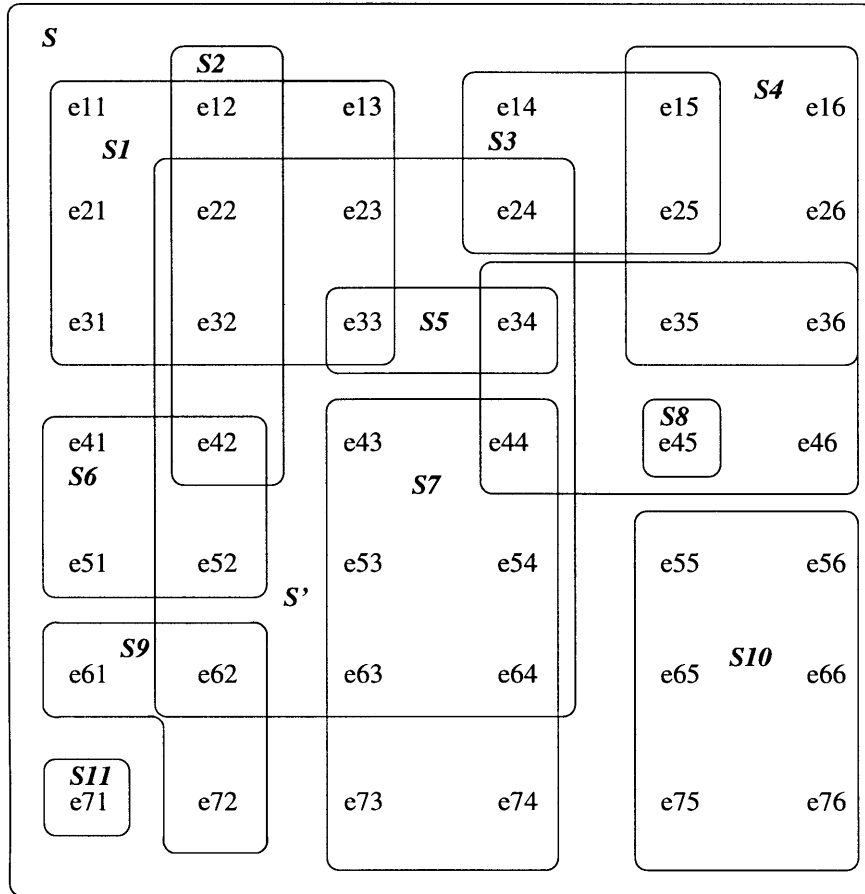


Figure 4-8: Example of Set Covering

Set Cover algorithms are NP-Complete. This will have a great impact on performance. As the library of reusable components increases, the time to run the set cover algorithm increases non-polynomially. Hence, it is necessary to reduce the search space based on some of the constraints like language, application domain and operating system. However, the user is given the option of not setting these constraints and searching the entire search space.

### 4.6.2 Multiple Software Component Retrieval

Reggia's Generalized Set Cover algorithm provides a solution set with minimal cardinality. However, it might be necessary to use several smaller components to build

a light weight application. Also, all the required features might not be available from the existing reusable components. In such a case, a solution that could cover as much of the functionality as possible should still be provided and the uncovered features should be notified to the user. Since the set cover algorithm is NP-complete the search space should be pruned where possible.

From the above considerations, a few steps are added before, after and during the use of Reggia's Generalized Set Cover algorithm. The following steps are used for retrieving multiple reusable software components

- A list of all the available features are maintained. Hence, when a new component is added to the library, all the new features that it has and that are not existing so far are added to the available features list. At the time of query, first the required features are filtered using this list. Then the remaining features are covered using the Reggia's Generalized Set Cover algorithm. In the above description of the algorithm, the set  $S'$  is filtered to  $S''$ . This is done prior to using the algorithm.
- As mentioned, the search can be pruned based on a few attributes like the language of implementation, operating system and application domain. Hence, the components are first filtered by verifying the specified attribute-value constraints. Then the remaining components are used to cover the filtered features, i.e., the subsets  $S_1, S_2, \dots, S_n$  are filtered based on the constraints. Let the filtered set of subsets be  $R'$ .
- Set Cover algorithm is run using  $R'$  and  $S''$ . It gives the possible solutions  $G_1, G_2 \dots G_m$ . Depending on the how the next component is selected in the Genset (Appendix D) of the algorithm<sup>2</sup> minimum set solutions or light weight solutions are obtained.

---

<sup>2</sup>A modified implementation of this algorithm in Scheme was given in the 6.034 course. I have reimplemented it in Java and converting a program written in a functional language to a pure object oriented language is an interesting experience.

- The solutions are ordered in the decreasing order of reusability.

### 4.6.3 Ranking the Retrieved Solutions

Because the same feature could be offered by multiple components, it is possible to have multiple solutions. When multiple solutions are possible there should be a way of ranking the goodness of these solutions. While it is not possible to have a theoretical expression for defining the goodness, empirical formulae could be provided. This ranking of the solutions will help the end user in making the decision of picking up the best solution.

The goodness of the solution is given as the Reusability Index on a scale of 100. Two main factors are considered for computing the reusability; *relevancy* and *composibility*. It is assumed that higher the relevancy, i.e., lesser the unrequired functionality in the solution, lighter is the solution. Few simple empirical formulae are used to provide three indices, composibility, relevancy and reusability. The formulae are only to have a feel for the ranking of the solutions and should not be given too much of importance<sup>3</sup>. However, the formulae are derived on certain relevant parameters. The indices and the relevant parameters are given below

**Composibility** Complexity of integrating components increases non linearly with increasing number of components. Hence, composibility which is a measure of ease of integrating effort decreases with increasing number of components non linearly. Also, larger the individual components greater is the integrating effort. Hence, composibility of a component is assumed to depend on the feature density of the solution  $G_i$ , and is calculated using the Equation 4.3.

$$featureDensity = \frac{\sum Si}{|Gi|} \quad \text{where } Si \in Gi \quad (4.3)$$

---

<sup>3</sup>several times I found the internet search results with lower relevancy percentage to b more relevant to my search



Composibility is then calculated using the Equation 4.4.

$$C = f(\text{noComponents}, \text{featureDensity}) \quad (4.4)$$

**Relevancy** Presently it is assumed that the features are matched hundred percent.

However, feature matching using both natural language processing and domain knowledge would result in lowering the relevancy. Also, as the number of un-required features present in the solution increases it's relevancy decreases. It is calculated using the Equation 4.5.

$$R = f(\text{requiredFeatures}, \text{totalFeatures}) \quad (4.5)$$

**Reusability** Reusability depends on both relevancy and composibility of the solution. It increases as either factors of the solution increases. Hence, it is simply measured as  $R_u = C * R$ .

The above indices can be used to help the user in choosing among the solution. These empirical formulae might depend on other factors but within the designed framework they are appropriate.

## 4.7 Summary of the Conceptual Model

Based on the requirements analysis presented in Chapter 3, a framework is developed for reusing software components. Following are the main functional features of this framework

- Automating the capture of design intent of reusable software components by improving the statistical analysis on available textual information by using the semantic knowledge derived from semantic analysis of available semantic information

- Providing a simple external view of the reusable software component to the end user
- Retrieving multiple reusable software components the composition of which could offer as much of the required functionality and ranking multiple solutions
- Serving as a framework which could be customized for individual domains

The following two simple semantic analysis on available semantic information are suggested which would increase the accuracy of classifying the components and identifying functional features using statistical analysis on textual information

- Constructing a Class Tree which provides the relative importance of individual classes in representing the high level functionality
- Analyzing for Creational Design Patterns to identify flexible functional features

Since the above reuse support model is based on the *available information* it does not add significant additional cost for implementing reuse methods as part of the traditional software life cycles. Such reuse methods could break the cultural and attitude barriers towards reuse. While keeping the cost of implementation low is attractive from the management point of view, keeping the user interface simple without the need for learning new specification languages and theorem proving techniques makes it attractive to the software development staff. Hence, reuse methods which could make both the top management and the development staff comfortable are more likely to become part of the existing software life cycles.

# Chapter 5

## Prototype Framework and Tools

*Prototypes helps to study, understand and observe new models*

---

---

Chapter 3 presented the requirements analysis and Chapter 4 presented the conceptual model for automatic capture of design intent which could be used for building large software systems by integrating reusable software components. This chapter describes the software framework and the tools developed based on the conceptual model. This framework is the design intent part of the DRIMER, a design recommendation and intent model extended for reusability [Vad96].

A reuse support system is developed as a client-server model. The client application provides a means of querying for required functionality and browsing the appropriate components. The server application provides a means of cataloging and retrieving the software components. Tools are developed to perform simple semantic analysis on the programs to derive semantic knowledge and use it in the statistical analysis of textual information.

The entire system is developed using Java [CH96] on Sun Sparc 5. The client application is a web based application. JavaCC [JCC97] is used to build the tools.

## 5.1 Framework

The design intent capture model is developed for building large software systems by integrating reusable software components. They are developed with the intention of making reuse methods part of the software development life cycles. It has been decided to have the reuse framework as a client-server model for the following reasons.

- Quality assurance is a part of traditional software development projects. It is necessary to have the same quality assurance for reusable components as well. Each company can define its own standards for deciding which component can be classified as reusable component and the implementation standards for software components. In order to maintain the quality of the reusable components, only a few people should have the permission to maintain the component library. Only these people will be able to use the tools to classify the component and add it to the library. The server takes care of administration permissions, storing the components and serving the clients in retrieving the appropriate components.
- It is assumed that the operating environment is behind a firewall and hence all the software documents could only be accessible to the employees of the companies.

The system has been implemented using Java for the following reasons.

- Java promises "write once and run every where" concept. This is important because most of the software companies have heterogeneous platforms.
- Java has constructs for concurrency as part of the language. Server implementation usually requires concurrency to serve multiple clients simultaneously.
- Java has libraries for user interface. This helps in providing friendly user interfaces for both the client and server applications.

- Java has libraries for network communication. Network communication is needed for client-server architecture.

### 5.1.1 Client

The client is developed as a Java applet accessible on any browser supporting Java. The DRIMER framework was initially developed using HTML forms [Vad96]. At that time Java was still in the initial stages and CGI scripts and HTML forms were more popular. However, forms can only be used for trivial user interface. Dealing with input errors requires either sending the information unprocessed to the server and getting the error information back or writing large Java Scripts for doing the error checking at the client side. Also, forms did not provide all the high level intuitive user interface that is possible using sophisticated user interface environments. Hence it has been decided to use Java and make the client application as an applet. Figure 5-1 shows the user interface of the client.

The user can first limit the search space by choosing the values for the following attributes

- Language of Implementation: The available options are C,C++ and Java.
- Operating System : The available options are Unix, Linux, Windows NT and Windows 95.
- Application Domain : The available options are Finance, Manufacturing, E-Commerce and Medical.

The user also has the option of specifying ANY for all these attributes. ANY means that the user is not concerned about the value of the attribute <sup>1</sup>. For example the user may not be particular about the language implementation. Also, some generic components may not be classified under any particular application domain.

---

<sup>1</sup>This is like a wild card '\*' to mention file names. Like "ls chap\*.tex"

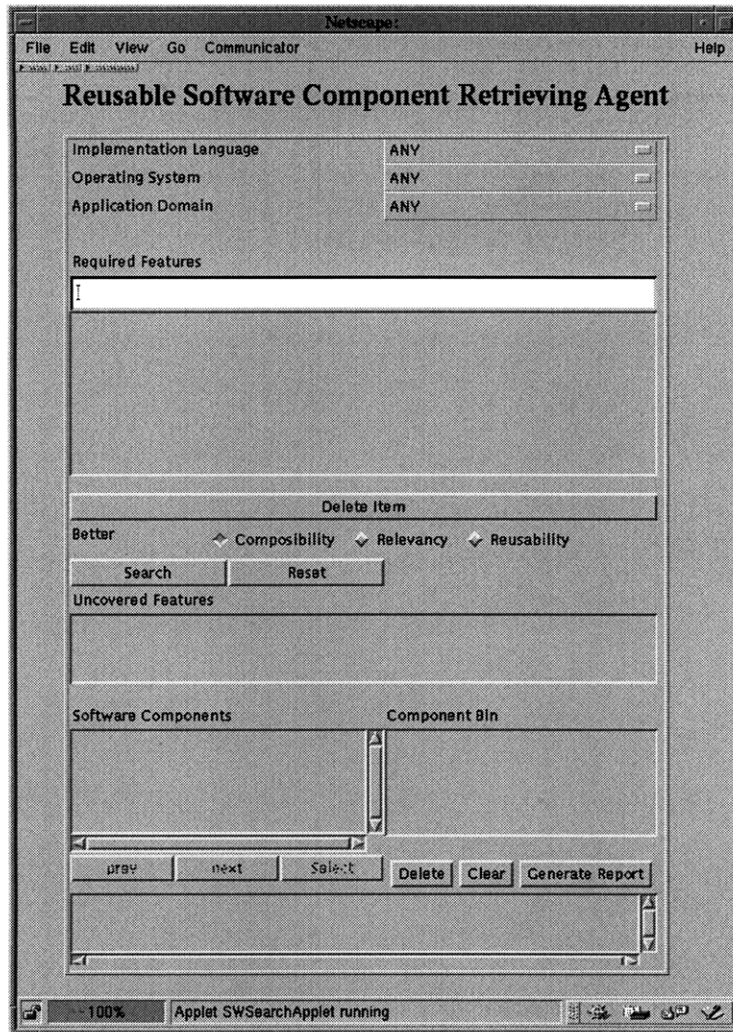


Figure 5-1: Web Client

Once the required attribute values are selected, the user can describe the required functional features of the components. Phrases like "capacitated transportation network" and "network communication using sockets" can be used. Once the features are specified and submitted, all these details goes to the server which returns the following results

- Set of uncovered features
- List of sets of components. Each set is a possible solution. Since several solutions could be possible, the user also has the option of specifying whether

he/she wants the solution optimized for *Composibility*, *Relevancy* or *Reusability* as shown in the Figure 5-1. In all cases the results are ordered in the decreasing order of *Reusability*. *Composibility* and *Relevancy* of each solution are also displayed.

The user can browse the results and select them into a folder. Once the components are placed in the folder the user can obtain the documents of each component online and browse them. A couple of examples describing the querying and browsing process is explained in Chapter 6.

At the end of the search process, the user can request for generating a report of the final selected components. The report is generated as a html page and is shown in a separate browser window. This is done by obtaining the JSObject of the applet's browser window and evaluating the JavaScript [JS97].

### 5.1.2 Server

The server is written in Java. Since the client is written as a Java applet, it is necessary to run the server on the same host from which the client url is accessed. This is because, Java has a security restriction that confines the applet's network communication capability to only the host from which it is loaded.

The server can be launched from the command line using the following command.

```
java SWCompMaintainer [-d < file.dat >] [-h]
```

The -d option is to load the details of reusable software components and features from a flat file. The -h option provides help on the server. If no -d option is specified then the server loads a default data which is useful for demonstration. File < *file.dat* > should contain all the functional features followed by the software components. One blank line should be left after the features. One feature is specified in each line as *id;feature*. Similarly, one component is specified in each line as *name;an;aemail;udir;pl;os;adomain;feature-ids*.

Field	Description
id	an unique name of the feature to be used in describing components
feature	the functional feature of the component
name	name of the component
an	component's author (or any contact person) name
aemail	author's email
udir	url directory of the component
pl	programming language of the component
os	operating system of the component
adomain	application domain of the component
feature-ids	a list of ids used in uniquely naming the features

Table 5.1: Software Component File Fields

A sample data file containing the details of software features and components is shown in Table 5.2 <sup>2</sup>.

The various documents of a software component are stored in the html documents directories so that they can be accessed by the client over the web. Since most of these documents are large text files and images, they are stored as normal files and no database is used. This also makes it easier to access them over the web. However, each of the component has a corresponding representation within the for the purpose of cataloging and retrieving. The server could have internal representation, external representation and also domain specific representation as mentioned in Chapter 4. The prototype presently has just the simple external representation. Sophisticated cataloging and retrieving can be achieved by providing the suitable `equals` function for the component features which is described in Section 4.5.2.

Adding new components into the component library requires the knowledge about the functional features of the component. If these details are readily available the library administrator could directly use the server to add these details. However, if these details are not readily available, a set of tools can be used to obtain the function-

---

<sup>2</sup>Each component is specified in only one line in the file. In the example shown in Table 5.2 is split to fit the page and in such cases the subsequent lines of a component are indented



```

s1;capacitated transportation
s2;network communication
s4;search algorithms
s5;markov model
s6;graphics editor
s7;single runway parameter calculation
s8;software component maintaining
s9;software component retrieval
s10;web based chat
s11;multiple discussion rooms
s12;remote console administration
s13;producer-consumer model
s14;html report generation
s15;web advertising agent

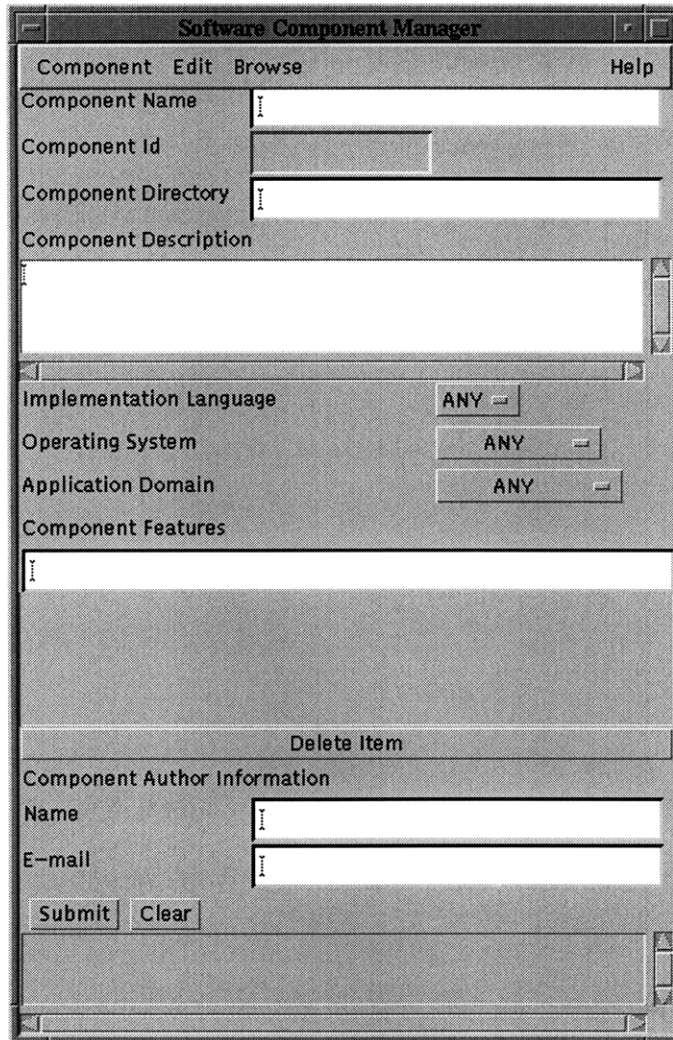
Algorithm for solving Capacitated Transportation Problems;*;*;csas;C;
  ANY;ANY;s1
Socket Library;*;*;socket;C++;ANY;ANY;s2
AI Search Algorithms;*;*;ais;C++;ANY;ANY;s4
Graphics Editor;*;*;gescreen;C++;ANY;ANY;s6
Algorithm for Calculation of single runway parameters;*;*;runway;C;
  ANY;ANY;s7
Software Component Reuse Support System;Siva K Dirisala;discu@mit.edu;
  http://web.mit.edu/discu/www/research/;JAVA;ANY;ANY;s2;s8;s9
Web Based Chat Application;Siva K Dirisala;discu@mit.edu;
  http://web.mit.edu/discu/www/chat/;JAVA;ANY;ANY;s2;s10;s11

```

Table 5.2: Datafile

ality of the component automatically. The details of the tools is given in the below section on Tools. Once the functional features of the component are obtained the user can enter these details along with other details like the language of implementation, operating system and application domain. Other details like the contact information of the component's author and the version number of the component also have to be input. Figure 5-2 shows the server side user interface for adding new components.

It is also possible to browse, edit and delete software components. Browsing features is also provided. This facility is useful for the component administrator to see if any of the features provided by the component are already existing and if so



The screenshot shows a window titled "Software Component Manager" with a menu bar containing "Component", "Edit", "Browse", and "Help". The main area is divided into several sections:

- Component Name**: A text input field.
- Component Id**: A text input field.
- Component Directory**: A text input field.
- Component Description**: A large text area with a vertical scrollbar.
- Implementation Language**: A dropdown menu with "ANY" selected.
- Operating System**: A dropdown menu with "ANY" selected.
- Application Domain**: A dropdown menu with "ANY" selected.
- Component Features**: A text input field.
- Delete Item**: A button.
- Component Author Information**: A section containing:
  - Name**: A text input field.
  - E-mail**: A text input field.
  - Submit** and **Clear** buttons.

Figure 5-2: Reusable Software Component Server

whether to refine the feature of the new component so that it is classified differently. The user interface for all these features of the server are provided in the next chapter on examples.

## 5.2 Tools

A Tool for simple semantic analysis of available semantic information is developed. The tool performs the first type of semantic analysis suggested in Chapter 4, which is

the construction of Class Tree to obtain relative weights. Construction of Class Tree requires parsing the programs. The tool is developed for parsing Java programs.

JavaCC [JCC97] is used to write the parser for parsing Java programs. JavaCC is a Java version of Compiler Compiler similar to LEX and YACC <sup>3</sup>. The grammar for Java is also available in the examples of using JavaCC. This grammar is used for parsing the Java code.

A tool is developed using JavaCC to perform semantic analysis on the programs. This tool can be run on the existing java programs and the corresponding information for building the class tree. This information is further used by the textual analysis tool which perform statistical analysis on textual documents and finally provide the possible words/phrases for functional features. The reusable component administrator could run all these tools, obtain the functional features of the component and then finally place the component in the library by specifying the component details to the server.

It is possible to come up with additional simple semantic analysis techniques to obtain semantic knowledge. New tools could be written for these techniques. The component administrator should be familiar with using all these tools. However, using these tools does not require learning new specification languages.

Presently only simple textual analysis is performed to provide the proof of the concept. The word frequencies are gathered and the weights are assumed to be proportional to these frequencies. Relative weights obtained from the semantic analysis are used to modify the weights obtained from textual analysis. This results in increased weights for the words representing the functional features. An example is provided in Chapter 6 showing how adding semantic information improves the performance. The semantic analysis tool stores the class names, their depth in the class tree and the relative weights in a file. Class names are sometimes abbreviated if they are long. For example, *SWCompMaintainer* actually means *Software Compo-*

---

<sup>3</sup>However, the functionalities of both LEX and YACC are put together in JavaCC.

*ment Maintainer* . Presently, it is assumed that the user supplies the actual name of the class in such cases. Hence, the user has to explicitly edit the relative weights file giving the full name of the class. Then, the textual analysis tool is run by giving the name of the text document file and the weights of the words are obtained. Then both the semantic and textual information is used to obtain new weights and then the resulting words with higher weights are presented to the user.

Sophisticated textual analysis like natural language processing and statistical analysis on pairs of words instead of individual words could be used to obtain phases instead of individual words. More about this is mentioned in the suggestions for future research directions in Chapter 7.

# Chapter 6

## Illustrative Examples

*A picture is worth thousand words, so is an example*

---

---

This chapter presents several examples showing the use of the client, server and the tools. The examples illustrate the retrieval of multiple components at the client, maintaining reusable software components at the server and using the tools to determine the design intent (functional features) of the component.

### 6.1 Client

The functionality of the client side interface is described in this section by giving several examples. The following main functionality is covered in the examples

- specifying constraints
- using wildcards to specify constraints and searching for multiple features
- uncovered features and relaxing constraints
- multiple solutions and the solution ranking

- selecting the solution for composibility, relevancy or reusability
- component bin and selecting preliminary components
- browsing the solutions online
- online report generation of the final selected components

### 6.1.1 Specifying Constraints and Features

The user can search for components by specifying the constraints and giving the features. The specification of constraints are shown in Figure 6-1. The constraints shown in the figure and their values are

Table 6.1: Constraint Attributes and valid values

Constraint Attributes	Values
Implementation Language	ANY, C, C++ and Java
Operating System	ANY, UNIX, LINUX, WINDOWS NT and WINDOWS 95
Application Domain	ANY, FINANCE, MANUFACTURING, E-COMMERCE and MEDICINE

The value ANY serves as a wild card. Specifying ANY will relax the particular constraint <sup>1</sup>. This is useful because sometimes even though there may not be a component with the required features and given implementation language, it would be useful to search irrespective of language and use the high level design documents of the retrieved components. Similarly, if one is looking for CORBA compliant components, both implementation language and the operating system can be ANY <sup>2</sup>. Also, certain features are common across multiple domains and in such cases they need not be classified as belonging to the one for which they were initially developed.

<sup>1</sup>the present prototype has very few components and hence they are constrained using only the implementation language

<sup>2</sup>provided the ORBs exist for the particular platform and there is a COBRA IDL mapping to the language

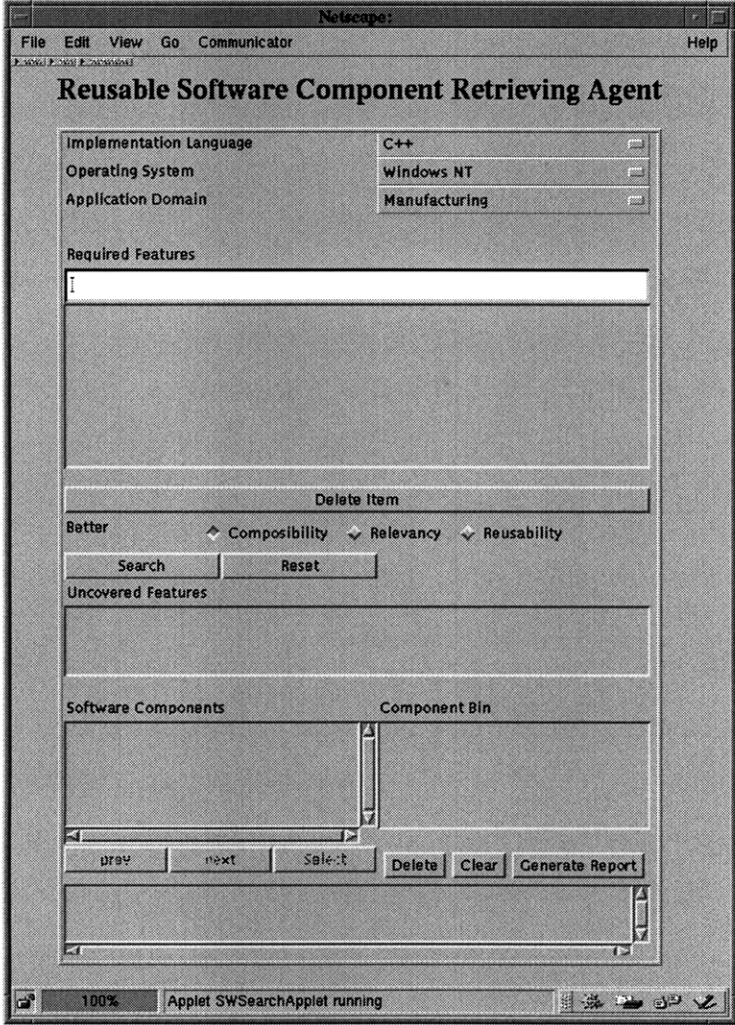


Figure 6-1: Specifying constraints for the required software component

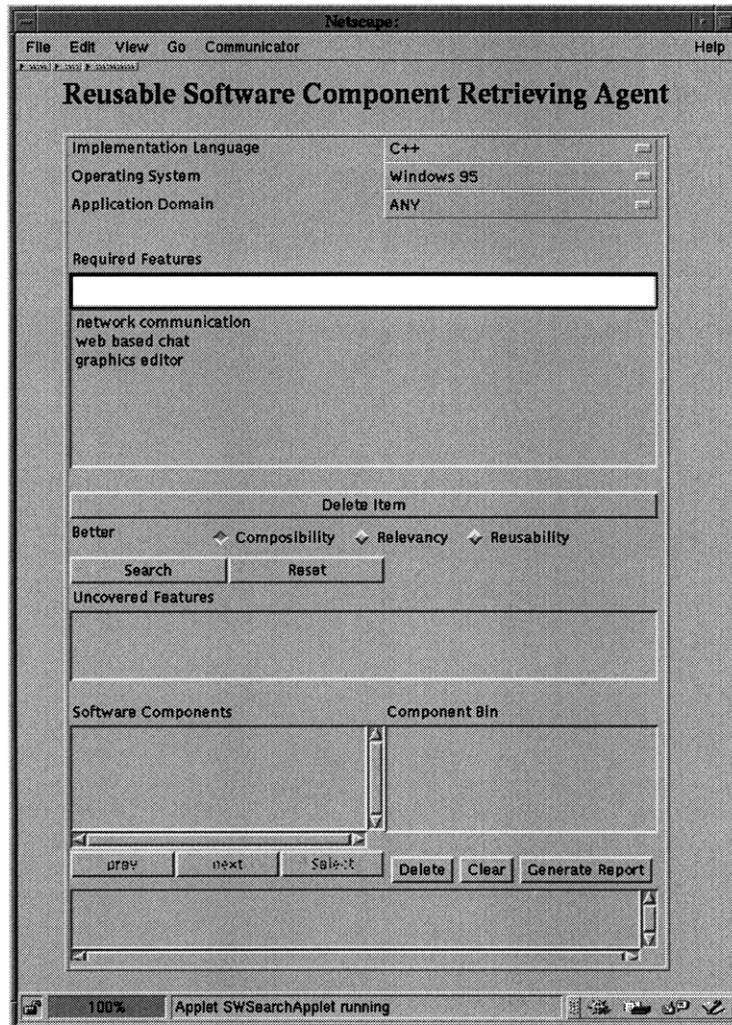


Figure 6-2: Specifying software features and relaxing the constraints

Figure 6-2 shows relaxing constraints and specifying required features. Also, the user has to choose the search to optimize for *Composibility*, *Relevancy* or *Reusability* discussed in Chapter 4. Examples are given in the next section explaining these alternative search goals.

The results of the search are shown in Figure 6-3. The search resulted in a single solution. The feature *web based chat* could not be covered by the existing components with the given constraints.

At this point, the user can change the language of implementation, say to Java



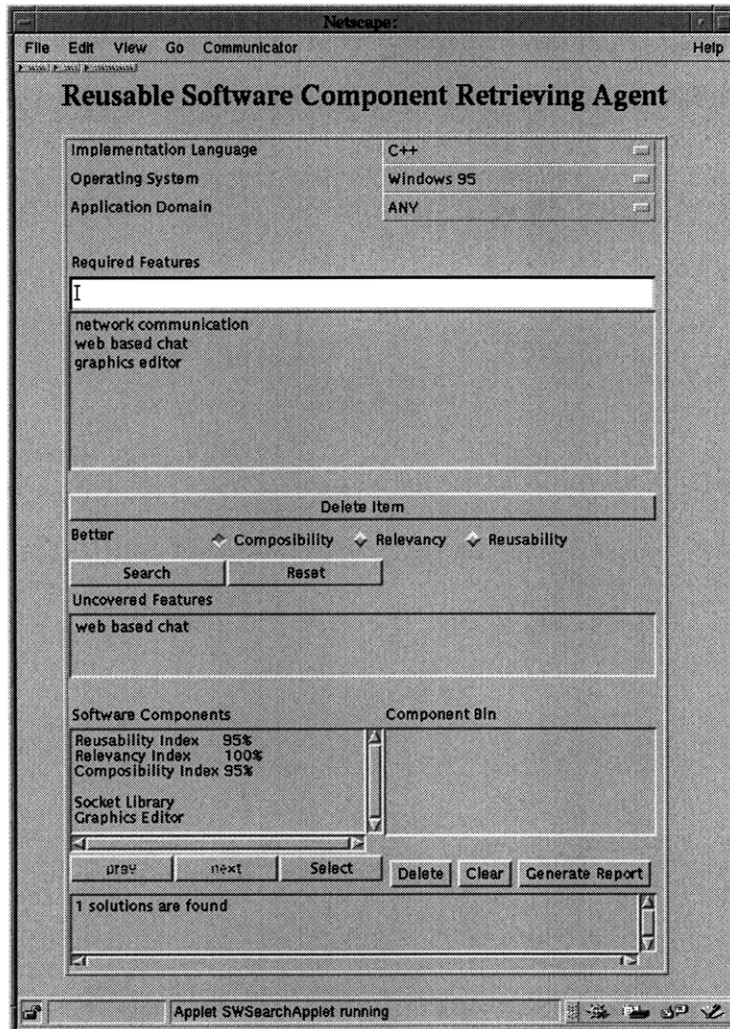


Figure 6-3: Results and unavailable features

and try for a solution that could cover all the features. However, this resulted in not providing some other functionality. If the user is more interested in high level design he/she can set the implementation language to be ANY and search again as shown in Figure 6-4. There are two possible solutions and the feature *web based chat* is also covered. Similarly, other constraints can be relaxed as needed and relevant components can be studied.

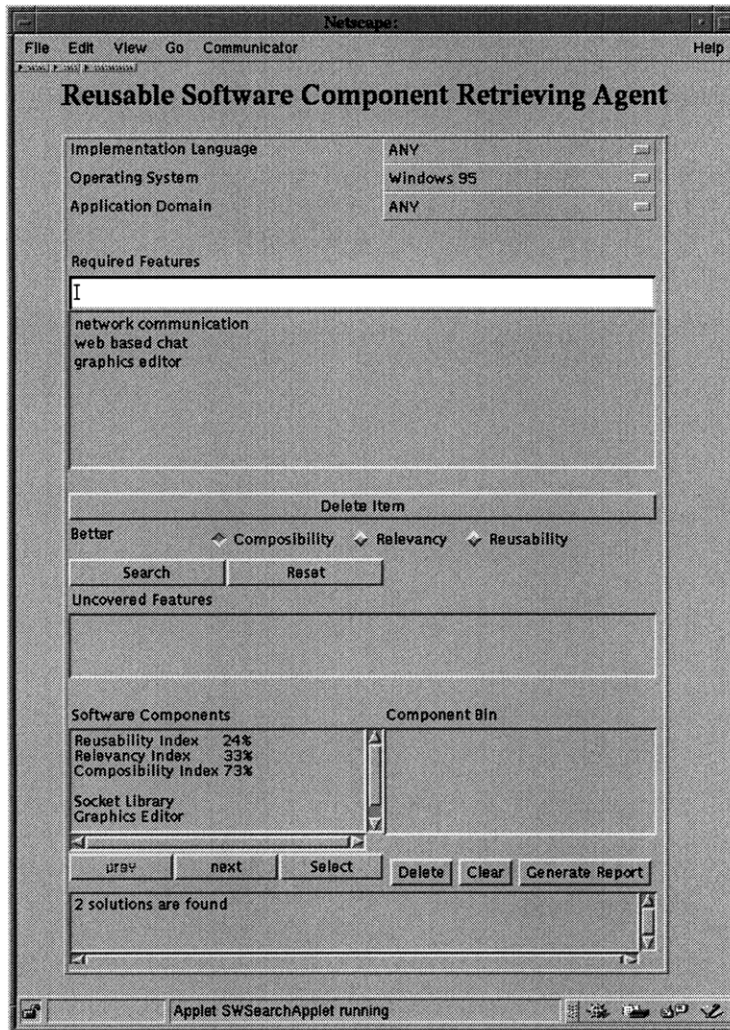


Figure 6-4: Searching for unavailable features by relaxing constraints

### 6.1.2 Selecting for Composibility, Relevancy or Reusability

When multiple solutions are possible the user should have an idea of choosing one over the other. Solutions are ranked according to the *Reusability*. The *Composibility* and *Relevancy* indices are also shown to the user. The following examples illustrates the concept of selecting the search for optimal *Composibility*, *Relevancy* or *Reusability*.

Figure 6-5 shows search requested for *Composibility* and there is one solution. The number of components given in the solution is two. The search for better *Composibility* tries to find solution with minimum number of components. Lesser the number

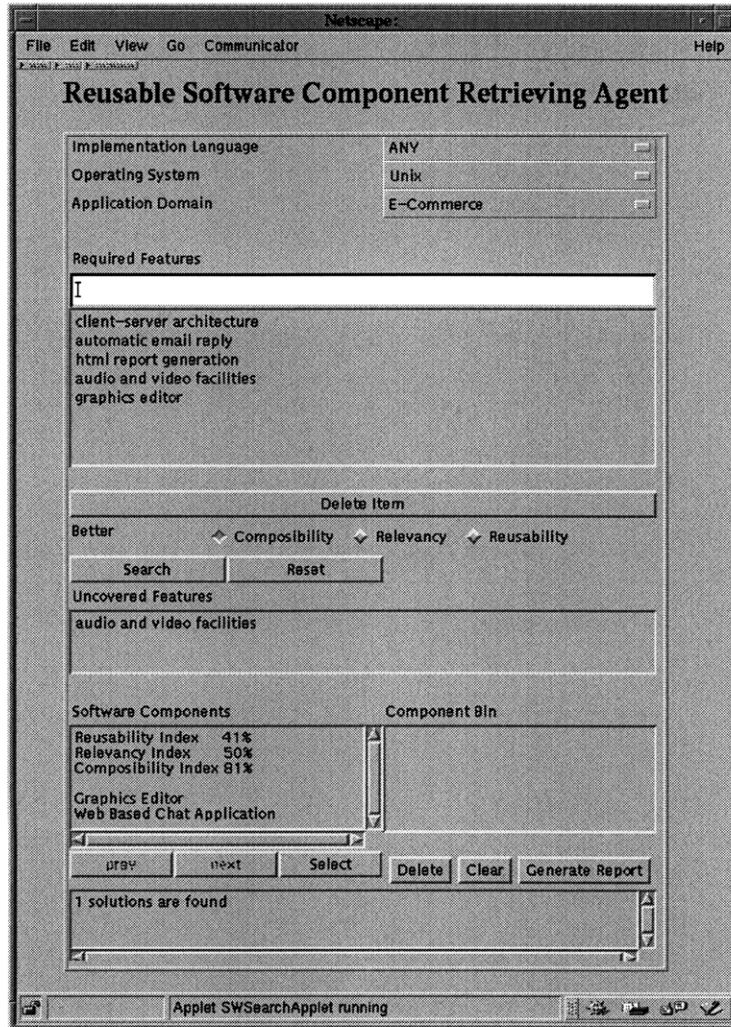


Figure 6-5: Search for better Compossibility

of components, lesser the integration effort and hence greater *Compossibility*.

Figure 6-6 shows search requested for *Relevancy* and there are two possible solutions. The number of components in each of these solutions is three. The search for better *Relevancy* tries to minimize the number of unrequired features of the components there by increasing the *Relevancy* of the solution to the requirements. Hence, the solutions in this search alternative usually consist of more components. It can be observed from Figure 6-5 and 6-6 that in the first case the *Compossibility* is higher while in the second case the *Relevancy* is higher.

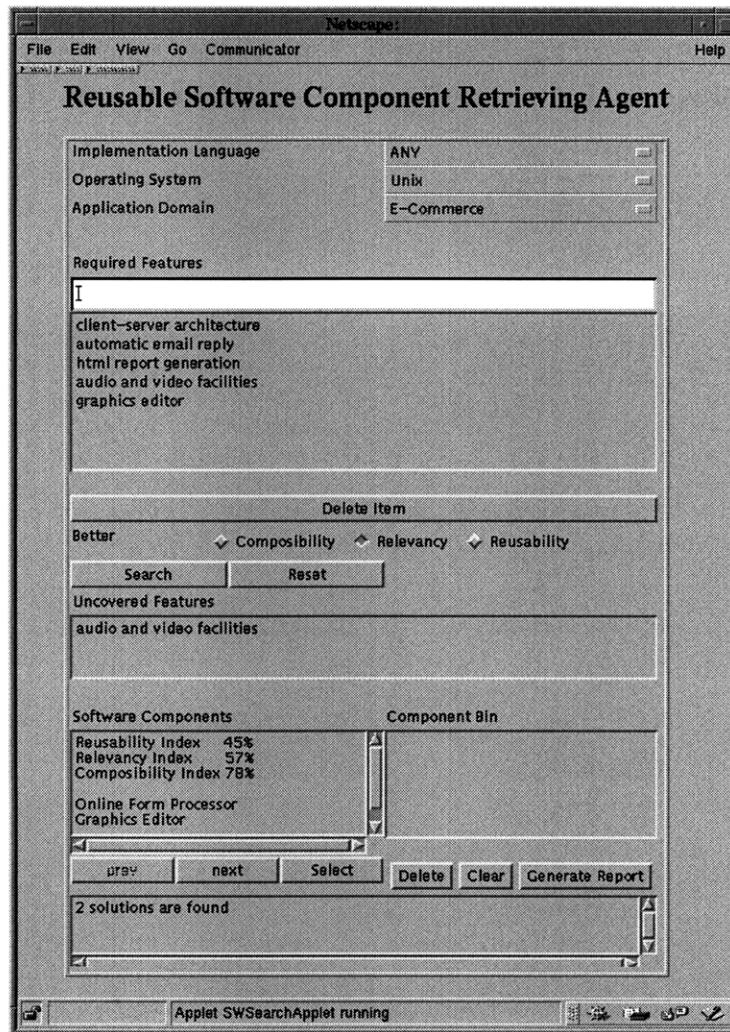


Figure 6-6: Search for better Relevancy

Both *Composibility* and *Relevancy* directly effect *Reusability*. If the user has no preference one over the other, then search can be selected for *Reusability*. This actually presents the solutions obtained in the other two cases.

Figure 6-7 shows search is requested for *Reusability* and all the above three solutions are provided.

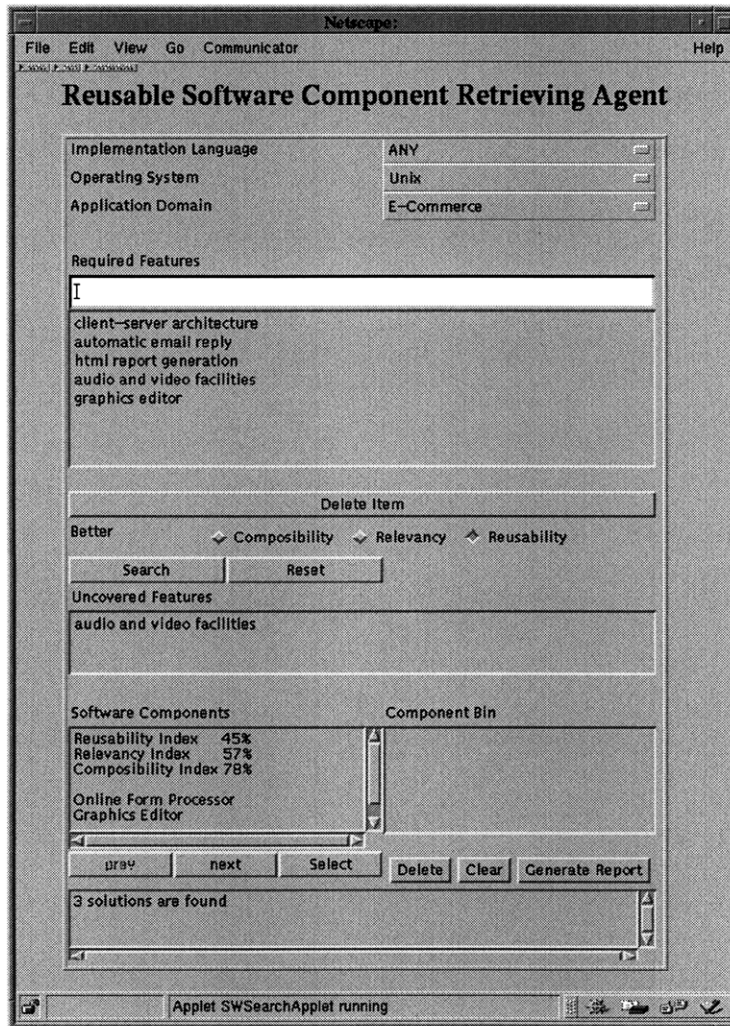


Figure 6-7: Search for better Reusability



### 6.1.3 Online Software Component Browsing and Report Generation

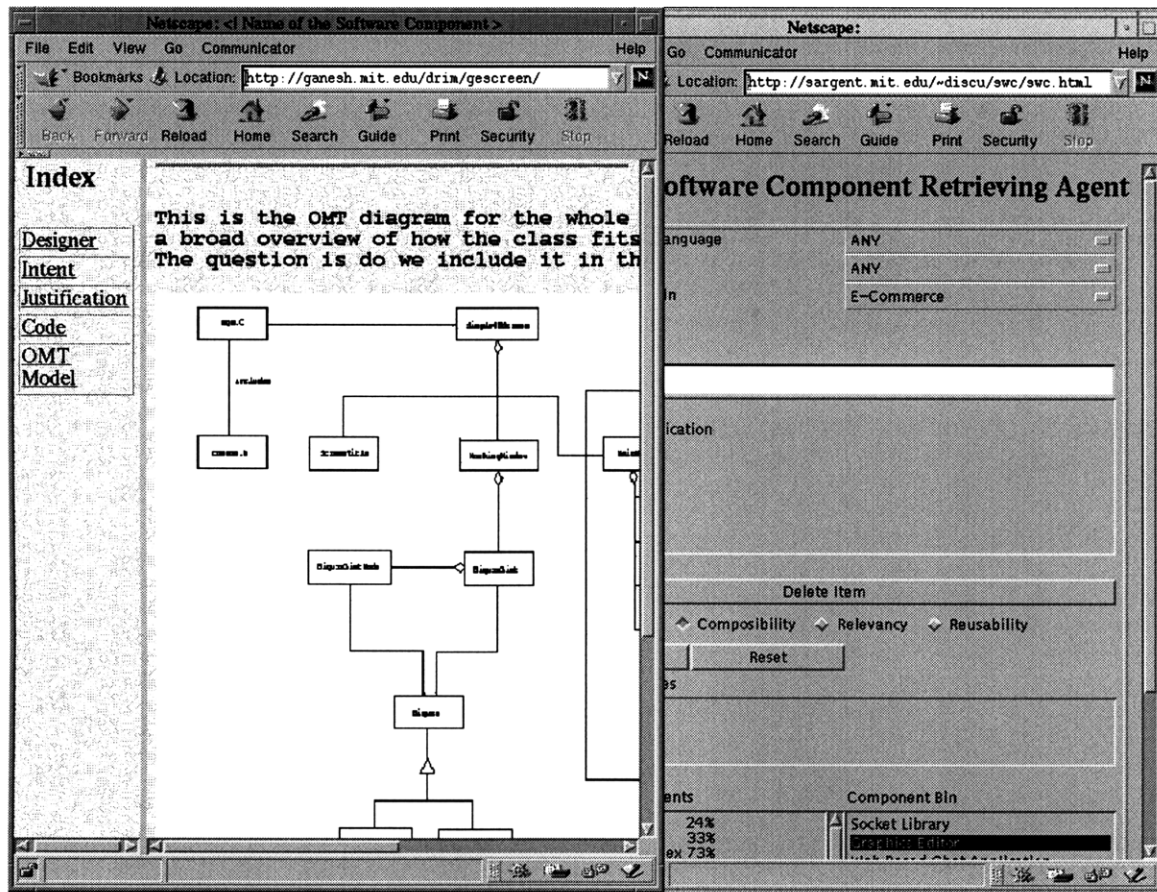


Figure 6-8: Online browsing of individual components

When there are multiple solutions possible, the user can browse the components online and decide on using them. A separate browser window is launched to browse the component details as shown in Figure 6-8.

All the useful components during the search process can be collected and a report can be generated as a web page as shown in Figure 6-9.

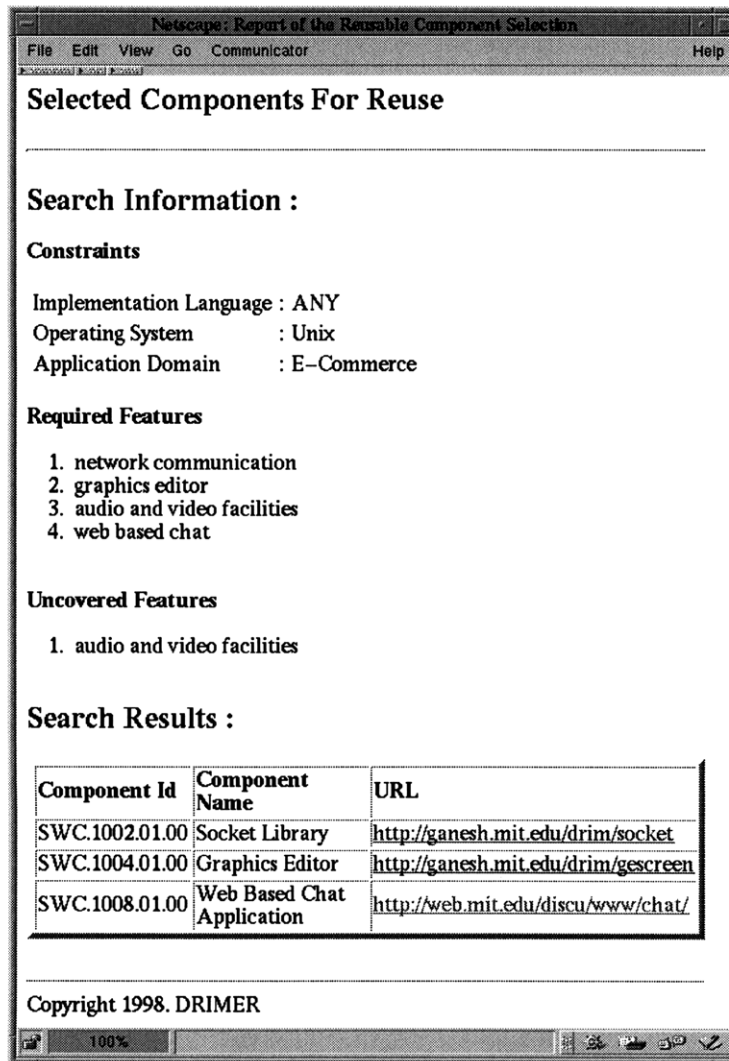


Figure 6-9: Report of selected components generated on the browser

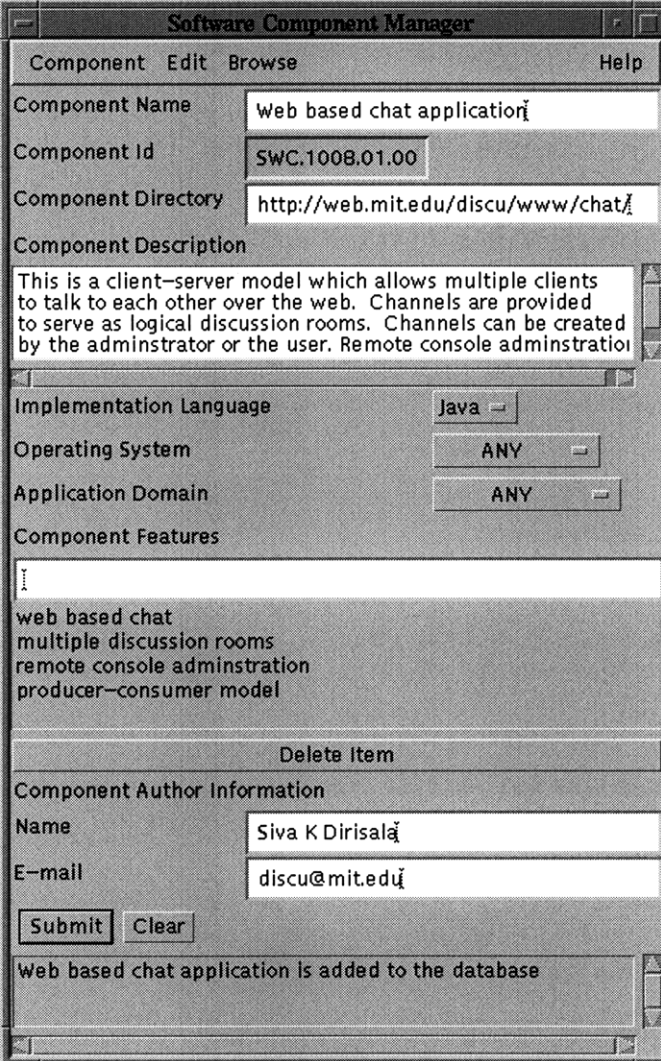
## 6.2 Server

The server is used to both catalog reusable software components and to serve the queries from the clients. The conceptual model in Chapter 4 and the architecture in Chapter 5 have described about the server and the retrieval algorithms. The cataloging functionality of the server is described using examples

- adding new reusable software component
- browsing existing reusable software components

- browsing existing features

### 6.2.1 Adding New Software Component



The screenshot shows a window titled "Software Component Manager" with a menu bar containing "Component", "Edit", "Browse", and "Help". The form contains the following fields and controls:

- Component Name:** Text box containing "Web based chat application".
- Component Id:** Text box containing "SWC.1008.01.00".
- Component Directory:** Text box containing "http://web.mit.edu/discu/www/chat".
- Component Description:** Text area containing "This is a client-server model which allows multiple clients to talk to each other over the web. Channels are provided to serve as logical discussion rooms. Channels can be created by the administrator or the user. Remote console administration".
- Implementation Language:** Dropdown menu set to "Java".
- Operating System:** Dropdown menu set to "ANY".
- Application Domain:** Dropdown menu set to "ANY".
- Component Features:** Text area containing "web based chat", "multiple discussion rooms", "remote console administration", and "producer-consumer model".
- Delete Item:** A button.
- Component Author Information:**
  - Name:** Text box containing "Siva K Dirisala".
  - E-mail:** Text box containing "discu@mit.edu".
- Submit** and **Clear** buttons.
- Status Bar:** A message box at the bottom stating "Web based chat application is added to the database".

Figure 6-10: Adding a New Software Component

Adding a new component is shown in Figure 6-10. There are three constraints shown in the figure namely programming language, operating system and application domain. The values of the constraints are specified. The values for these constraints can be easily obtained from the component's documents. Then the features are added.



Features are obtained by using the tools described later. The details of the person to be contacted regarding the component are optional.

## 6.2.2 Browsing and Modifying Existing Components

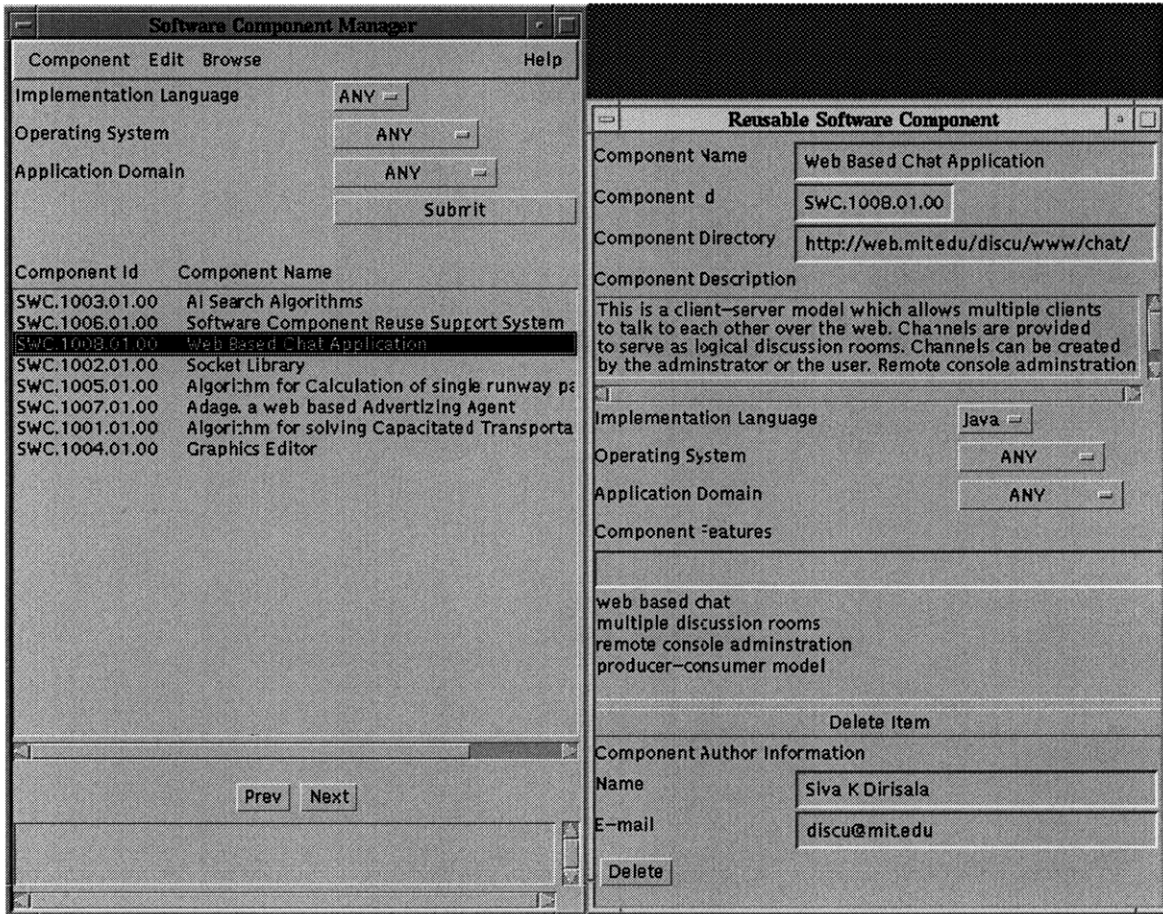


Figure 6-11: Browsing Software Components

The user can browse all the components. Components to be browsed can also be selected by specifying the above mentioned constraints. Figure 6-11 shows an example of browsing all the components without specifying any constraints. Selecting an individual component will further popup a window (Figure 6-11) giving a detailed description of the component. Then the component can be modified or deleted. Deleting a component will also delete all the associated features if they are not present in

any other component.

### 6.2.3 Browsing Existing Software Features

The user can also browse all the features. Figure 6-12 shows an example of browsing all the software features.

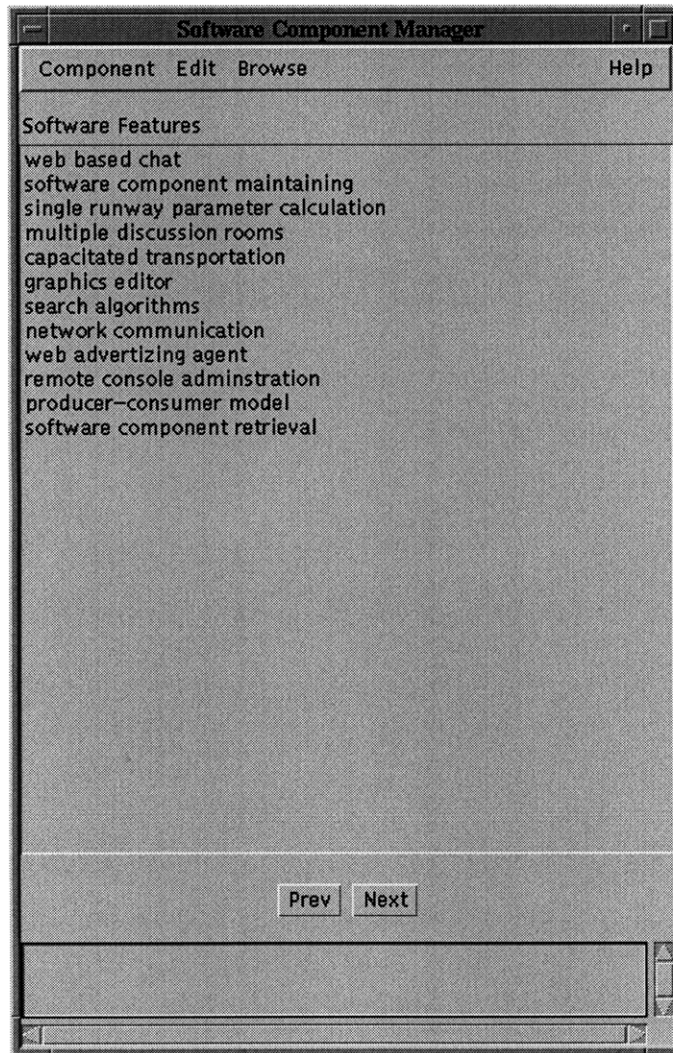


Figure 6-12: Browsing Software Features

## 6.3 Tools

Tools are developed to construct Class Tree and to perform a simple word frequency based statistical analysis. These tools are used to find out how the semantic information could help in the statistical analysis of textual information. Both types of information and analysis is done on a known Java application which is a client server based chat application. First the details of the chat application are given and then results of the statistical analysis, construction of Class Tree and finally the combined results are presented. The results in either cases are compared and observations are presented.

### 6.3.1 Chat Application

It is a client-server application written in Java. It allows several users to chat with each other over the web. It is possible to have several discussion channels and the user can enter any of these channels. It is possible to send messages to individual user or to the entire channel. The server administrator can remove a client or a channel. It is also possible to perform remote server monitoring to find out who is entering and leaving the chat and what are the new discussion channels being created.

This application has three separate parts, the server, the remote console and the client. While the server is a Java Application, the remote console and the client are Java Applets providing access to them over the web.

### 6.3.2 Analysis using Textual Information

The above mentioned application has a user's manual written as part of the project and without concerning reuse. The textual analysis tool simply parses the plain textual document and obtains the statistical frequency of individual words. Based on the frequency, they are given weight of importance to represent functional features of the application. Here the weights are directly taken as the frequency count.

The tool presently has a common words dictionary that has words used mainly for the structure of the sentence and also the most common words. The tool also considers only words that have atleast three characters. Presently it does not have the capabilities to identify the different forms of a word as a single word.

Table 6.2: Results of Analysis using only Textual Information

Word/Phrase	Freq	Weight	Rank	Word/Phrase	Freq	Weighth	Rank
channel	47	47	1	run	5	5	26
channels	18	18	2	root	5	5	27
persistant	15	15	3	insensitive	5	5	28
message	13	13	4	given	5	5	29
description	13	13	5	follows	5	5	30
chatserver	13	13	6	add	5	5	31
transient	12	12	7	using	4	4	32
user	11	11	8	time	4	4	33
server	11	11	9	shutdown	4	4	34
remove	11	11	10	sent	4	4	35
commands	10	10	11	running	4	4	36
client	10	10	12	java	4	4	37
channelid	10	10	13	displayed	4	4	38
send	9	9	14	details	4	4	39
see	8	8	15	created	4	4	40
list	8	8	16	create	4	4	41
clients	8	8	17	chatting	4	4	42
case	8	8	18	chatclient	4	4	43
above	8	8	19	chat	4	4	44
syntax	7	7	20	channeltype	4	4	45
userid	6	6	21	cannot	4	4	46
possible	6	6	22	below	4	4	47
people	6	6	23	always	4	4	48
enter	6	6	24	about	4	4	49
connected	6	6	25	users	3	3	50

When this tool is run on a three page manual <sup>3</sup>, there are 257 unique non-common words with atleast three letters. The weight (frequency) and the ranking based on

<sup>3</sup>the manual has 222 lines, 1559 words and 11,104 characters. This information is obtained using wc utility on Unix

the weights for the first 50 words are given in Table 6.2.

### 6.3.3 Class Tree, a semantic analysis

Class Trees are constructed for the server, client and the remote console modules of the chat application. The class trees for the server and client are shown in Figures 6-13 and 6-14.

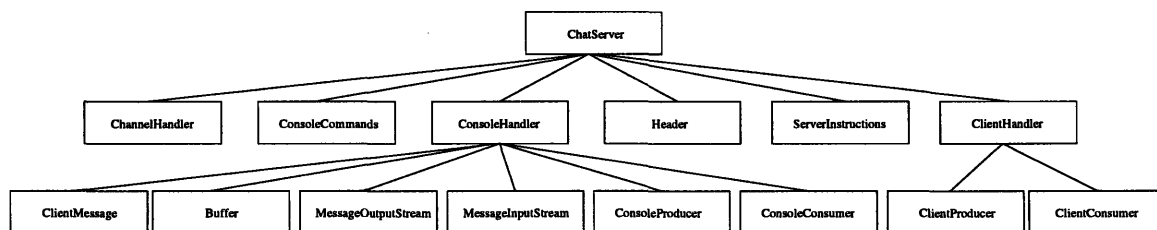


Figure 6-13: Class Tree of the Chat Server

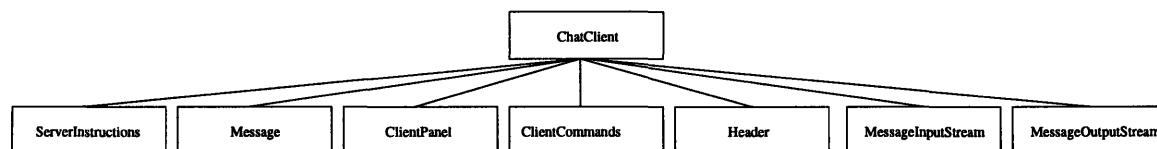


Figure 6-14: Class Tree of the Chat Client

The class tree for the console module is similar to the chat client except the root class is ChatConsole instead of ChatClient <sup>4</sup>.

### 6.3.4 Analysis using Textual Information and Semantic Information

The relative weights given to the class names based on the class tree are shown in Table 6.3. These relative weights are used to modify the weights found from the textual based on the frequency count. As a result the rank of some of the words have

<sup>4</sup>actually, most of the code has been reused

Table 6.3: Relative Weights from the Class Tree of Chat Application

Words/Phrases	Class Tree Depth	Relative Weight Factor
ChatClient	0	1.000
ChatServer	0	1.000
ChatConsole	0	1.000
Message	1	0.500
Header	1	0.500
MessageOutputStream	1	0.500
ServerInstructions	1	0.500
ClientCommands	1	0.500
MessageInputStream	1	0.500
ClientPanel	1	0.500
ConsoleHandler	1	0.500
ConsolePanel	1	0.500
ClientHandler	1	0.500
ChannelHandler	1	0.500
ConsoleCommands	1	0.500
MessageOutputStream	2	0.333
ConsoleConsumer	2	0.333
Buffer	2	0.333
ConsoleProducer	2	0.333
ClientConsumer	2	0.333
ClientMessage	2	0.333
NonNode	3	0.250

changed. The first 50 words, their frequency, modified weights and the new rank are shown in Table 6.4.

Following observations can be made from Tables 6.2 and 6.4

- The word *channel* which had a initial weight of 47 is much higher than the other words. This might mislead the user to give high importance to this. The modified weights are much closer to each other there by reducing the chance of giving too much of weightage to a particular word.
- The word *channel* which ranked 1 with the initial weights is ranked 2 with the modified weights. The word *chatserver* got rank 1 and this actually serves as

Table 6.4: Results of Analysis using Textual and Semantic Information

Word/Phrase	Freq	Weight	Rank	Word/Phrase	Freq	Weight	Rank
chatserver	13	13.00	1	enter	6	1.50	26
channel	47	11.75	2	connected	6	1.50	27
message	13	6.50	3	run	5	1.25	28
channels	18	4.50	4	root	5	1.25	29
chatclient	4	4.00	5	insensitive	5	1.25	30
persistant	15	3.75	6	given	5	1.25	31
description	13	3.25	7	follows	5	1.25	32
transient	12	3.00	8	add	5	1.25	33
chatconsole	3	3.00	9	using	4	1.00	34
user	11	2.75	10	time	4	1.00	35
server	11	2.75	11	shutdown	4	1.00	36
remove	11	2.75	12	sent	4	1.00	37
commands	10	2.50	13	running	4	1.00	38
client	10	2.50	14	java	4	1.00	39
channelid	10	2.50	15	displayed	4	1.00	40
send	9	2.25	16	details	4	1.00	41
see	8	2.00	17	created	4	1.00	42
list	8	2.00	18	create	4	1.00	43
clients	8	2.00	19	chatting	4	1.00	44
case	8	2.00	20	chat	4	1.00	45
above	8	2.00	21	channeltype	4	1.00	46
syntax	7	1.75	22	cannot	4	1.00	47
userid	6	1.50	23	below	4	1.00	48
possible	6	1.50	24	always	4	1.00	49
people	6	1.50	25	about	4	1.00	50

a primary functional feature than the fact that the chatserver has discussion channels.

- The word *chatconsole* which has a frequency of 3 is not ranked among the top 50 even though the application provides remote server monitoring console. With the modified weights, it is ranked 9. Hence, when the weights are assigned directly from the textual analysis assigning a threshold frequency count, which is the frequency count beyond which the words need not be considered, is very

difficult. However, once the weights are modified with the relative weights derived from the semantic analysis, important words tend to obtain better weights making them to move up to get a better rank. Hence, it is possible to set a threshold weight beyond which the words may not be considered.



# Chapter 7

## Conclusions

*Intelligence is Natural, Knowledge is Acquirable and the search for it is Perennial*

---

---

This chapter first presents the conclusions from the research and then suggests the future research directions.

### 7.1 Conclusions

After a careful study of the existing literature both on the technical and cultural aspects of software reuse it has been realized that the reuse is practical only if the technology is oriented in a direction that breaks the cultural and attitude issues towards software reuse. To that effect, two major factors have been identified

- Keeping the cost of implementing the reuse technology as minimal as possible. This made designing the reuse technology using as much of the existing information as possible to derive the knowledge for reusing the software components. Once the cost of implementation of reuse methods is negligible it attracts the high level management to initiate the reuse process.

- The reuse techniques should be easier to use. Programming itself is a non-ending bug fixing process and it is difficult to express the details of the software component in yet another specification language and get it right at the first attempt. Especially at the stage of preliminary design phase it is necessary to have simple methods to identify the existence of reusable components. Suppose, if the user formally expresses the required functionality and could not find a reusable component satisfying the functionality then he/she loses a day or week of work in formalizing the requirements using the new specification language. So, the component search should be as simple as possible initially. However, if there are several components with similar functional features then one could use more rigorous methods to find the most appropriate component among those retrieved from the initial search.

From the above considerations a reuse model which automatically captures the design intent of the software components and helps in retrieving multiple components whose composition could offer the required functionality has been designed. The functionality uses only the information that already exists as part of the project without reuse as a concern. Also, it provides a simple interface for the end user.

The main conclusions made in this research are

- Reuse techniques could be designed based on only the *available information* without the need for a new specification language. Since this reduces the cost of implementation such techniques are more likely to become part of the new software life cycle models.
- Textual information and statistical analysis provide simple user interfaces. However, to better the performance based on these it is necessary to use some semantic knowledge that could be derived from simple semantic analysis on the available semantic information.

- The representation of the software component should be simple from the end user's point of view for the preliminary search. Complex internal representations and domain knowledge could be used to further enhance the classification accuracy and search relevancy.

Even though this research has mainly focused on the design intent reuse of software components, it is possible to extend the model to the reuse of any artifact. Usually any artifact development has both textual and semantic information. Coming up with some means of deriving the semantic knowledge that enhances statistical analysis is the key to extending this model to reuse other types of artifacts.

## 7.2 Future Research Directions

This research has concentrated mainly on the automatic design intent capture from available information. As mentioned in Chapter 3, capturing design rationale requires inherent information along with the available information. It could be however, possible to try capturing design rationale automatically from the design of the artifact itself when the design is based on the design patterns. Instead of making the user express the product in a new specification language, restricting the user to use design patterns could help in capturing the design rationale. Capturing both the design rationale and intent will significantly improve the reuse support system. But at the same time, the model for design rationale capture should mostly depend only on the available information and the possibility of such models could be investigated.

It would be interesting to study the relation between the simple external representation of the software component with some complex internal representations mentioned in Chapter 4 and how to map from one to the other. Also, more natural language processing techniques could be added to the user interface by giving the flexibility to the user to express the features in English more naturally without confining to simple phrases.

It could also be possible to use databases providing textual information as one of the data type and also providing SQL (Structured Query Language) extensions to retrieve information from the textual documents. For example, Oracle's ConText [ORA97] extends SQL to provide retrieving relevant information from textual documents. In addition to exact word and phrase searches and traditional Boolean operations, ConText handles multilingual stemming, to match plurals, past tenses, and other alternate forms of words; and uses fuzzy-match and sounds-like, to match misspelled words and other "close" words. ConText also features proximity searching, relevance ranking, stop lists and an ISO-compliant thesaurus framework for synonym and category-type searching. Hence, this type of databases could be used to perform statistical analysis on textual documents more accurately with their rich natural language processing capabilities.

# Appendix A

## OOP and Design Patterns

*Objects Provide Life, Patterns Provide Longevity*

---

---

### A.1 Object Oriented Programming (OOP)

Object Oriented Programming is a paradigm that helps in developing "industrial strength" software [Boo94]. The main shift in this paradigm is from procedure centric computation to data (object) centric computation. It is easier to understand and decompose large real life systems in terms of objects. Also, it is easier to develop generic frameworks using object oriented paradigm with the concept of polymorphism and inheritance. The following language features are required for developing object oriented software systems

**Abstraction** is the means of decomposing large systems into several smaller objects (class of objects) that have a well defined behavior and interact with each other to provide the required functionality. It is the isolation of a well defined entity/functionality/process. Physically tangible objects are the primary candidates for abstraction. Once these objects are identified the relations and

interactions among these objects become either messages or abstractions by themselves depending on the degree of flexibility and granularity required for the objects and the system.

**Encapsulation** is the means of providing the public interface of an abstraction.

The same interface of an object can have different implementations and often not all the implementation data/functions are needed as the public interface. Hence, encapsulation provides a means of hiding the implementation details and providing only the interface of an abstraction.

**Inheritance** is the means of organizing the objects (classes) hierarchically. More generic the abstractions, higher are they in the hierarchy. There are two types of inheritance; implementation inheritance and type inheritance [Boo94]. It is possible to have multiple inheritance by inheriting from multiple abstractions. However, it is better to have only multiple type inheritance and not multiple implementation inheritance.<sup>1</sup>

**Polymorphism** is the means of having different behavior based on the target object.

The same interface can be implemented in different ways based on the type (class) of the object. Irrespective of the specific object, it is often possible to generalize the computation based on certain interface from the participating objects. This interface is defined in the base class and the framework is written using the base class. However, the actual behavior of the system depends, at the runtime, on the specific target object (class) derived from the base class. The derived object (class) has its own implementation of the interface.

Languages providing the first two features mentioned above are object based and those providing all the above features are object oriented. Object oriented languages

---

<sup>1</sup>C++ does not distinguish between implementation and type inheritance and but it provides multiple inheritance. However, Java differentiates implementation and type inheritance to some extent. It provides single implementation inheritance and multiple type inheritance.

are more useful for developing generic frameworks.

## A.2 Design Patterns

Software design patterns are tested and proven recurring programmatic idioms offering flexibility. There are several kinds of design patterns and Creational patterns, Structural patterns and Behavioral patterns can be found in [GHJV95]<sup>2</sup>. Only the details about Creational patterns taken from [GHJV95] are reviewed here.

### A.2.1 Creational Design Patterns

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object. Creational patterns can be thought of as virtual constructors.

Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hard coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class.

There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete class the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. Consequently, the creational patterns give you a lot of flexibility in *what* gets created, *how* it gets created, and *when*. They let you configure a system with "product" objects that vary widely in structure and functionality. Configuration can be static (that is, specified

---

<sup>2</sup>The authors are well known as gang of four, GOF

at compile-time) or dynamic (at run-time).

**Factory Method** lets a class defer instantiation to subclasses. Only the interface for creating an object is defined, but which class to be instantiated is given to the subclasses. Figure A-1 shows the structure of this pattern.

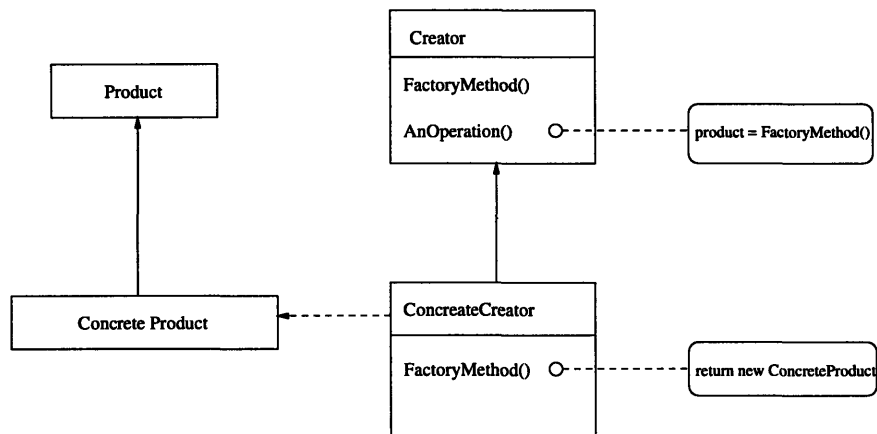


Figure A-1: Structure of Factory Method *adapted from [GHJV95]*



# Appendix B

## COM and JavaBeans

*Language, environment and location are no barriers for communication*

---

---

COM and JavaBeans are two component object models. While COM defines a binary and network standard for interoperability and hence language independent, JavaBeans is specific to Java. The next two sections present these two component models.

### B.1 Component Object Model, COM

COM defines a binary standard to allow interoperability on any operating system or hardware platform, and a network standard for interaction on multiple platforms. Figure B-1 shows the diagram of a VTable to implement COM binary standard.

These standards allow interoperability of object and applications written by different programmers. The following example from [COM95] tells why the binary and network standards are important.

”For example, a word processor application from one vendor can connect to a spreadsheet object from another vendor and import cell data from that spreadsheet

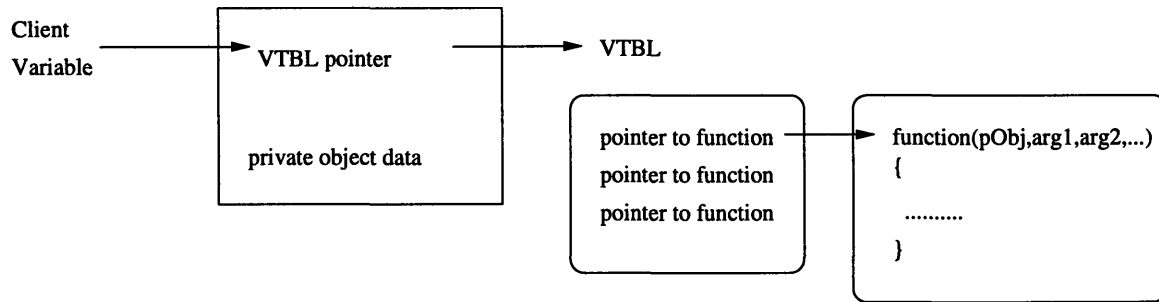


Figure B-1: VTable to implement COM binary standard

into a table in the document. The spreadsheet object in turn may have a "hot" link to data provided by a data object residing on a mainframe. As long as the objects support a predefined standard interface for data exchange, the word processor, spreadsheet, and mainframe database don't have to know anything about each other's implementation. The word processor need only know how to connect to the spreadsheet; the spreadsheet need only know how to expose its services to anyone who wishes to connect. The same goes for the network contract between the spreadsheet and the mainframe database. All that either side of a connection needs to know are the standard mechanisms of the Component Object Model."

Some of the features of COM [COM95] are,

- It uses globally unique identifiers to identify object classes and the interfaces those objects may support.
- It provides methods for code reusability without the problems of traditional language-style implementation inheritance.
- It has a single programming model for in-process, cross-process, and cross-network interaction of software components.
- It encapsulates the life-cycle of objects via reference counting.
- It provides a flexible foundation for security at the object level.

The standards set by COM allows object interoperability through the use of interfaces. If a generic component is designed it can be reused offering many interfaces for many programs.

The use of interfaces offer several benefits:

- The evolution of applications
- Low overhead of component object interaction
- Local/remote transparency
- Language independence

## B.2 JavaBeans

The definition of a Java Bean as given in [JBn97] is

”A Java Bean is a reusable software component that can be manipulated visually in a builder tool.”

JavaBeans allows developers to create reusable software components that can then be assembled together using visual application builder tools from independent software developers. JavaBeans is designed to be the platform-neutral, component architecture for Java. The JavaBeans specification defines a set of standard component software APIs for the Java platform.

The typical unifying features that distinguish a Java Bean are [JBn97]

- Support for ”introspection” so that a builder tool can analyze how a bean works.
- Support for ”customization” so that when using an application builder a user can customize the appearance and behavior of a bean.
- Support for ”events” as a simple communication metaphor that can be used to connect up beans.

- Support for "properties", both for customization and for programmatic use.
- Support for persistence, so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later.

Java Beans is designed to work well in a distributed world-wide-web environment and the three primary network access mechanisms that are available to Java Bean developers on all Java platforms are Java RMI, Java IDL and JDBC.

Scenarios of building applets using Java Beans are given in [JBn97].

Studying the features of both COM and Java Beans shows that these standards make it possible to write reusable components that can be easily integrated.

# Appendix C

## CORBA

*Be it stocks or objects, trading needs a broker*

---

---

Common Object Request Broker Architecture, CORBA, is an architecture from the Object Management Group (OMG). It is an architecture for developing distributed object oriented systems. Typically multi-tier applications are built using CORBA.

The advantage of using CORBA is that the client and the server can be on different types of platforms and also they can be implemented in different languages. For example, the server can be written in C++ on a Sun Sparc workstation and the client can be an applet written in Java and browsed on a PC.

In a distributed environment, the objects could reside at the client or on several servers. With CORBA, the application programmer need not be concerned about the location of the object because the methods are automatically invoked on the corresponding objects. To the client application programmer it appears as if writing a single application, i.e., as if the objects exist in the same address space. Figure C-1 shows the CORBA distributed object model [ORB97].

CORBA specifies an Interface Definition Language, IDL, for defining the services

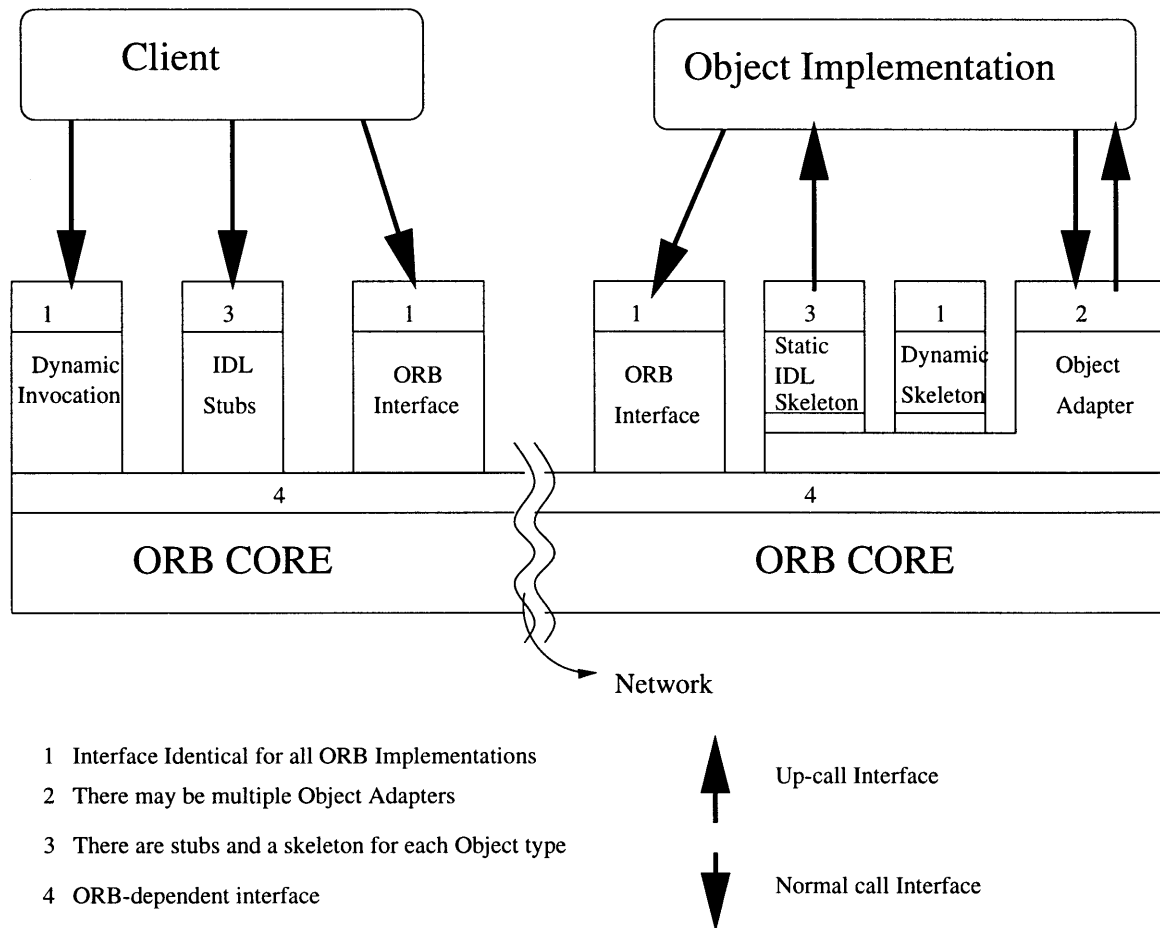


Figure C-1: Common Object Request Broker Architecture, CORBA *adapted from [ORB97]*

provided by the server. Once the services are published using IDL, both the server and the client can be developed independently on different types of machines and using different languages. This provides greater flexibility in design and implementation choices.

# Appendix D

## Generalized Set Cover Algorithm

*Hard problems can be often solved by suitable modeling, algorithms and heuristics*

---

---

### D.1 Generalized Set Cover (GSC)

Generalized Set Cover Algorithm (GSC) is proposed by Reggia, *et al* [RNW85, RNWP85]. It tries to produce several possible minimal cardinality set solutions. A set covering problem is typically stated along the following lines:

For a finite set  $S$  of elements and a family  $F$  of subsets of  $S$ , a cover  $K$  of  $S$  from  $F$  is a subfamily  $K \subseteq F$  such that  $\cup(K) = S$ . A cover  $K$  is called minimum if its cardinality is as small as possible [RNW85]

GSC is described using the notation given in Table D.1.

#### D.1.1 GSC Pseudo code

Figure D-1 shows the pseudo code for the GSC [RNW85] using the notation defined in Table D.1. Figure D-2 gives the pseudo code for the *Genset* function used in the pseudo code for GSC.

Table D.1: Notation for Reggia's Generalized Set Cover Algorithm

Symbol	Description
d	disease
s	symptom
consequences(d)	set of symptoms of disease d
causes(s)	set of diseases causing s
D	set of all known diseases
M	set of all possible symptoms
$C \subseteq D \times M$	relation with domain(C) = D and range(C) = M
$M^+ \subseteq M$	distinguished subset of M
$P = \langle D, M, C, M^+ \rangle$	diagnostic problem
order(P)	is the cardinality of an explanation for $M^+$

```

1. function Solve[P]
2.   variables n integer, S generator-set; /* initialize  $n \leq \text{order}(P)$ 
            $s = \phi$  indicates unknown solution */
3.   begin
4.     n:=0; S:= $\phi$ ;
5.     while S= $\phi$  do
6.       begin
7.         S:= Genset[causes( $M^+$ ),  $M^+$ , n]; /* s is assigned  $\phi$  if  $n < \text{order}(P)$ ,
           generator set for Sol(P) if  $n = \text{order}(P)$  */
8.         n:= n+1 /* increment n and try again */
9.       end;
10.    return S
11.  end.

```

Figure D-1: Pseudo code for Reggia's GSC Algorithm



```

1. function Genset[scope,manifs,n]
2.   variables I set-of-disorders, F G H generator-set;
3.   if n=0
4.     then
5.       if manifs =  $\phi$  /* check if there are no manifestations to cover */
6.         then return  $\{\phi\}$  /* if so, empty solution */
7.       else return  $\phi$  /* if not, n=0, so no solution is possible */
8.     else
9.       if |scope| < n
10.      then return  $\phi$  /* not enough causes to cover. so no solution */
11.      else /* recursively try to construct the solution */
12.        select  $d \in$  scope;
13.        I:=  $\{d' \in \text{scope} \mid \text{man}(d') \wedge \text{manifs} = \text{man}(d) \wedge \text{manifs}\}$ ;
14.        F:= Genset[scope-I,manifs,n];
15.        H:= Genset[scope-I,manifs-man(d),n-1];
16.        G:=  $\{H_i \cdot (I) \mid H_i \in H\}$ ; /*  $\cdot$  is a list append operation */
17.        return  $G \vee F$ 
18.      endif
19.    endif.

```

Figure D-2: Pseudo code for Genset used in GSC Algorithm

On line 12 of Genset, one disease,  $d$ , among the scope has to be selected. If there is a disease with a pathognomonic symptom, then it has to be selected because it will appear in all solutions. If not, the way we choose this disease will give solution that has minimal cardinality or one with lesser unaccountable symptoms. How  $d$  is selected for each of these cases is given below.

### D.1.2 Minimum Cardinality Set Solution

If there is a disease with a pathognomonic symptom, then it is selected. Otherwise, for minimal set solution  $d$  is selected as shown in Figure D-3.

$$d_{select} = d \in \text{scope s.t } |\text{consequences}(d) \cap |\text{manifs}| \text{ is maximum}$$

Figure D-3: Selection of disease/component for Minimum Cardinality Set Solution

The disease is selected such that more number of manifestations are accounted by the disease. This selection will always give the optimal solutions.

### D.1.3 Light Weight Set Solution

If there is a disease with a pathognomonic symptom, then it is selected. Otherwise, for light weight set solution  $d$  is selected as shown in Figure D-4.

$$\begin{array}{l}
 d_{select} = d \in \text{scope and } S = \{|unrequired(d) = 0\} \text{ s.t} \\
 \text{if } |S| \geq 1 \\
 \text{then } d \in S \text{ s.t consequences}(d) \text{ is maximum} \\
 \text{else } |unrequired(d, manif)| \text{ is minimum} \\
 \text{where } unrequired(d, manif) \text{ is defined as consequences}(d) - manif.
 \end{array}$$

Figure D-4: Selection of disease/component for Light Weight Set Solution

The disease is selected such that if there are diseases in the scope with no unrequired features, then a disease with more consequences is chosen among them. But if there are no such diseases, then we pick up the one with the least number of unrequired diseases. This is a greedy selection and may not necessarily produce optimal solutions <sup>1</sup> like in the case of minimum cardinality set solution.

---

<sup>1</sup>Dynamic Programming could be used for definitely getting optimal solution

# Bibliography

- [BOB97] *Business Objects DTF, Common Business Objects*. Object Management Group, 1997.
- [Boo94] Grady Booch. *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1994.
- [BP89] Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability, Volume 1, Concepts and Models*. Frontier Series. ACM Press, 1989.
- [CA93] S. Castano and V. De Antonellis. A constructive approach to reuse of conceptual components. In William B. Frakes Rubén Prieto-Díaz, editor, *Advances in Software Reuse*. IEEE Computer Society Press, 1993.
- [CFA97] *Common Facilites Architecture*. Object Management Group, 1997.
- [CH96] Gary Cornell and Cay S. Horstmann. *Core Java*. SunSoft Press/Prentice-Hall, 1996.
- [COM95] *The Component Object Model Specification*. Microsoft Corporation, 1995.
- [COS97] *CORBA Services: Common Object Services Specification*. Object Management Group, 1997.
- [DA96] Andy Dong and Alice M. Agogino. Text analysis for constructing design representations. In *Artificial Intelligence in Design*, 1996.

- [DCO98] *DCOM Architecture*. Microsoft Corporation, 1998.
- [dJ96] D. de Judicibus. Reuse: a cultural change. In Marjan Sarshar, editor, *Systematic Reuse: Issues in Initiating and Improving a Reuse Program*, pages 44–51, 1996.
- [FRC] <http://www.freecode.com/>.
- [GAM] <http://www.gamelan.com/>.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [Gog89] Joseph A. Goguen. Principles of parameterized programming. In Alan J. Perlis Ted J. Biggerstaff, editor, *Software Reusability, Volume 1, Concepts and Models*, Frontier Series. ACM Press, 1989.
- [HW93] Pat Hall and Ray Weedon. Object oriented module interconnection languages. In William B. Frakes Rubén Prieto-Díaz, editor, *Advances in Software Reuse*. IEEE Computer Society Press, 1993.
- [JBn97] *JavaBeans<sup>TM</sup>*. Sun Microsystems, 1997.
- [JCC97] *Java Compiler Compiler Manual*. Sun Microsystems, 1997.
- [JS97] *JavaScript Guide*. Netscape, 1997.
- [KRT89] Shmuel Katz, Charles A. Richter, and Khe-Sing The. Paris: A system for reusing partially interpreted schemas. In Alan J. Perlis Ted J. Biggerstaff, editor, *Software Reusability, Volume 1, Concepts and Models*, Frontier Series. ACM Press, 1989.
- [Lam86] Leslie Lamport. *A Document Preparation System, L<sup>A</sup>T<sub>E</sub>X, User's Guide & Reference Manual*. Addison-Wesley Publishing Company, 1986.

- [LM89] Steven D. Litvintchouk and Allen S. Matsumoto. Design of ada systems yielding reusable components: An approach using structured algebraic specification. In Alan J. Perlis Ted J. Biggerstaff, editor, *Software Reusability, Volume 1, Concepts and Models*, Frontier Series. ACM Press, 1989.
- [MS93] N.A.M Maiden and A.G. Sutcliffe. People-oriented software reuse: the very thought. In William B. Frakes Rubén Prieto-Díaz, editor, *Advances in Software Reuse*. IEEE Computer Society Press, 1993.
- [MV97] F. Peña Mora and S. Vadhavkar. Augmenting design patterns with design rationale. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, 11(2), 1997.
- [NR86] Dana S. Nau and James A. Reggia. Relationships between deductive and abductive inference in knoweldgebased diagnostic problem solving. In *Expert Database Systems, Proceeding from the first International workshop*, pages 549+, 1986.
- [OMG] <http://www.omg.org/>.
- [ORA97] *ORACLE Magazine*, volume XI. Oracle, 1997.
- [ORB97] *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 1997.
- [PD89] Rubèn Prieto-Díaz. Classification of reusable modules. In Alan J. Perlis Ted J. Biggerstaff, editor, *Software Reusability, Volume 1, Concepts and Models*, Frontier Series. ACM Press, 1989.
- [PDF93] Rubén Prieto-Díaz and William B. Frakes, editors. *Advances in Software Reuse*. IEEE Computer Society Press, 1993.

- [PM94] Feniosky A. Peña-Mora. *Design Rationale for Computer Supported Conflict Mitigation during the Design-Construction Process of Large-Scale Civil Engineering Systems*. PhD dissertation, Massachusetts Institute of Technology, Department of Civil and Environmental Engineering, 1994.
- [Pri97] Chris Price. *CASPIAN manual*. 1997.
- [RNW83] James A. Reggia, Dana S. Nau, and Pearl Y. Wang. Diagnostic expert systems based on a set covering model. *International Journal on Man Machine Studies*, 37:227–256, 1983.
- [RNW85] James A. Reggia, Dana S. Nau, and Pearl Y. Wang. A formal model of diagnostic inference. 1. problem formulation and decomposition. *Information Sciences*, 37:227–256, 1985.
- [RNWP85] James A. Reggia, Dana S. Nau, Pearl Y. Wang, and Yun Peng. A formal model of diagnostic inference. 11. algorithmic solution and application. *Information Sciences*, 37:257–285, 1985.
- [Sar96] Marjan Sarshar, editor. *Systematic Reuse: Issues in Initiating and Improving a Reuse Program*. Springer, 1996.
- [TNF] <http://www.10fold.com/>.
- [Tra93] Will Tracz. Lileanna: A parameterized programming language. In William B. Frakes Rubén Prieto-Díaz, editor, *Advances in Software Reuse*. IEEE Computer Society Press, 1993.
- [Vad96] Sanjeev S. Vadhavkar. Augmenting design patterns with design rationale. Master's thesis, Massachusetts Institute of Technology, Department of Civil and Environmental Engineering, 1996.