Augmenting Design Patterns With Design Rationale

by
Sanjeev Sureshchandra Vadhavkar

B.Tech.(Honors) Civil Engineering
Indian Institute of Technology, Kharagpur (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
February 1997

© Massachusetts Institute of Technology 1997. All rights reserved.
Abstract

A precondition for reusability in software development is the existence of reusable resources. The number of reusable resources is limited owing to a definite lack of systematic methods for producing reusable information. It is this deficiency which prevents effective software reuse within a collaborative environment. To address this problem, a new methodology for supporting the collaborative development of software applications by a group of software developers from a library of building block cases has been proposed. The new methodology focuses on the use of an explicit software development process to capture and disseminate specialized knowledge that augments the description of the software cases in a library. The importance of preserving and using this knowledge has become apparent with the recent trend of combining the software development process with the software code. There are three main components in the new methodology: design patterns, design rationale and case-based reasoning. Design patterns have been chosen to represent, record and reuse the recurring design structures and associated design experience in object-oriented software development. The Design Recommendation and Intent Model (DRIM) was used in the proposed methodology to capture the specific implementation of reusable software systems. DRIM provides a method by which design rationale from multiple software designers can be partially generated, stored and later retrieved by a computer system. To address the issues of retrieval, indexing and adaptation of project information, the methodology used case-based reasoning mechanisms for reusing software from previously developed projects taking into account the difference in contextual information. The present work also details an initial proof-of-concept prototype based on the new methodology. The proposed methodology is expected to provide a strong information base for software understanding by supporting non-obtrusive capture as well as effective access of design rationale information regarding the software development process.
Acknowledgments

I would like to acknowledge the support received from the Charles Stark Draper Laboratory (CSDL). Funding for the present work comes from CSDL IR&D Project Number DL-H-4847757.

I would like to thank my thesis supervisor, Feniosky Peña-Mora for his support, guidance and a constant source of encouragement since my first day at MIT. Feni, thanks for everything!

The technical assistance from Dr. Michael J. Ricard, Senior Member of the Technical Staff at CSDL, is greatly appreciated. He proved that there is a large community of Emacs users in the industry. Mike, thanks for the Emacs macros!

It goes without saying that this research would not have been possible without the support of numerous people. In particular, the DaVinci-ites: James Kennedy Hong Chen, Lucio Soibelman, Karim Hussein, Wissam Ali-Ahmad, Yee-Sue Chiou, Chun-Yi Wang and Siva Dirisala, have contributed their magic, faith, time, energy, vision, passion, support and friendship in an important and appreciated way.

And most importantly, I am very thankful to my family: mother, father, sister and brother-in-law, for providing constant emotional support and encouragement and tolerating my carefree attitude during my stay at MIT.
Contents

1 Introduction .................................................. 9
   1.1 Basic Concepts of Software Reusability .................. 11
      1.1.1 Reusable Information ............................... 11
      1.1.2 Code Fragments .................................. 12
      1.1.3 Logical Structures ............................... 13
      1.1.4 Functional Architectures ........................... 13
   1.2 Domain Analysis ........................................ 14
      1.2.1 A Context for Domain Analysis in Software Reuse ...... 14
      1.2.2 Domain Analysis Process ........................... 16
   1.3 Reuse and the Software Lifecycle ........................ 18
   1.4 Thesis Scope ............................................ 19
   1.5 Roadmap ................................................ 21

2 Background .................................................... 22
   2.1 Difficulties In Achieving Reusability At Code Level ...... 23
   2.2 Case Studies ............................................. 25
   2.3 Survey of Current Design Rationale Models and Systems .... 26
   2.4 Experiences of Design Pattern Based Systems ............... 31
   2.5 Chapter Summary .......................................... 32

3 DRIMER ........................................................ 34
   3.1 Overview of Building Components .......................... 34
      3.1.1 An Overview of Design Patterns ....................... 36
3.1.2 An Overview of DRIM ........................................... 43
3.1.3 An Overview of Case-based Reasoning Systems ................. 46
3.2 Combined Methodology for Software Reusability .................. 47
  3.2.1 Design Patterns for Software Reusability ..................... 51
  3.2.2 Using DRIM for Software Reusability ........................ 53
  3.2.3 Design Recommendation and Intent Model Extended to Reusabil-
          ity ................................................................. 55
  3.2.4 Role of Case-based Reasoning in Software Reusability ........ 57
3.3 Chapter Summary .................................................. 59

4 System Architecture ............................................... 60
  4.1 Framework ....................................................... 62
    4.1.1 User Interface .............................................. 63
    4.1.2 Middle Layer ............................................... 64
    4.1.3 Back End .................................................. 65
  4.2 Implementation ................................................ 66
  4.3 Chapter Summary ................................................ 68

5 Prototype Example .................................................. 69
  5.1 Web Interface .................................................. 70
  5.2 Emacs Interface ................................................. 80
  5.3 Integrated Design Environment ................................ 80
  5.4 Code Evolution ................................................ 84
  5.5 Chapter Summary ................................................ 84

6 Conclusions and Future Work ....................................... 88
  6.1 Research Summary .............................................. 88
  6.2 Future Work ..................................................... 90

A Design Patterns Template .......................................... 92

B OODesigner .......................................................... 95
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Hierarchy Of Software Development adapted from Freeman, 1989</td>
<td>12</td>
</tr>
<tr>
<td>1-2</td>
<td>Domain Oriented Reuse adapted from Prieto-Díaz and Arango, 1991</td>
<td>14</td>
</tr>
<tr>
<td>1-3</td>
<td>Context Diagram for Domain Analysis adapted from Prieto-Díaz and Arango, 1991</td>
<td>17</td>
</tr>
<tr>
<td>1-4</td>
<td>Roles of Reuse adapted from Freeman, 1989</td>
<td>18</td>
</tr>
<tr>
<td>2-1</td>
<td>Comparison of Design Rationale Models for Reuse</td>
<td>27</td>
</tr>
<tr>
<td>3-1</td>
<td>Design Patterns Relationships</td>
<td>37</td>
</tr>
<tr>
<td>3-2</td>
<td>Design Patterns Model With Their Intents</td>
<td>38</td>
</tr>
<tr>
<td>3-3</td>
<td>Prototype Design Pattern for drawing a structural frame</td>
<td>40</td>
</tr>
<tr>
<td>3-4</td>
<td>Builder Design Pattern for a Text Converter</td>
<td>40</td>
</tr>
<tr>
<td>3-5</td>
<td>Design Recommendation and Intent Model</td>
<td>44</td>
</tr>
<tr>
<td>3-6</td>
<td>The Case-based Reasoning Cycle</td>
<td>46</td>
</tr>
<tr>
<td>3-7</td>
<td>Software Development Scheme</td>
<td>48</td>
</tr>
<tr>
<td>3-8</td>
<td>Basic Cycle in Reusable Software Development</td>
<td>49</td>
</tr>
<tr>
<td>3-9</td>
<td>Structure and Implementation of Abstract Factory</td>
<td>54</td>
</tr>
<tr>
<td>3-10</td>
<td>DRIMER: Design Recommendation and Intent Model Extended for Reusability</td>
<td>56</td>
</tr>
<tr>
<td>4-1</td>
<td>Hypertext Example</td>
<td>62</td>
</tr>
<tr>
<td>4-2</td>
<td>System Architecture</td>
<td>63</td>
</tr>
<tr>
<td>4-3</td>
<td>Functions of the Prototype Middle Layer</td>
<td>64</td>
</tr>
<tr>
<td>5-1</td>
<td>Flow Control for the Prototype</td>
<td>70</td>
</tr>
</tbody>
</table>
5-2 Prototype Home Page ........................................ 71
5-3 Searching Options ........................................... 72
5-4 Searching the Intent Field .................................... 73
5-5 Search Results .................................................. 75
5-6 Abstract Factory Home Page ............................... 76
5-7 Structure of Abstract Factory ............................... 77
5-8 Different ways to add to the component repository .......... 78
5-9 Adding to the augmented code flat file system ............... 79
5-10 Emacs Interface: Adding Software Components .......... 81
5-11 Emacs Interface: Search Results ........................... 82
5-12 C++ skeleton code generation in OODesigner ............ 83
5-13 Integrated Design Environment ............................. 85
5-14 Start of Code Evolution ...................................... 86
5-15 Capturing Code Evolution .................................... 87

B-1 OMT Models in OODesigner™ ............................... 96
B-2 Pseudo Code Generation in OODesigner™ ................. 97
B-3 Advanced OMT Modeling in OODesigner™ .................. 98
B-4 Code Documentation in OODesigner™ ...................... 99
List of Tables

1.1 A Framework For Reusability Technologies adapted from Biggerstaff and Perlis, 1989 ................ 11

2.1 Potential Productivity Improvement Software/Life Cycle adapted from Biggerstaff and Perlis, 1989 ................. 24
Chapter 1

Introduction

I would like to see the study of software components become a dignified branch of software engineering. I would like to see standard catalogs of routines classified by precision, robustness, time-space requirements and binding time of parameters

M.D. McIlroy, 1969.

It has been argued that at present, the software development process is artifact oriented [32]. As a complex software system evolves, its implementation tends to diverge from the intended or documented design model. Without using design rationale information, such undesirable deviation makes the computer system hard to understand, modify and maintain. It has been estimated that up to 70% of life cycle costs in a software development process are incurred in the maintenance phase, and that up to half of system developers’ resources are taken up with reverse engineering designs in order to make appropriate changes [32]. Reusable software offers a paradigm-shift from this specify-build-then-maintain life cycle assumed in the past. Design of reusable software involves the application of a variety of kinds of knowledge about one software system to another software system. The reused knowledge includes concepts such as domain/context knowledge, development experience, design decisions, design history, code and documentation. Reusing this knowledge results in a design process offering considerable saving in both time and cost, while developing, running and maintaining the current software system. The emphasis in current software projects is on gener-
ating and tracking design artifacts like software requirements, design specifications, software prototypes, documentation and the final code, throughout the life cycle of the software project. However, the process by which these artifacts are designed and the reasoning underlying the design decisions (i.e., design rationale) remains implicit and is often forgotten until the time of reuse. This information on design rationale exists in obscure forms like design notebooks, minutes of design reviews or designer’s memory. Consequently, it is difficult to recover and reuse this vital information. Platform (heterogeneous hardware and software systems), topology (distributed systems) and evolutionary (rapidly changing constraints) requirements in present-day software applications put an additional burden on the designer to capture and maintain the design rationale in order to cope with the complexity introduced by the new requirements.

The technologies that are applied to the reusability problem can be divided into two major groups depending upon the nature of the components being reused. These groups are composition technologies and generation technologies. Table 1 shows a framework for classifying the available technologies [3]. In composition technologies, the components to be reused are largely atomic, and ideally, unchanged in the course of the reuse. Deriving new programs from building blocks is a matter of composition. Generation technologies are not as easy to characterize as composition technologies, because the components that are being reused are not concrete, self-contained entities. The components being reused are often patterns woven into the fabric of a generator program. The resultant structures often bear only distant relations to the original patterns. Further, each resulting instance of such a pattern may be highly individualistic, bearing only slight resemblance to other instances generated from the same seed pattern. In this case, reuse is less a matter of composition than of execution.
1.1 Basic Concepts of Software Reusability

Although the idea of reusing software programs is quite old and has become an important aspect of the practical field of computer utilization, the current popular revival of the idea has yet to produce any significant agreement of the term software reusability. It is important to consider following three main questions: What is being reused? How is it being reused? What is needed to enable successful reuse?

1.1.1 Reusable Information

To offer a broad definition of what is being reused in software engineering, consider the following observations. First, most of the economic motivations for reusability in programming relate to a reduction in the labor component necessary to bring into existence a new software system. Second, when defined narrowly, programming is only a small portion of the cost of creating a system. This leads to broadly define the object of software reusability to be any information which a software designer may need in the process of creating software systems.

This definition is far too broad to help understand and push forward specific research. To narrow the scope of interest, Figure 1-1 presents a hierarchy of types of information that software developers typically need. The idea is to provide a concrete context in which to think about reusability [7]. The labels on the right side of the Figure 1-1 indicate levels of information. In the following sections, each layer is defined and the types of information contained within it. External knowledge.

---

### Table 1.1: A Framework For Reusability Technologies

adapted from Biggerstaff and Perlis, 1989

<table>
<thead>
<tr>
<th>Feature</th>
<th>Approaches To Reusability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Composition</td>
</tr>
<tr>
<td>Component Reused</td>
<td>Building Blocks</td>
</tr>
<tr>
<td>Nature of Component</td>
<td>Atomic and Immutable</td>
</tr>
<tr>
<td>Emphasis</td>
<td>Application Component Libraries</td>
</tr>
</tbody>
</table>
and environmental level information shown in the figure pertain more to the domain analysis side of software reuse and are covered in Section 1.2.

![Hierarchy Of Software Development](adapted from Freeman, 1989)

**Figure 1-1: Hierarchy Of Software Development adapted from Freeman, 1989**

### 1.1.2 Code Fragments

Executable code is often viewed as the primary product of the programmer and its effective reuse is one of the oldest objectives of software technology. However, it is also a very narrow definition of software reuse and has shown not to offer a very large return on investment [3]. Examples of narrow views of reuse are “Reuse is the re-application of code”, “Reuse is the use of subroutine or object libraries” or “Reuse of C++ packages”. All of these views center around the reapplication of code components. Building systems out of small code components leaves a lot of work to be done in building the architectural superstructure that binds the components into a whole system. The cost to build this superstructure is typically much larger than
the savings afforded by reusing a set of small components. While the reuse of a code component *implicitly* involves the reuse of analysis and design, great opportunity is missed by not being able to reuse *explicitly*, the associated design and analysis information such as design structures, domain knowledge and design decisions.

### 1.1.3 Logical Structures

This level of information normally consists of both the process and data architectures of the software. These architectures are characterized by their identification of the parts (modules and data collections) and the relationships between them (calling, parameter passing, inclusion). These architectures are also characterized functionally and semantically, but are not tied to any specific implementation. This is often called the *internal design* of the system. Reusability in the context of internal system design is the ability to convey the overall procedural structure and principal data designs of large complex software programs to those who wish to develop the same or similar functions in new environments. Hence, the essential idea of reusable design is that a particular application domain should be studied in a formal way and that the artifacts of this study should be used to design software systems that automate that domain.

### 1.1.4 Functional Architectures

The external design of a system includes those aspects seen by the user or others that are not familiar with its internal structure. This level of information is normally a specification of functions and data objects - a description of these objects without the details of how they are realized. This reuse of subsystem and package designs is the most powerful level of software reuse. At this level, it is the high level interfaces that are important to the reuser, not how they are supported. Design reuse provides a much richer approach to building systems and collection of systems than do current design techniques. Design-for-reuse efforts must determine how to provide a common base on which many common systems can be easily built. These efforts typically result in the embodiment of domain knowledge in a platform of software design,
1.2 Domain Analysis

Figure 1-2: Domain Oriented Reuse adapted from Prieto-Díaz and Arango, 1991

1.2.1 A Context for Domain Analysis in Software Reuse

Until recently, most research in providing computer support for software design has focused on issues concerning the synthesis and development of reusable software components [33]. The synthesis and development of reusable software components involves the difficult and time-consuming process of composing and combining parts of code from previous projects and getting the selected code validated by domain experts [39]. However it should be noted that building of reusable software components is
only a small part of the whole reuse effort. It is being realized that effective software reuse requires more than building easy to browse, well cataloged, convenient software components [31] and [17]. There are several issues in making a component based approach to software design work in practice. First, a reusable component must have a formal (but still human-comprehensible) specification to say what it does. No software designer can be expected to use a component without a complete understanding of its functionality. Secondly, the component should explicitly include the justification that the implementation is correct and efficient for that particular intent. Finally, programming languages and systems must support this approach to software architecture [38]. Hence, methodologies combining libraries of reusable code with models that capture design rationale need to be formalized. These methodologies would use catalogs of standardized software components and corresponding retrieval tools with models that capture and retrieve relevant design rationale in a computer understandable form. Thus, the goal of reuse research should be to establish a software engineering discipline based on such methodologies.

The capture and retrieval of design rationale falls under the broad premise of domain analysis. Domain analysis is the process of identifying and organizing knowledge about some classes of problems as well as the problem domain in order to support the description and solution of those problems [27]. Domain analysis, therefore, determines the selection of problem domains, the types of knowledge that must be captured and the kinds of representations used to make that knowledge explicit as well as reusable. Under this broad area of domain analysis, design rationale research, focuses primarily on developing effective methods and computer-supported representations for capturing, maintaining and reusing domain records such as design intents, design recommendations and design justifications.

The application of reusability techniques depends on a cache of previous problem descriptions, solutions or methods to derive the solutions. The existence of such reusable resources cannot be taken for granted. In practice, those who have attempted
to test the promises of reuse have learned a painful lesson: it is very difficult to identify the right kinds of reusable information. Furthermore, once such information has been identified, there are questions of representation, granularity, indexing and searching, that must be solved in order to reuse the information. In other words, an infrastructure for reuse must be in place before an organization developing software can start reusing. A software developer may ask, how can I systematically and reliably capture the information I need now and in the future, to maximize reuse in our software development process? Domain analysis provides one possible answer. Figure 1-2 shows the various interplays between domain oriented reuse [27]. As shown, vertical reuse involves the reuse of components within the same parent domain. On the other hand, subsystem reuse could be a typical example of horizontal reuse.

1.2.2 Domain Analysis Process

Domain analysis can be conceived of as an activity occurring prior to system analysis and whose output (i.e., a domain model) supports systems analysis in the same way that systems analysis output (i.e., requirements analysis and specification document) supports the systems designer's tasks. Domain analysis is a knowledge intensive activity for which no methodology or any kind of formalization is yet available. Domain analysis is conducted informally and all related experiences concentrate more on the outcome than on the process. The first step in domain analysis related activities is to find abstractions for groups of components. The attributes selected to characterize these abstraction should be relevant to the intended use of the components. Classification is the next step in domain analysis. Classification is the act of grouping similar components. It can be visualized as the product of integrating the knowledge inherent to the objects of collection with the knowledge of the domain where the classification is going to be used. Post domain analysis activities include encapsulation and producing reusability guidelines. During encapsulation, reusable components based on the domain model are identified and implemented. Figure 1-3 shows the context diagram for the domain analysis process. The domain analyst and the domain expert are two actors responsible for the domain analysis process. The domain analyst has
Figure 1-3: Context Diagram for Domain Analysis adapted from Prieto-Díaz and Arango, 1991
the procedural know-how on the domain analysis. A set of guidelines are used to extract relevant knowledge from the domain expert and from existing systems. The product of domain analysis is a collection of domain specific reusable components and domain standards used by software engineers for building new systems in that domain.

1.3 Reuse and the Software Lifecycle

COMPETING/COOPERATING INTERESTS

<table>
<thead>
<tr>
<th>Supply Side</th>
<th>Demand Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Developer</td>
<td>System Developer</td>
</tr>
<tr>
<td>Development-for-Reuse Expert</td>
<td>Development-by-Reuse Expert</td>
</tr>
</tbody>
</table>

Figure 1-4: Roles of Reuse adapted from Freeman, 1989

Reuse has been described earlier as a basic component of a disciplined approach to solving problems. Reuse-based software development consists of a supply side and a demand side [19]. The supply side provides generic domain products and a framework for reusing them. New products are continually incorporated into this framework. The demand side consists of ongoing system development, in which products and services provided by the supply side are used to build systems. Within the context of supply and demand sides, there are four engineering roles as shown in Figure 1-4. Two of the roles - the Domain Developer and the Development-for-Reuse Expert - perform the supply side functions; the other two - the System Developer and the Development-by-Reuse Expert - execute the demand side operations. Each role aims at a different
set of goals. The System Developer builds new systems based on the constraints defined by a particular set of software requirements. The Domain Developer identifies recurring patterns in previous and current systems and organizes this into a framework for reuse. Based on these patterns, the Domain Developer defines an integrated set of processes and tools - an environment - by which future systems may employ the knowledge gained through earlier systems. The Development-by-Reuse Expert has the function of identifying products and knowledge that are applicable to the present system, evaluating the issues involved in reusing this information and ensuring the successful incorporation of reusable products into the new systems, possibly by tailoring them to the new context. The Development-for-Reuse Expert is responsible for refining newly developed products so that they are reusable.

1.4 Thesis Scope

To provide a “complete solution” to the problem of reusable systems development is a massive research undertaking, dealing with methodological issues, development tools and environments, software classification and retrieval, and project management tactics and strategy [20]. The research conducted had a more refined objective, which was to integrate principles from the domains of design rationale, design patterns and case-based reasoning in intelligent software classification and retrieval for software reusability.

Design rationale research in reusable software engineering domain is concerned with developing effective methods and computer-supported representations for capturing, maintaining and re-using records of why software designers have made the decisions they have. The main goal of the research is to capture a designer’s expertise which rests on the knowledge and skills which develop with experience in a domain [1]. The challenge is to make the effort of recording rationale worthwhile and not too onerous for the designer, but sufficiently structured and indexed so that it is retrievable and understandable to an outsider trying to understand the design at a
later date. Software reuse involves the use of well documented and popular libraries of software components. Design patterns for software engineering are becoming popular in the object-oriented software community. Design patterns can be considered as descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. Design patterns came about as a way to help the object-oriented designer reuse designs that have been used in many different applications. The challenge in using design patterns in software development lies in capturing the transition from the basic structure of classes described in the pattern literature to the actual implementation in a working environment. To use libraries of software components efficiently, intelligent indexing and retrieval mechanisms are required. Most reuse schemes require some form of adaptation of existing libraries. Hence, case-based reasoning [14] principles were integrated in the research endeavor. Case-based reasoning is one of the fastest growing areas in the field of knowledge-based systems. Case-based reasoning systems are systems that store information about past cases in their memory. As new software problems arise, similar software cases from the past are searched out to help solve the problems. There are several kinds of challenges in bringing case-based systems to their full potential. There are technological issues that must be confronted to make case-based reasoning systems scale up, in size of library, in speed of case retrieval and in their ability to deal with multiple domains and so on. There are knowledge engineering issues like protocols of case library maintenance which need to be defined before the integration of case-based reasoning into reusable software engineering domain.

The general focus of the research can be visualized as the use of intelligent techniques for better management of the software development process. Specifically, the research focused on the following main goals:

1. To use an object model integrating reusable software libraries with explicit schemes of design rationale capture and retrieval.
2. To develop and test a methodology that combines design rationale, design patterns and case-based reasoning principles. The new methodology adheres to
a software development process and allows the recording and easy retrieval of valuable design rationale information.

3. To develop a prototype using the above mentioned methodology as its base and test it in an industrial setting for use as an integrated design tool for software developers working in the domain of reusable software engineering. The prototype was able to record and present the knowledge gained during the collaboration between software designers separated across geographic space (different locations) and time (different projects).

1.5 Roadmap

The following chapters cover in details the ideas put forward to address the challenges proposed earlier:

- Chapter 2 presents a background of the research domain.
- Chapter 3 presents an overview of the components in the proposed methodology. The model combining principles from the domains of design patterns and design rationale plus case-based reasoning principles is discussed in details.
- Chapter 4 is devoted to the system architecture for the prototype devised based on the above mentioned model.
- Chapter 5 presents a detailed example explaining the functionality of the prototype.
- Chapter 6 summarizes the findings and the research contributions and discusses possible further work. Limitations of the approach, logical extensions to the approach and alternative paths of exploration are discussed as well.
Chapter 2

Background

There are millions of particular solutions to any given problem; but it may be possible to find some one property which will be common to all these solutions. That is what a pattern tries to do.

Christopher Alexander in *The Timeless Way Of Building*

Design of software systems can be characterized in terms of prototype (or initial design) phases, expansion phases, consolidation phases and testing phases. While object-oriented analysis and design methods give valuable hints on how to design software systems, little support is given for reusing components or making components reusable. In an object-oriented setting, applications are ideally built by plugging together reusable components. For components to be truly reusable building blocks, they have to meet several requirements. In particular, precise descriptions of the components are necessary. The difficulties in reusability at the primary code level are outlined in Section 2.1. The ultimate responsibility of implementing reusable software lies with the software designers. Hence, case studies were conducted to discover needs and attitudes of software designers towards reusability. The findings of these case studies is outlined in Section 2.2. As explained in Section 1.2.1, to make component based approach to software design work, it is necessary to combine the libraries of software components with models that capture the reasoning process of the designers, i.e., design rationale. It was necessary to understand existing design rationale models and systems and analyze how these models could be integrated into the proposed
reusability domain. A survey of current design rationale models and systems in the reusability domain is presented in Section 2.3. As mentioned in Section 1.4, design patterns were chosen as libraries of reusable components. To support the applicability of design patterns, Section 2.4 presents research findings from the studies of design patterns based communication systems.

2.1 Difficulties In Achieving Reusability At Code Level

Until recently, most research in providing computer support for reusable software design has focused on issues concerning the synthesis and development of reusable software components. However, in order for the concept of reusable software to become a reality, the following problems must be successfully addressed [3]:

- **Mechanisms for identifying components**: How can one determine what components are generally useful and adaptable to many different projects?

- **Method for specifying components**: Once a functional component is decided upon, in what way does one write its description so that others can understand it?

- **The form of the components**: Should the components be implemented in some programming language or described by a program design language that permits one to intermix programming constructs with natural language description?

- **Testing of the components**: How should the components be tested so that they can be reused in different environments with the same or similar performance?

- **Component indexing and cataloging scheme**: Under the hidden assumption that there will be a fairly large number of software components, what kind of indexing scheme needs to be implemented so that the components can be searched and retrieved for reuse?

Table 2.1 outlines the basic life cycle phases of a software development process [3]. The first column gives the percentage of effort typically consumed by each activity.
Table 2.1: Potential Productivity Improvement Software/Life Cycle adapted from Biggerstaff and Perlis, 1989

<table>
<thead>
<tr>
<th>Feature</th>
<th>Current Cost</th>
<th>Improvement</th>
<th>Net Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Requirements</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Hardware Requirements</td>
<td>8</td>
<td>25%</td>
<td>6</td>
</tr>
<tr>
<td>Software Requirements</td>
<td>10</td>
<td>20%</td>
<td>8</td>
</tr>
<tr>
<td>Software Design</td>
<td>12</td>
<td>40%</td>
<td>7</td>
</tr>
<tr>
<td>Coding</td>
<td>13</td>
<td>75%</td>
<td>3</td>
</tr>
<tr>
<td>Unit Test</td>
<td>24</td>
<td>50%</td>
<td>12</td>
</tr>
<tr>
<td>Integration Test</td>
<td>13</td>
<td>30%</td>
<td>9</td>
</tr>
<tr>
<td>Documentation</td>
<td>6</td>
<td>30%</td>
<td>4</td>
</tr>
<tr>
<td>System Test</td>
<td>12</td>
<td>25%</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>40%</td>
<td>60</td>
</tr>
</tbody>
</table>

The next columns depict the improvement factor and the changed costs respectively. This table shows that the reuse of code alone will not have a major impact on the software development process as a whole. Because more time is spent on requirements, design and maintenance than on coding, hence these areas need to be addressed as well. A final difficulty in achieving reusability at the code level is that a user may want to change or extend the function of the unit to be reused. This may require extensive knowledge about the implementation of the module, thereby negating much of the advantage. If these changes are to be made over severable reusable components without prior knowledge, the virtues are severely diminished. The major conclusion is that a large improvement in any of the categories of Table 2.1 will not substantially improve total productivity. For example, attempts to improve the coding phase by designing more reliable programming languages are very limited in the extent to which they can affect the entire process. Traditional software design approaches based on specify-build-then-maintain principles are inadequate for influencing the overall software design process. To achieve major reuse, it is essential to target overall improvement in each of the categories mentioned in Table 2.1.
2.2 Case Studies

Effective integration of design rationale into the reusability endeavor, requires that information be recorded about why design decisions are made, why particular solutions were not undertaken, and why some solutions are accepted given certain constraints. There are several important issues in this context. These issues include the model, capture, and use of design rationale. For a better understanding of these issues, it was decided to conduct a set of case studies. There were two main goals in the case studies conducted as an initial step in the research endeavor. The main aim was to discover user needs and attitudes towards software reuse. Secondly, the case studies were undertaken for the representation and active computer support for capturing design rationale [22] and [24].

The survey of application programmers conducted to discover the needs and attitudes of software developers towards software reusability shows that software developers consider reuse worthwhile, but most of them expect more from application generators (for example, CASE tools) than from reuse systems like browsers. In the case study, designers acknowledged the potential for capturing design rationale for historical record, corporate memory and reusability. Designers were concerned about the time required to input such information and emphasized that the capture of design rationale must be unobtrusive. The designers also requested that the presentation of the design rationale should include not only computer decisions but also decisions made by designers to override computer decisions or to suggest courses of action not anticipated by the computer.

The design case studies also shed light on the software design process including the interactions among participants, the types of information used in design, how this information is used, and the iterative nature of the design process. The software design process is not performed by one designer but by several, who must interact and get feedback from each other. To improve the effectiveness of this interaction, not only is
it necessary to represent and manage software evolution, design intent evolution and their relationships (i.e., design rationale), but it is also important to use that information to provide active computer support for reusability. Design rationale provides a record of what each participant needs to satisfy. The findings in the case studies indicate that many issues of reusability could be more efficiently solved if designers were aware of all the requirements satisfied by the software components they want to reuse. Thus, a record of design rationale can be used to increase awareness of all the requirements in the software component library. This awareness can be achieved through active computer support in which the computer provides information through inference mechanisms and a set of established design patterns or standards, in contrast to passive support in which the computer just records information inputed by the designers [25].

2.3 Survey of Current Design Rationale Models and Systems

Capturing rationale has been a research topic for several decades [23]. There have been a number of models and systems developed by researchers in different application areas ranging from discourse [35], to engineering design [9]. Figure 2-1 shows a classification scheme based on the role of design rationale in reusable software design.

In Figure 2-1, the Y coordinate represents the number of software designers who are able to record their interacting rationale and are able to participate in the reuse scheme. The scale is divided into single and multiple designers. In other words, this parameter represents how the different models or systems handle the relationships of different software designers on generating software code. The X coordinate represents the computer support for recording and retrieving design rationale. The scale is divided into passive and active computer support. Passive computer support indicates that the computer helps the designer to store the rationale. The designer inputs the rationale in the computer and the system creates some links among the different com-
Figure 2-1: Comparison of Design Rationale Models for Reuse
ponents of the rationale. *Active computer support* indicates that the computer helps in recording the rationale by providing part of it. Such a scheme also helps in capturing design rationale while it is being generated at ongoing projects. The *Z* coordinate represents the reuse axis. The scale is divided into *passive* and *active* depending on the nature of the reuse information. *Passive* understandable reuse information implies that the design rationale was stored in a semi-structured form with names and links understandable only after human intervention. *Active* understandable reuse information implies that the design rationale was stored in an object-attribute-value form. The information was careful indexed to allow a computer to understand a part of the reuse rationale.

The *X* scale in Figure 2-1 is a continuous measurement with more computer support for design rationale capture as the boxes get farther away from the origin. The *Y* scale is discrete and there is no relation among the distances of the boxes to the origin. The *Z* scale is a continuous measurement ranging from mostly human understandable reuse mechanisms to the automated machine understandable ones.

In Figure 2-1, the *single designer-passive computer support for design rationale* quadrant has the designer’s notebook which represents the notes taken by the designer during the design process. This document is usually private and manually developed usually in the form of emails or notes taken during design reviews. It also has Rossignac *et al.*’s MAMOUR [28] and Cassotto *et al.*’s VOV [4] which keep a trace of the design as it evolves, but leave the design intent implicit in the trace. The idea behind these systems is that a sequence of transformations represents the design and captures some of the designer’s intent. Here, the transformations are operations performed on a model, and the sequence of these operations give the final product. Thus, it is believed that by recording that sequence, the product could be reproduced, if needed. One important point is that design rationale is defined as the operations that can re-create the product while intent is believed to be the operations performed. However, it is the position of the proposed research effort that design intents are more
than operations. Design intents also refer to objectives to be achieved which are not related to a specific task but to the comparison among design alternatives.

The *multiple designers-passive computer support for design rationale* quadrant has a series of research efforts from academia and industry. For example, Toulmin’s Model [35] and Kunz and Rittel’s Issue Based Information System (IBIS) in the Urban Planning domain [15]. This quadrant also includes systems catering to the reusable software engineering domain. For example, Potts and Bruns’ Model [26]; Conklin and Begeman’s Graphical Issue Based Information System (gIBIS) [5]; Lee’s Design Representation Language (DRL) [16]. Gruber *et al.*’s SHADE [10] system is an example of a design rationale capture system in the mechanical engineering domain for multiple designers, but providing passive computer only. A similar example in the Civil and Construction engineering domain include Favela *et al.*’s CADS [6].

It is important to note in this quadrant the ontology used by these systems. Their ontology lacks a representation and a structure for the process and the product as they evolve. Missing is the notion of artifact evolution\(^1\). Most of them concentrate on the decisions made but without any underlying model of the artifact. The artifact model is important because that is the product developed which connects the design to the resulting entity. This entity in turn guides all the subsequent design decisions. Also missing is the notion of classification of the intents (i.e., objectives, constraints, function, and goals), as well as the classification of the justifications for a proposal (i.e., rules, catalog entry and first principles) since they have different characteristics and are used different by the designers. Section 3-5 explains in more detail these classifications.

Models or systems in the area of reuse have focused primarily on creation of reusable libraries. The *multiple designer - passive computer support for design rationale - passive computer support for reuse* quadrant has a series of research efforts.

---

\(^1\)Artifact evolution refers to the evolution of a design from a general concept to a specific implementation.
These systems take designers’ options, evaluate them, and help the designers select the best option. The design patterns effort [8] provides an excellent library in the reusable software engineering domain. The design rationale in design patterns is closer to the human understandable form as the designer’s expertise is inherent in the patterns and is only a part of the description. However, the computer does not provide any support in generating some of these options and their accompanying preferences. Smith’s KIDS system [33] provides a semi-automatic program development system. Isakowitz et al.’s ORCA and AMHYRST [11] systems provide automated support for software developers who search large repositories for the appropriate reusable software objects. These systems do not consider design rationale during the software development process.

The multiple designers - passive design rationale capture - active computer support for reuse quadrant has research efforts like Nierstrasz’s Composing Active Objects [20] and Luqi’s CAPS project [18]. These research efforts concentrate on computer-aided rapid prototyping systems. These systems help in rapid prototyping in well-defined and well-understood domains. These systems do not pay much attention to the design rationale information generated during the software design process. In these systems, the role of design rationale is reduced to a mere off-product of the design process and the design rationale, if recorded at all, is in an unstructured form.

There is little or no documentation of research in the multiple designers - active design rationale capture - active computer support for reuse quadrant. Reusable software can best be accomplished with knowledge not only of what the software system is and what it does, but also why it was put together that way and why other approaches were discarded [13]. This makes this quadrant the one that needs most attention in order to satisfy reusable software requirements. Thus, a computer based model and a system for capturing and retrieving the rationale of software developers collaborating on a software development project is necessary.
2.4 Experiences of Design Pattern Based Systems

This section outlines important usability studies of design pattern based communication software systems presented in [30]. These studies confirmed some strengths of design patterns and revealed precautions that designers have to take during actual implementation of design patterns in software systems. Although these findings are specific to communication software systems, they can be generalized to other similar systems as well.

One of the key findings of these studies was that even if reuse of algorithms, implementations, interfaces or detailed designs is not feasible using design patterns, widespread reuse of software architectures is possible. Because design patterns abstract the properties of successful designs, they are not limited to a single implementation. Hence, any pattern description should include concrete examples. Software designers had a hard time understanding precisely how to implement patterns when they are described using only object diagrams and structured prose. By providing software developers with shared vocabulary and concepts, the intercommunication between the software designers is greatly enhanced. Patterns are validated by experience than by conventional testing schemes. Testing patterns meant a peer reviewed process of discussing the strength and weakness of each pattern. The use of a computer based tool to help these pattern review sessions between software designers, was greatly felt. The studies showed that successful integration of design pattern into a software development process was a human intensive activity. Identifying and documenting useful design patterns required software designers with substantial domain experience to make significant investments to make a visible difference. Documentation was seen to play an important role in effectively leveraging design patterns for reuse. The use of hypertext media such as HTML based browsers was noted in this context for creating compound documents that possess multiple levels of abstraction. Design patterns in these communication systems abstract behavior of a single process and let software developers interconnect these processes to form a complete system.
There are no higher level design patterns that described the structure and behavior of a collection of processes. Developers using design patterns based systems often feel restricted in the high-level model because the final implementation is not flexible enough to accommodate customizations.

2.5 Chapter Summary

Traditional software approaches are inadequate for influencing the design of large scale reusable software systems. To achieve major reuse, reusable component development must be viewed as a separate activity from customized-product development. Hence, a formal development cycle considering domain analysis and infrastructure implementation needs to be supported. Models of software related domains should be treated as separate entities, conceptually as well as practically. Potentially, there will be a demand for active computer-supported domain analysis methods in the reusable software engineering arena.

The experiences of using design patterns in the reorganization process are very encouraging. Research in the following areas can further increase the usefulness of the design patterns approach:

- Collecting generic and domain specific design patterns.
- Formalizing the semantics of the design patterns.
- Integration of design patterns with other design methods.

In the systematic reorganization process, the need for computer tools with the following functionalities was felt:

- Graphic browsing and editing tools which allow the users to consider design patterns as units in the design.
- Hypertext tools for documenting design patterns and applications.
- Analysis tools which uncover design flaws and suggest improvements.
Design patterns are revolutionizing the way object-oriented software developers are designing and teaching object-oriented systems. Patterns are increasingly becoming important tools for documenting successful practices and techniques in the software community. Over the next few years, a wealth of software design knowledge is expected to be captured in the form of design patterns that span application domains such as business and financial systems, client/server programming, network programming and human interface design.
A pattern-based derivation of an architecture is like the presentation of a mathematical theorem. Beginning with a problem to solve, you use well-known, independent steps to incrementally refine the problem to a solution. The result is that you not only understand the final system, you understand the reasoning that led to it. This should make it easier for programmers to use and extend systems documented with pattern-based derivations.

Kent Beck in ECOOP’94

3.1 Overview of Building Components

In order to satisfy the goals of this research in terms of an integrated methodology and a prototype based on the methodology, three main components were selected. Object-oriented systems often exhibit idiomatic and recurring patterns and structures of communicating objects that solve particular design problems and make design more flexible, elegant, and ultimately reusable. Design patterns [8] were chosen to represent, record and reuse this general concept of reuse (i.e., recurring design structures and associated design experience). The Design Recommendation and Intent Model (DRIM) [21] was used in the proposed methodology to capture the specific implementation of reusable software systems. Case-based reasoning principles [14] were used in the proposed methodology to index, retrieve and adapt past software cases into the
current implementation.

Design patterns capture and describe in a systematic and a general way the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context. Design patterns capture expertise in building object-oriented software. It should be noted that design patterns are not a software development method or process by themselves. They complement existing methods or processes. The novelty of design patterns in software engineering domain arises primarily from the fact that they constitute a preliminary and simple effort to build on the collective experience of skilled designers and software engineers. Such experts already have solutions to many recurring design problems. By capturing these proven solutions in an easily-available and well-written form, patterns hope to gain from the expertise of these software designers. A brief overview of the design patterns described in [8] is given in Section 3.1.1.

Design Recommendation and Intent Model (DRIM) [21] captures the design experience in a form that others can use effectively at a later stage and allows the application of the concept of design rationale in a software development scheme. DRIM provides a method by which design rationale information from multiple designers can be partially generated, stored and later retrieved by a computer system. It uses domain knowledge, design experiences from past occasions and interaction with designers to capture design rationale in terms of artifact evolution, intent evolution and the relationships among them. The Design Recommendation and Intent Model is described in Section 3.1.2.

Case-based reasoning systems [14] store information about situations in their memory. As new problems arise, similar situations are searched out to help solve these problems. Problems are understood and inferences are made by finding the closest cases in memory, comparing and contrasting the problem with those cases, making inferences based on those comparisons and asking questions when those in-
ferences cannot be made. Learning occurs as a natural consequence of reasoning where novel procedures applied to problems are indexed in memory. An overview of case-based reasoning systems is given in Section 3.1.3.

### 3.1.1 An Overview of Design Patterns

Design Patterns can be considered as descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. By describing a solution to a recurring design problem in a systematic and general way a design pattern in effect, names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. Design Patterns capture the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context. Design pattern can be used as mechanisms for matching information with knowledge from previously developed projects. They have proven popular in the object-oriented software community as it gives the designer a reference to proven design solutions along with guidance on how to implement them. The study of patterns is well established in many other fields including architecture, anthropology, music, and sociology. Early adopters of software patterns were highly influenced by Christopher Alexander’s work on patterns [2] found in architecture for houses, buildings, and communities. Specific objectives of design patterns in the domain of reusable software engineering are:

- Identify good design that maps the solution to the implementation.
- Explicitly specify how reusable classes relate to the design.
- Define the context in which the design patterns are valid.
- Explicitly specify key issues, assumptions, constraints and dependencies in prior designs.

The catalog of design patterns as described in [8] contains 23 design patterns. Figure 3-2 shows the design patterns described in the [8] along with their intents. Although the present research concentrates more on the design patterns described in [8], the catalog is by no means exhaustive. Design patterns are evolving at a fast pace
Figure 3-1: Design Patterns Relationships
Components of Software System

CREATIONAL
- Abstract Factory
  - Interface for creating families of related objects
- Builder
  - Separate creation of objects from their representation
- Factory Method
  - Allow a class to defer instantiation to subclasses
- Prototype
  - Create new objects using a prototypical instance
- Singleton
  - Ensure a class has only one instance and provide a global point of access

STRUCTURAL
- Adapter
  - Make interfaces of classes compatible
- Bridge
  - Decouple an abstraction from its implementation
- Composite
  - Compose objects into tree structures
- Decorator
  - Attach additional functions to an object dynamically
- Facade
  - Define a higher level interface
- Flyweight
  - Use sharing of large number of objects
- Proxy
  - Provide means to control access to an object

BEHAVIORAL
- Chain of Responsibility
  - Chain the receiving objects and pass request along the chain
- Command
  - Encapsulate a request as an object and support undoable operations
- Interpreter
  - Define a representation for the grammar of a language along with an interpreter
- Iterator
  - Provide a way to access objects sequentially
- Mediator
  - Define an object that encapsulates the object's interactions so that the object can be later restored
- Memento
  - Capture and externalize an object's internal state so that the object can be later restored
- Observer
  - Define a one-to-many dependency between objects so as to link the objects
- State
  - Allow the object to alter its behavior when its internal state changes
- Strategy
  - Allow the subclasses to define certain steps of an algorithm without changing its structure
- Template Method
  - Define a new operation without changing the classes on which it operates
- Visitor
  - Define a new operation without changing the classes on which it operates

Figure 3-2: Design Patterns Model With Their Intents
thanks to a growing interest in the object-oriented software community [8]. However, the model, methodology and prototype being developed will be able to incorporate new design patterns as they get developed and accepted by the software community. Design pattern relationships are shown in Figure 3-1. From Figure 3-1, it is clear that there is considerable interplay between the design patterns themselves. Any significant reuse approach based on patterns, will need substantial domain knowledge in related patterns, how they work together and what each one of them is trying to achieve in a given context (i.e., design rationale).

The catalog of design patterns contains 23 design patterns. Two criteria have been considered for classifying the design patterns. Design patterns have been classified into three types, based on their purpose or the function they play in object-oriented design. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility. Design patterns can also be classified according to their scope, specifying whether the pattern applies primarily to classes or objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static. Object patterns deal with object relationships which are dynamic as they can be changed at run-time. Creational design patterns are described in Section 3.1.1. In Section 3.1.1, structural design patterns are reviewed. Behavioral design patterns are discussed in Section 3.1.1.

**Creational Design Pattern**

The creational design patterns help to make a system independent of how its objects are created, composed and represented. They give a flexibility in what gets created and when it is created.

Five types of creational design patterns have been defined: Prototype, Builder, Abstract Factory, Factory Method, Singleton.

- **Prototype** specifies the kinds of objects to create using a prototypical instance
Figure 3-3: Prototype Design Pattern for drawing a structural frame

Figure 3-4: Builder Design Pattern for a Text Converter
and creating new objects by copying this prototype. Consider a graphic tool that draws frame structures. As shown in figure 3-3, the basic structure of the frame gets repeated successively in some form or the other. The idea is to create a new graphic, by cloning an instance of a graphic prototype.

- **Builder** separates the construction of a complex object from its representation so that the same construction process creates different representations. As shown in figure 3-4, the builder pattern can be used to convert a text format from a typical document editor, to other formats. The builder pattern separates the algorithm for interpreting a textual format from how a converted format gets created and represented.

- **Abstract Factory** provides an interface to create families of related or dependent objects without specifying their concrete classes. Abstract factory pattern can be used in generating windows under different types of operating systems like XWindows and SunView. An abstract base class can be created that defines the interface for creating objects that represent the various parameters such as size, location, font and color of the window. Concrete subclasses implement the interfaces for a specific system.

- **Factory Method** defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method design pattern is particularly useful in cases when the parent class cannot anticipate the class of objects it must create.

- **Singleton** ensures a class has only a single instance and provides a global point of access to it. For example, there may be many windows but only one window manager. The singleton pattern ensures that no other instance of the window manager is created and it also provides a way to access the window created by the window manager.

**Structural Design Patterns**

Structural patterns are concerned with how classes and objects are composed to form larger structures. There are two types of structural patterns depending on whether
they are applied to objects or classes. Structural class patterns use inheritance to compose interfaces or implementations. Structural object patterns describe ways to compose objects to realize new functionality. The design pattern catalog contains 7 structural patterns. Some of the structural design patterns are described below.

- **Adapter** converts the interface of a class into another interface thereby allowing classes with incompatible interfaces to work together. *Adapter* design pattern can be used effectively in a drawing application to combine interface elements such as menus, scroll bars with graphic objects such as line, circle, polygon.

- **Bridge** allows abstraction and implementation to vary individually. Consider the implementation of a window tool kit in different operating systems. An abstract class window can be defined with subclasses for different operating systems. For every kind of window, different subclasses will have to be generated to account for the operating systems. Using the *bridge* pattern, the window abstraction and its implementation are placed in separate class hierarchies. There is a class hierarchy for window interfaces and a separate hierarchy for operating system specific implementation.

- **Decorator** attaches an additional responsibility to an object dynamically. For example, a document editor should add properties like borders and scrolling facilities to the user’s document. Instead of inheriting these properties, the *decorator* design pattern encloses it as an object and forwards requests to the object.

- **Facade** provides an unified interface to a set of interfaces in a subsystem. Clients communicate with a subsystem by sending requests to the facade design pattern, which in turn forwards the requests to the appropriate subsystem objects.

**Behavioral Design Patterns**

Behavioral design patterns are concerned with the algorithms and the assignment of responsibilities between objects. Behavioral patterns describe patterns of objects and also patterns of communication between them. Behavioral class patterns use
inheritance to distribute behavior between classes. Behavioral object patterns use object composition. The design pattern catalog contains 11 behavioral patterns. Some of the behavioral patterns are described below.

- **Chain of responsibility** decouples the sending and receiving objects by allowing multiple objects a chance to handle the request. The request gets passed along a chain until one of the objects handles it. For example, in a menu driven help system, the user can obtain help on a specific topic. If the information is not available on that topic then the request is passed on to higher levels, until a more general help is available.

- **Iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. The iterator pattern provides the means to access a list of employees without going through the internal structure of the list.

- **Memento** captures and externalizes an object’s internal state so that the object can be restored to its state later. Memento design pattern proves useful in implementing undo mechanisms. In such cases, the object can be restored to its earlier state.

- **Observer** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. For example, in a spreadsheet program, when the data is changed, the bar chart and pie chart diagrams change automatically.

### 3.1.2 An Overview of DRIM

In Figure 3-5, the Design Recommendation and Intent Model (DRIM) is presented [21]. DRIM uses the Object-Oriented Modeling Technique (OMT) described in [29] as a representation language. DRIM represents a software designer who can be either a human expert or a specific computer program. The software designer after collaborating with other designers, presents project proposals based on a design intent. The design intent refers to the objectives of the software project, the constraints in-
Figure 3-5: Design Recommendation and Intent Model
volved, the functions considered or the goals of the project. The software designer can present a number of different proposals satisfying a common design intent. The proposals presented can be either a version of a proposal or completely alternative proposals allowing for a representation of the dynamic nature of design in which solutions change or are refined. A given proposal may consist of sub-proposals allowing for a representation of the decomposition approach taken in a design process to reduce complexity. A proposal may react to an existing proposal by either supporting, contradicting or changing the ideas put in the existing proposal representing the argumentative nature of collaborative design. A project proposal includes the designer’s recommendation and the justification of why that particular proposal is recommended representing a relationship between a solution (i.e., artifact) and the problem (i.e., intent). The design recommendation can either introduce or modify a design intent, a plan or an artifact. When a design intent is recommended, it refers to more objectives, functions, constraints or goals that need to be satisfied in order to achieve the original design intent. The introduction of design intents and recommendations allows for a representation of the evolutionary nature of design in which problems and solutions are discovered as the design process proceeds. With a plan, more goals are brought into picture representing the relationships between the tasks to achieve and the end product. The artifact denotes the end product in a design process. An artifact has both behavioral and structural properties. The artifact comprises the software system as well as the components in the system. Justification explains why the recommendation satisfies the proposed design intent. A justification can be either a rule, (e.g., a suggestion from a past experience), or a case, (e.g., pertaining to similar performance in an existing software system), or a catalog, (e.g., from a standard library of classes), or a principle, (e.g., a set of equations you would like to use), or a trade-off, (i.e., the best design considering trade-off between two constraints), or a constraint-network, (e.g., satisfying all the systems constraint considered in proposing the design intent), or a pareto optimal surface, (e.g., the design falls on the surface of best possible design after considering many factors). A justification reacts to other justifications by either supporting or contradicting their claims. A context is the data
generated during the entire design process and consists of evidence and assumptions.

### 3.1.3 An Overview of Case-based Reasoning Systems

Case-based reasoning systems are knowledge-based systems that store information about situations in their memory. Figure 3-6 from [14] shows the various relationships between case-based reasoning’s primary processes. The primary process in any type of case-based reasoning system starts with *case retrieval*. In order to make sure that poor solutions are not repeated along with the good ones case-based reasoning system *evaluate* their solution. In the problem-solving role, case-based reasoning systems *propose initial solution* by extracting the solution from some retrieved case. This is followed by *adaptation*, the process of fixing an old solution to fit a new solution, and *criticism*, the process of critiquing the new solution before actual implementation. In the interpretive role, after *proposing initial solutions*, case-based reasoning systems *justify* the solution proposed by creating an argument in favor of the proposed
solution. As shown in Figure 3-6, the various processes are recursive and a successful implementation of a case-based reasoning system involves several loops between the various processes.

Case-based reasoning systems suggests a model of reasoning that incorporates problem solving, understanding, learning and integration with past cases [14] and [36]. The following premises underlie the model:

- Reuse of past software cases is advantageous.
- Descriptions of software problems are inadequate and hence a further understanding or re-interpretation of the software problem is a necessary prerequisite to learning which in turn helps the reuse of the past software cases.
- It is necessary to adapt the proposed solution as the past software case is not an exactly similar to the new software case. For example, even with closely matching intents, the final software code may be different depending on the context.
- Learning occurs as a natural consequence of reasoning.
- Feedback and feedback analysis along with reasoning are an integral part of the understanding/learning/reuse cycle.

### 3.2 Combined Methodology for Software Reusability

A design pattern only describes a solution to a particular design problem. Most software developers find it difficult to make the transition from the basic structure of a design pattern to the actual implementation. A design change might require substantial reimplementation, because different design choices in the design pattern can lead to vastly different final code. To address these needs, a methodology combining design rationale, design patterns and case-based reasoning principles is proposed. The following sections cover in details the possibility of using design patterns/augmented
code, DRIM and case-based reasoning principles for software reusability. First, an overview of how design patterns help in code reuse is presented in Section 3.2.1. In Section 3.2.2, the use of DRIM in design reuse is documented. DRIMER (Design Recommendation and Intent Model Extended for Reusability) is outlined as a means for achieving software reusability in Section 3.2.3. Role of case-based reasoning in software reusability is outlined in Section 3.2.4.

Figure 3-7 shows a simplified reusable software development scheme. The problem definition includes the requirements of the software system and the constraints on the final product. The design process is concerned with the design of the software system from the understanding of the problem to the code generation. The code is the final end-product of the design process. The justification for the code satisfying the problem definition is generated during the design process.

As shown in Figure 3-8, the envisioned basic cycle for the software process using the proposed methodology goes as follows:

1. The design process starts with the conception of various goals, objectives and functions of the software code. This is a part of the problem definition mentioned
Figure 3-8: Basic Cycle in Reusable Software Development
earlier in Figure 3-7.

2. Based on the requirements, the designers come up with a preliminary design satisfying some intents and taking into consideration some constraints. DRIM helps in capturing the transition from the conception to the preliminary design.

3. Next stage involves the retrieval of software designs from the database. Case-based reasoning helps in this stage by providing an insight into similar design cases and offers smart retrieval techniques based on advanced search mechanisms.

4. After retrieving relevant cases, the designers select a general method to achieve their intents. Design patterns provide a library to select the general method from a host of standard techniques. These design patterns help designers reuse successful designs by basing new designs on prior experience. DRIM helps in capturing the relationship between the preliminary design and the general method to be implemented.

5. From the general method and taking into consideration the design issues pertaining to the system at hand, the designers adapt a specific method. During this transition, the initial intents and constraints for the final software code could be revised to reflect changing view of the software product, thereby giving the whole software design process the characteristic iterative property. DRIM helps in capturing the design rationale in the evolution of the general method to the specific method. Case-based reasoning principles are used for describing what needs to be adapted from the existing software cases and how to bring about the adaptation.

6. Once the specific method is finalized, the designers go ahead with the specific implementation. The various issues in the specific implementation of the software code can be indexed and recorded using case-based reasoning principles based on DRIM constructs.

7. The software code is the final product of the design stage. It satisfies most of the initial and revised requirements and takes into account the various constraints. DRIM explicitly presents the justification, that the code generated satisfies the
requirements of the problem definition. The evolution of the code is captured in the above mentioned steps.

8. To reuse the newly generated code requires a well thought out indexing mechanism. After the new code is indexed and stored along with previous cases, it is ready for reuse\(^1\).

The earlier sections presented an overview of the design patterns, Design Recommendation and Intent Model and the case-based reasoning principles. It is essential to concentrate more on the role these building blocks play in the context of reusability. Section 3.2.1 shows how design patterns achieve some aspects of software reusability. The use of Design Recommendation and Intent Model in the reusable software engineering domain, is covered in Section 3.2.2. Role of case-based reasoning in software reusability is discussed in Section 3.2.4.

3.2.1 Design Patterns for Software Reusability

Maximizing software reuse lies in anticipating new requirements and changes to an existing model. Design patterns let some aspects of a model vary independently of other objects. In this way the model is made more robust to a change and redesigning is rendered unnecessary. Some cause of redesigning along with the design patterns that address them are described below:

1. Creating an object by specifying the class explicitly: A run-time creation of objects by explicit class creation commits the designer to a particular implementation. Creational design patterns like Abstract Factory, Factory Method and Prototype \([8]\) create object indirectly. This ensures that an object is committed to a particular interface as opposed to a particular implementation.

2. Dependence on specific operations: Instead of specifying a particular operation, behavioral design patterns such as Chain of Responsibility and Command \([8]\),

\[^1\text{A part of the current research endeavor involves working on specifying what kind of indexes are most useful, designating vocabularies for indexes and creating algorithms and heuristics for automating indexing choice.}\]
make it more easier the way a request is satisfied. These design patterns either
chain the receiving objects or encapsulate a request as an object. By doing
that, the design patterns ensure that it is easier to change the way a request is
satisfied both at compile-time and run-time.

3. Algorithmic dependencies: Algorithms are often extended or replaced during
reuse. Objects depending on an algorithm need to be changed when that hap-
pens. Creational design patterns such as Builder [8], avoid that by separating
the construction of an object from its representation. Behavioral patterns such
as Strategy [8], define and encapsulate a family of algorithms. This makes the
algorithms vary independently of the clients that use it.

4. Tight coupling: Tight coupling between classes leads to a systems which are
hard to reuse in isolation. Design patterns such as Abstract Factory, Facade
and Mediator [8] use techniques like abstract coupling and layering to promote
loosely coupled systems.

5. Extending functionality: Object composition and delegation offer flexible alter-
natives to inheritance for combining behavior. Design patterns such as Bridge,
Chain of Responsibility and Composite [8] introduce functionality by defining
a subclass and composing its instances with existing ones. Design pattern Ob-
server [8], extends functionality by defining a one-to-many dependency between
objects.

6. Inability to change classes conveniently: In some cases a class cannot be modi-
fied because the code is not available or may involve altering many subclasses.
Design pattern Adapter [8], converts the interface of class into the interface that
clients want.

While design patterns are useful in utilizing code reuse, they have some limitations
for achieving reusability. For instance, design patterns fail to keep a track of what
objects have been created and the reason why the objects had to be created. Design
patterns without explicit documentation fail to provide the software designer with
clear requirements and design alternatives that can help in solving the problem and
make the reuse effort worthwhile. There is a growing consensus, that simply providing
a library of reusable software artifacts is insufficient for supporting software reuse [31] and [17]. To make reuse worthwhile, the library of components should be used within well-defined and well-understood domains. To date, little research has been focused on the development of techniques for discovering workable patterns that can be captured, formalized, indexed and quantitatively evaluated. The change from design patterns as a general method to the adapted patterns as a specific method is lost during the design process. The design patterns fail to capture this important transition.

### 3.2.2 Using DRIM for Software Reusability

Design Recommendation and Intent Model can be considered as a framework of complimentary classes that make up a reusable design for a specific class of reusable software. Design Recommendation and Intent Model can be used for supporting the capture, modular structuring and effective access of design rationale information needed for building reusable software. By capturing the evolution of a particular software in a form understandable by the computer, DRIM taps the computer’s resources to record, represent and index the software process underlying the created product. DRIM is the key in the present effort to build comprehensive software environments which integrate design rationale much more tightly with the software code under development.

In addition, design patterns need validation by experience rather than by conventional testing. Design patterns are usually validated by periodic patterns reviews [30]. The mechanics of implementing design patterns is left to the programmer in the individual domain. The transition from a predefined general structure to a specific implementation needs to be validated and captured in a form understandable at a later stage. It is this transition that helps the designers in actual implementation of the reused software. To drive home the point, consider the following example from [8] describing the structure and motivation of the design pattern Abstract Factory (Figure 3-9). The example has been simplified to point the use of DRIM in validating the design patterns. The first box represents the OMT [29] notation of the various classes
Figure 3-9: Structure and Implementation of Abstract Factory
and their interactions in the design pattern Abstract Factory. The second box shows the implementation of the design pattern inside a usertoolkit program. The usertoolkit supports multiple look-and-feel standards (Motif and Presentation Manager). By using an abstract WidgetFactory class, an interface for creating each basic kind of widget is provided. Thus the design pattern suggests using an Abstract Factory class, for providing an interface for creating families of related or dependent objects without specifying their concrete classes. Designers employing design patterns directly would supply application-specific names for the classes and objects in the pattern. Software developers would then implement class declarations and definitions as described by the paradigms defined by the pattern. This is only half the work expected from a software designer. The true complexity lies in the fact that there are many trade-offs in a pattern to consider before the basic structure described by a design pattern can be converted to an actual implementation in the designer’s project. Different trade-offs could result in a proliferation of variant implementations leading to vastly different code. By capturing this transition and indexing it, DRIM provides a leverage to use the design pattern effectively at a later stage in a different domain. DRIM can also be used for providing better documentation of pattern reviews by capturing the strength and weakness of each pattern from past experience. Thus, DRIM provides a way to solve the shortcoming of design patterns in capturing the rationale of choosing the objects.

3.2.3 Design Recommendation and Intent Model Extended to Reusability

Figure 3-10 shows the Design Recommendation and Intent Model Extended for Reusability (DRIMER). In DRIMER, the artifact component in a software design context represents the components in a software system. A software designer presents a proposal which includes a recommendation and a justification. The recommendation introduces or modifies the components in a software system. The design patterns either create these components (Creational Patterns) or define their structure (Structural
Figure 3-10: DRIMER: Design Recommendation and Intent Model Extended for Reusability
Patterns) or define their behavior (Behavioral Patterns).

DRIMER allows for the explicit capture of design rationale during a software development process. If this design rationale is not captured explicitly, it may be lost over time. This loss deprives the maintenance teams of critical design information, and makes it difficult to motivate strategic design choices to other groups within a project or organization [30]. Identifying, documenting and reusing useful design patterns requires concrete experience in domain. By capturing the past experience, the combined DRIM-Design Patterns model offers essentially a mechanism to leverage patterns effectively. It is essential to note that design patterns by themselves are not the final artifacts of the software design process (i.e., software code). To leverage patterns, in effect, means deriving the code from the information inherent in the pattern description. DRIMER integrates the concepts of domain analysis and pattern analysis. The combination leads to the “patterns-by-intent” approach which refers to the process of selecting patterns based on their initial intents and then refining the choice of the pattern by specific constraints. This also accelerates the discovery and evaluation of new design patterns. The power of design patterns or any library approach derives from the reuse of components. DRIMER will achieve the same advantages of knowledge reuse and automation, but for a more general class of domains and for multiple modeling purposes.

3.2.4 Role of Case-based Reasoning in Software Reusability

The combination of reasoning and learning behavior in a case-based reasoning system and the ability of the cases to hold experience-acquired associative knowledge [36] provides a strong mechanism to leverage experience from the past. In general, reuse of past software cases serve three principal benefits:

1. Provide suggestions of solutions to the problems faced by the user.
2. Provide a context for understanding or assessing the solution.
3. Provide explicit justification of how the implementation solves the particular context.

In general, software designers tend to interleave the above mentioned processes according to the need of time. For example, in problem solving situations, software cases can be used to solve problems whereas in interpretive situations, the same cases could be used for criticism, justification and evaluation of the solution.

The various primary processes in a case-based reasoning approach to reusable software engineering can be itemized as follows:

**Case retrieval:** The primary goal in this step is to retrieve “good” cases that support the user’s primary interest. This involves partial pattern matching of intents from the cases in memory and coming with the ones closest to those of interest.

**Proposing a solution:** Relevant portions of the cases selected during the retrieval process are extracted and presented in a form easily understandable by a novice user as a first approximate solution. There are several issues that arise in proposing a solution to the initial query. The key issue is the question of how appropriate portions of an old case can be selected for focus. The DRIM constructs discussed in Section 3.1.2 provide a leverage to tackle this problem. They provide a common platform to dispense relevant information from various software cases.

**Adaptation:** In case-based problem solving, old solutions are used as inspirations for solving new problems. Because new situations rarely match the existing cases exactly, old situations need to be adapted to fit the new situation. To allow the user to adapt the proposed solution to the one that fits his/her specific query, the sample code, OMT diagram and DRIM constructs would be provided for effective adaptation.

**Evaluative Reasoning:** A solution to the user’s query needs justification to convince the user that the proposed solution is reasonably closer to his/her needs and has worked in prior cases. With the help of DRIM construct justification, the
prototype discussed earlier keeps a record of the justifications of the cases. This step also involves criticism of the proposed solutions. Criticism involves retrieval of additional cases and additional adaptation to propose new solutions.

*Case Update:* The case is evaluated either in the context of previous cases or on feedback from real world or on simulation. The new case is then stored appropriately in the case memory for future use.

### 3.3 Chapter Summary

Design patterns have revolutionized the way software developers think about software design and create object-oriented systems. However, it should be noted that design patterns should not be applied indiscriminately. A design pattern should be applied only in those cases when the flexibility it provides is needed. Design patterns, by themselves are not the final artifacts of the software design process. Software designers design software systems to generate software code as an artifact. Design patterns are recorded during this process and indexed for reuse in later projects. To derive the code for a new project from the information inherent in the pattern description, it is necessary to capture the documentation of the reasoning process of a designer in terms of the assumptions made, the constraints applied and the objectives satisfied during the creation of the design pattern itself. These requirements are addressed by formal object-oriented software design. DRIMER, the formal object-oriented model proposed, integrated the concept of domain and pattern analysis. By tightly integrating the basic principles of case-based reasoning system, DRIMER provides a strong mechanism to leverage experience from the past. Case-based system principles used in the model are helpful in adapting the past code components to the new situations.
Chapter 4

System Architecture

One of the principal objectives of research in any departments of knowledge is to find the point of view from which the subject appears in its greatest simplicity.

J. Willard Gibbs.

In order to operate successfully, a software reusability system must address five fundamental problems:

- finding components,
- understanding components,
- modifying components,
- composing components and
- discovering components and expanding the library.

The finding process involves more than locating an exact match. It includes locating “partially similar” components as well, because even if the new component has to be partially redeveloped, an example similar to the existing component will reduce the effort. The understanding process is required for all components as no designer can be expected to use a component without a complete understanding of its functionality. Furthermore, it is necessary to make this process as simple as possible to reduce the overhead associated with software reuse. The above issues make component understanding a fundamental problem in the development of any reusable software system,
regardless of the underlying technology chosen for implementation. One approach to this problem is represented by hypertext systems. Hypertext systems are tools for building a web of information that smoothly integrates text, graphical models and other information such as code. The most widely known example of one such system is the World Wide Web. The web also allows geographically separated software designers to share this web of information conveniently. Figure 4-1 shows such a system providing the user of the reusable component with instant access to a variety of supporting information. In the figure, the software requirements are the primary source of information. The design model based on the requirements can be viewed in varying details. The software code for each aspect of the model can be viewed along with the components used. The modifying process involves changing the static library of building components to a living system of software components that evolve as the requirements of the software system changes. Component composition process involves combining existing components to create a larger component for satisfying the requirements of the new software system. One of the main purpose of the composition process is to establish and maintain the integrity of the components throughout the life cycle of the software development process. Hence to make component composition feasible and useful, it is essential to maintain a mandatory version history with changes in various attributes explicitly described. Component discovery refers to the notion of a living component repository. To make reuse worthwhile, it is necessary to update the component repository. New components can be discovered during the domain analysis and application design processes along with the accompanying design rationale. The processes of finding, understanding, modifying, composing and discovering components have typically been an informal part of the software design process. A focus on software reusability requires that they be formalized so that computer supported tools can be formalized for the reusability problem. Section 4.1 describes in the details the top down view of the system model for handling each of the above mentioned processes. On the other hand, Section 4.2 describes the system implementation issues.
4.1 Framework

Investing in a software reuse tool is a long-term investment with potentially little short-term goal. It is important to note that there is no widely available, established reusable software application in the commercial domain. There are still several issues about standardizing software components which need to be tackled as well as general disagreement in the software reuse community in the whole as to the role played by reusable software tools.

From a technical standpoint, it would be desirable to be able to separate the individual components of a reuse library system into modular units that could be replaced as the system evolved. Thus, the component library should be relatively independent of the user interface. As shown in figure 4-2, there are three main system layers: User Interface, Middle Layer, Back End; and each is being covered in details in the next couple of sections.
4.1.1 User Interface

The user interface plays an important role in the whole reuse effort as it ensures the task of recording design rationale during the software design process is made as easy as possible for the software designer. The prototype provides two user interfaces for finding, understanding and modifying components from the library. The World Wide Web is increasingly becoming a popular user interface standard for multi-platform applications. The World Wide Web acts as a hypertext medium described earlier and provides the software designer with an interface to browse through existing components as well as add new components to the library. The Web interface also provides greater flexibility for the software designer in case of adding newly discovered components to the library. The example in Section 5.1 shows the full functionality of the Web interface. To simplify the tasks of finding and modifying existing components, a text based interface was also provided. In this case, the functionality of the user interface was brought to a popular text based editor. This text-based interface ensured
that the software designer could reuse existing components within his favorite design environment. The growing popularity of object-oriented programming languages has promoted the use of modeling techniques like Rumbaugh's Object Modeling Technique (OMT) [29]. To allow the user to manipulate object diagrams based on OMT, the text based user interface also included a CASE tool. Chapter 5 describes the text based editor and the CASE tool used in the prototype for reusing components.

4.1.2 Middle Layer

The middle layer performs a number of functions depending on the role of the prototype. As shown in figure 4-3, when the prototype is being used to find components, it uses an intelligent search mechanism to search for components from the index. The classification scheme for the components is important in accessing collections of components grouped semantically in some manner. The classification scheme employed along with the storage of components is covered in details in Section 4.1.3.

Figure 4-3: Functions of the Prototype Middle Layer
As shown in figure 4-3, when the prototype is being used for adding newly discovered components to the library, the middle layer performs the primary function of maintaining a complete record of the design rationale. In addition, the middle layer also indexes the newly added component based on a pre-determined classification scheme. The middle layer also generates the text output in multiple formats to ensure that the back end is accessible by both the user interfaces. This means that the text file needs to be converted into multiple HTML (Hyper Text Markup Language) files so that the vital design rationale information as well as the newly added component can be accessed with the Web interface.

When the prototype is being used to capture the evolution of the code during the component composition phase, the middle layer generates the necessary HTML files and also keeps a track of the various code versions to keep a track of the entire life cycle of the software. It is important to note here that the use of the prototype for capturing code evolution depends a lot on the accepter version control schemes. The version control scheme used in the present system is very similar to the one adopted in the RCS tool for UNIX systems [34]. In particular, the following constraints are imposed by the version control scheme used for the prototype:

- Upto 6 major revisions are allowed within the sublevel (for example, 1.0 to 1.6) until the next major level.
- Upto 3 minor revisions are allowed within the subsublevel (for example, 1.1a to 1.1c).

The example in Chapter 5 shows the use of the prototype in capturing code evolution.

### 4.1.3 Back End

There are many potential ways of organizing, browsing and accessing the information stored in a software component library. Three of the major options are: Flat File System, Database Management System and Artificial Intelligence Based Information Retrieval System. The Flat File System is the simplest of all - various component file
systems are essentially stored in the same format they are being used. A Database Management System is ideal for highly structured data like records with finite attribute values. Such a system is less suited for software component libraries as in this case the full text has to be stored. Artificial Intelligence based systems like Case-Based Reasoning provide the best solution in this scenario. These systems and their implications in this research are discussed in details in Section 3.1.3.

In the present prototype, the choice of the back end system was influenced by two factors: ease of operation and relatively few software components in the system. Flat File System was chosen as it provided the flexibility to store full text information in multiple formats. The presence of a middle layer in the system architecture also ensured that the Flat File System could be made independent of a particular browsing or searching capability.

The classification scheme was kept relatively simple keeping in mind the choice of the back end. The fields of this classification scheme included: the software component intent, the software component name and the software component location on the server. The choice of the software component intent in the classification scheme ensured that the software designer had access to the design rationale information of the component. The use of the software component name in the classification scheme was driven by the growing use of popular and widely accepted naming schemes being used in the software community. The location was important for indexing and retrieving purpose for the search programs. These fields and the full text information of the various components in the library was stored as a Flat File System.

4.2 Implementation

This section describes the implementation of a low-cost prototype based on the above mentioned architecture using freely available software systems. The prototype, like many other contemporary research efforts, has drawn widely on many Internet re-
sources. The entire system has been implemented for UNIX, using C++, PERL, X-Windows/Motif, Emacs Macros, OODesigner™ and Netscape™ with a SUN-SPARC 10 workstation acting as a server. The World Wide Web based user interface part of the prototype utilized the power of Web Forms and Web Gateways. Web Forms presented the user with several data fields to fill with a single submit button rather than just a single text entry field. The Web Gateway interface provided a method for clients to pass information to scripts residing on the server to provide information on the software components, text or images. The scripts were written in PERL [37] to perform the tasks of updating and querying the component library. PERL scripts were also written to parse and format the search results from the search program and create dynamic hypertext links to the references. The creation of text based interface involved the use of EmacsLISP for writing the macros which in turn called the PERL scripts. The choice of CASE tool for the prototype was greatly limited owing to a dearth in freely available CASE tools for the UNIX environment. OODesigner™ [12] is becoming increasingly popular in the academic environment. The OODesigner™ uses the OMT. (Object Modeling Technique) described in [29] to automatically generate C++ skeleton code\(^1\). The OODesigner™ source code was altered primarily to reflect the documentation format required for the prototype. To make an integrated text based interface, Emacs had to be made the default text editor for OODesigner™. The prototype primarily used two search techniques. C++ based search routines were written to handle the text based user interface querying. The primary focus of these routines was a fast and simple indexing mechanism for searches over a small number of fields in the Flat File System. The Web interface involved a much more advanced search program called Swish™ from Enterprise Integration Technologies\(^2\). Swish™ allows for wildcard searches along with boolean values for files stored in a Flat File System. The source code of Swish™ was modified to allow for synonym based searching. This feature, similar to the search features in case based reasoning systems ensured that the search program would allow for se-

\(^1\)The internal working of OODesigner™ can be found in [12].

\(^2\)A copy of Swish can be obtained from http://www.eit.com:80/goodies/software/swish/
mantically similar words to be grouped together. The code evolution functionality of the prototype was handled by PERL scripts which kept a track of the various versions of the code and created appropriate hyperlinks for the designer to browse through.

4.3 Chapter Summary

The system architecture has been designed to help in the reusability scheme by helping a software designer find, understand, modify, compose and discover software components. The proposed system uses a three tier client server mechanism using the Internet as the backbone. With the use of Internet, the interactions between the client (Web browser) and the server are left to the system implementation. Besides supporting the component library, the system architecture ensures the capture of code evolution and maintains the integrity of the components throughout the lifecycle of the process. The component library and the associated files in a file system on the server are kept relatively independent of the user interface. As the present system architecture was designed to handle relatively small number of software components, the flat file system chosen as the back end served the purpose. For a full scale implementation in an industrial setting there would be a need for migration to a database management system.
Chapter 5

Prototype Example

Software is often expected to provide a formal description of the very same properties for which mankind has yet failed to evolve and accept a satisfactory linguistic system..... We, the software makers, are contributing to the development of application domain theories by exposing, albeit in a painful and costly way, highly concentrated consequences of the inadequacies of available domain descriptions.

W. Turski, 1986.

Based on the system architecture described in Chapter 4, the prototype was developed to test the methodology. This chapter describes scenarios to illustrate how the prototype can be used in an object-oriented software development environment. The following sections detail the following functionalities of the prototype:

1. Web interface for adding, indexing and searching software components.
2. Emacs interface for adding, indexing and searching software components.
3. Integrated design tool for reusable software engineering.

Charles Stark Draper Laboratory (CSDL) provided the initial funding to develop and test an Internet based prototype using the methodology described earlier\(^1\). Fig-

\(^1\)The URL for the early prototype is http://ganesh.mit.edu/sanjeev/dp.html
Figure 5-1 shows a typical use of the prototype developed on the methodology discussed in earlier sections. The process begins with a software designer searching for a design pattern with a particular intent. From the matching design pattern, the designer tries to adapt the code into his/her project. As a final stage, the designer tries to add the completed code to the code database. Each of these stages is covered within the context of the user interface. The next couple of sections are arranged according to the user interfaces described in Section 4.1.1. The web interface for the prototype is explained with examples in Section 5.1. Section 5.2 is devoted to the Emacs interface with examples. The role of the prototype as an integrated design environment is covered in Section 5.3. Section 5.4 explains with examples to show the use of prototype for capturing code evolution.

5.1 Web Interface

Figure 5-2 denotes the key controls of the Web interface. As shown, with a few mouse sequences, the software designer can add as well as search the component repository. To illustrate the mechanism of searching the component repository using the web interface, consider a software designer searching the database for design patterns handling interfaces\(^2\). As shown in Figure 5-3, there are three options for searching. The software designer can search on the Intent field or on the Name field or alternatively use the advanced index based search mechanism. Design patterns are increasingly becoming popular in the object-oriented software community. To cater to software

\(^2\)An interface denotes the general boundary separating two classes in an object-oriented programming paradigm.
The DRIMER (Design Recommendation and Intent Model Extended to Reusability) research aims to explore the role of design rationale in achieving reusable software design. A framework is being proposed combining the Design Recommendation and Intent with a library of software components. This library is made of design patterns and code from free domains on the web. Software developed as a part of the Air Traffic Control Software from Draper Laboratory is also used in this library. The proposed framework will be used as an integrated design environment for reusable software design. It will support collaborative development of software applications by a group of software specialists from a library of software components. The effort also describes the use of an explicit software development process to capture and disseminate specialized knowledge that augments the description of the cases in a library during the development of software applications by a heterogeneous group. This specialized knowledge constitutes an important part of a software organization's memory. The importance of preserving and using this specialized knowledge has increased by the on-going practice of combining the software development process with the product (software code).

As a part of this effort, tools are being developed to ease the effort spent in reusing software. The Internet is being used as a backbone. This makes the entire effort platform independent. It also ensures that geographically spaced software designers can work collaboratively on a common software project. The web-based tools are also being integrated with Emacs macros which functions as a CASE tool.

Last updated by Sanjeev Vadhavkar – May 24 1996.

Figure 5-2: Prototype Home Page
Figure 5-3: Searching Options
Figure 5-4: Searching the Intent Field
developers who have had some exposure to design patterns and augmented code, the option of searching on the name field has been provided. For searching on the Intent field, the software designer accesses the appropriate Web page. He/she submits the required query on interfaces to the database search engine as shown in Figure 5-4. Figure 5-5 shows the search results. The search engine returns the pointer to the design pattern with matching intent (in this case Abstract Factory). The software developer can access more information on the design pattern by going to the web page of the pattern, as shown in Figure 5-6. The information on the design pattern is classified according to the widely accepted templates described in Gamma’s book [8]. These templates are described in details in Appendix A. With this information, the software developer can find out more about the applicability, motivation, the known uses of the design pattern and the justification for using that design pattern. He/she can also obtain the sample code and find out details about the implementation. Figure 5-7 shows the software designer finding about the structure of the Abstract Factory design pattern. A similar search on the name field can be made.

To illustrate the use of the web interface for adding software components and DRIM wrappers consider the following scenario: A software developer has developed software code and wants to add his/her code to the database as he/she thinks the code could be reused at a later stage. From the home page, the software developer can choose five ways to submit his effort (See Figure 5-8). He/she can either send a text file or an email or ftp the file to the home server or send a pointer to this information in the form of a URL or give the path name of the directory. All this information is recorded on the home server. In case the code to be submitted is already written in accordance with DRIM wrappers, he/she submits the formatted file. Figure 5-9 shows a designer adding the software code and information on various DRIM wrappers to the code database. In this case, the designer has chosen the option of inputting the DRIM wrappers using the web interface.
Figure 5-5: Search Results

The following design pattern matches your query:
http://ganesh.mit.edu/sanjeev/dbase/absfact.html

- Get more info from Design Pattern Database
- Return to Da Vinci Project Home Page
Figure 5-6: Abstract Factory Home Page
Figure 5-7: Structure of Abstract Factory
Figure 5-8: Different ways to add to the component repository
Figure 5-9: Adding to the augmented code flat file system
5.2 *Emacs Interface*

During the case studies, it became apparent that software designers were concerned about the time required to input design rationale information along with the software code into the reusable system. Previous reuse efforts have emphasized strong documentation skills. To provide an easier interface with the working environment of most software designers, it was decided to create an *Emacs* based interface for adding and searching the software components. *Emacs* macros were written to bring the feature of database addition and database access to a text editor. Figure 5-10 shows the *Emacs* for adding the code to the database directly from the *Emacs* window. Figure 5-11 shows the typical result after searching the code database. As shown in Figure 5-11, the designer has the choice of browsing through a software component with the matching intent. Additional utilities in the prototype ensure that the reuse effort takes only a fraction of the time taken from actual coding.

5.3 Integrated Design Environment

To provide active computer support to software designers working in the domain of reusable software, it is necessary to integrate this web based tool with the CASE tools used for software development. Combining the above mentioned Web based prototype with a commercial CASE tool will result in an integrated design environment for reusable software creation. In order to fully and beneficially implement reuse, the documentation effort essential for reuse has to be made as simple as possible for the software designer. As a first step towards this direction, the web based prototype were integrated with a freely available CASE tool named *OODesigner*™ [12]. The *OODesigner*™ uses the OMT (Object Modeling Technique) described in [29] to automatically generate C++ skeleton code\(^3\). The *OODesigner*™ source code was altered to reflect the documentation format required for the web prototype. After the designer finishes drawing the OMT model, the skeleton C++ code is generated

\(^3\)The internal working of *OODesigner*™ can be found in Appendix B.
Intent: a graphical interface for controlling positions on the screen

Name: draw_tool

Justification: based on basic mouse movements

RCS ID: $Id$

Author: Dan Brown

Email: dbrown@draper.com

Creation Date: 12-Jun-96

Modification History:

*/

/** Include Files */
#include <stdlib.h>
#include <stdio.h>

/** RCS ID */
static char rcsid[]="$Id$";

/** Table of Contents */

Figure 5-10: Emacs Interface: Adding Software Components
Figure 5-11: *Emacs* Interface: Search Results
Figure 5-12: C++ skeleton code generation in OODesigner
as shown in Figure 5-12. Figure 5-13 shows the functioning of the prototype as an integrated design environment. The software designer can reuse the actual code in the *Emacs* buffer. The OMT model can be viewed for modeling purpose using *OODesigner*\textsuperscript{TM}. Additional information about the reused code can be browsed on the web using the browser.

### 5.4 Code Evolution

The prototype can also be used to capture the code evolution and maintain the integrity of the reused software components throughout the life cycle of the process. Figure 5-14 shows the prototype at the start of the code evolution process. Figure 5-15 depicts a software designer using *Emacs* to add the next version of the software into the library. The *Netscape*\textsuperscript{TM} window shows the changes made by the prototype. The prototype creates the proper hypertext links and maintains the latest version of the software. A software designer could then collect all the relevant design rationale information by accessing the relevant web pages.

### 5.5 Chapter Summary

On the functionality issue, this chapter describes scenarios to illustrate how the prototype can be used in an object-oriented software development environment. On the implementation side, the prototype, like most contemporary research efforts, has drawn widely on many Internet resources. For example, *OODesigner*\textsuperscript{TM} was used as a CASE tool. *Swish*\textsuperscript{TM} was used as an index based search program. The various stages in reusable software development are covered with the help of detailed scenarios which highlight the following functionalities of the prototype:

- User interface for adding, indexing and searching software components.
- Integrated design tool for reusable software engineering.
- Web based tool for capturing code evolution.
Figure 5-13: Integrated Design Environment
Figure 5-14: Start of Code Evolution
Figure 5-15: Capturing Code Evolution
Chapter 6

Conclusions and Future Work

The DRIMER model and the initial prototype based on this model represents an initial attempt in building a prototype environment that can offer varying degrees of assistance to a software designer. By combining the fast evolving area of design patterns with design rationale principles, significant benefits are envisaged above the current software design practices. This chapter summarizes the findings and the research contributions and discusses possible further work. Limitations of the approach, logical extensions to the approach and alternative paths of exploration are discussed as well.

6.1 Research Summary

The research concentrates on building a design rationale based framework and a computational environment based on that framework to assist software designers in designing reusable software systems efficiently and reliably. The proposed framework allows for active computer support to multiple designers in the area of active design rationale capture for software reusability. The approach is based on abstracting the design as well as the process involved in designing a software system and reusing them to design new system. The computational environment functions as an assistant that delivers the relevant knowledge to a designer so that the designer may make more informed decisions during the design of the reusable software system.
The research described herein has significant technological and economical benefits to the modern software design process. The project envisions a paradigm shift from the specify-build-then-maintain life cycle assumed in the past to one of reusable software. Reusable software offers an economic relief to the change activity associated with modern software development wherein costs are incurred disproportionate to the size of the change. By supporting the capture as well as effective access of design rationale information, a strong information base for software understanding can be provided.

In summarizing this work, the contributions can be divided into two parts. First, a model for representing design rationale has been developed and tested. This model (i.e., DRIMER) provides primitives for representing design knowledge in terms of the reasoning process used by the designers to generate an artifact which satisfies their design intents. This model also takes into consideration the different collaborating designers and is used to provide active computer support for capturing designers’ reasoning process. In addition, the model allows human designers interacting with computers to record the interacting rationale of their design.

Second, the framework combining Design Recommendation and Intent Model with design patterns and case-based reasoning, offers active assistance to software designers in designing reusable software systems. Although the framework emphasizes the importance of documenting the software process, instead of laying extra burden on the programmers, it assists them by providing active computer assistance in recording the key design decisions. The framework acts as a software design tool that facilitates software reuse by:

- Using an excellent library of tested software components.
- Recording and allowing easy retrieval of decisions made during the software design process.
- Providing economic methods for systems by providing a context for design mod-
ifications when the requirements change over their life time.

- Providing a strong information base for software understanding.

6.2 Future Work

The prototype mentioned in Section 5 needs to be refined and tested in an industrial setting in a normal working environment where there are time pressures and organizational constraints. Exploration of how the designers use the system and cope with its limitations needs to be done. There is a need to integrate the prototype with a database consisting of large number of software code. The present number of design patterns and augmented code used for testing the prototype does not match the amount of software code written in any average size software project. For an implementation of the prototype in an industrial setting, the back end of the prototype will need an upgrade to a database management system (DBMS).

A design rationale is the representation of the reasoning which has been invested in the design. The prototype developed shows how computer support tools could augment design rationale capture process making it relatively simple with immediate benefits to the designer. Like all formalism, design rationale has political and organizational dimensions. Design rationale requires the adoption of particularly open style of design and accountability within an organization which may be considered by some to be idealistic. Design rationale faces the harsh realities of industrial settings where there is a distrust between software developers and programmers and the process of capturing design rationale could become intimidating. There is no doubt that an organizational culture shift towards corporate self-awareness and collective design process is necessary for effective use of design rationale. With such a shift, it will be possible to use design rationale as design-specific memory of the entire software organization. It would also be interesting to carry out research in producing a design rationale model which filtered and integrated design rationale information from a wide range of sources across different software projects.
There is a tight relationship between design patterns applied during system development and the resulting software architecture. At this stage, the development and validation of a predictive model for recording design pattern architectural impact would be of great interest. An initial step in this direction would be the architectural measurement of existing object-oriented systems where design patterns have been used extensively. This would result in a measurement base which is invaluable on its own but also from which a predictive model might be extrapolated. Such a model would also require to identify, analyze and classify the design pattern making the most impact in the entire software architecture. This would help software designers devote more energy in refining the design pattern as well as identify new patterns arising out of combination of existing design patterns.

To make the tool more versatile, additional issues in case-based reasoning like criticism of old cases and memory update need to be studied in detail and implemented in conjunction with the existing prototype. The methodology needs tighter integration with case-based reasoning principles whereby it will be possible to use the system for fast prototyping from existing software cases in the system’s database. Additional research needs to be done on mechanisms which will allow the addition of software cases on the fly to the database in the absence of a case master and allow the evaluation of these cases in the absence of algorithmic methods.
Appendix A

Design Patterns Template

How do we describe design patterns? To reuse the design, it is essential to record the design decisions, alternatives and trade-offs that led to a particular design. Design patterns have been described in a consistent format [8]. The template described below lends a uniform structure to the information, making design patterns easier to learn, compare and use:

Pattern Name and Classification

The name conveys the essence of the pattern and can be used as an identifier while sharing information amongst the various designers. The classification indicates whether the design pattern is object related or class related.

Intent

The intent relates to the particular rationale of the design pattern. It relates to a particular design issue or problem that the pattern addresses.

Also Known As

Other well-known names of the patterns are considered to avoid confusion within the design community.

Motivation

The motivation describes a scenario that illustrates the design problem and how the class and object structures in the pattern solve the problem. The motivation is aimed to help the designer understand the more abstract description of the
design pattern.

**Applicability**

Applicability refers to the design situations where the design pattern can be applied. It also serves as an aid for designers to recognize potential situations where the design pattern could be applied to jump over common design pitfalls.

**Structure**

The design pattern structure refers to the graphical representation of the classes in the pattern using the OMT notation described in [29].

**Participants**

A description of the classes and objects in the pattern drawn in the design pattern structure is included in this feature of the template.

**Collaborations**

A description of how the individual classes and objects in the design pattern collaborate and carry out their responsibilities.

**Consequences**

Consequences describe how the design pattern supports its intent. It also refers to the trade-offs and results of using the design pattern.

**Implementation**

The actual implementation issue of the design pattern are described in this feature of the template. This also includes language-specific issues which may become important during the implementation of the design pattern.

**Sample Code**

Code fragments that illustrate how the design pattern might be implemented help in reusing the design pattern in a different domain of interest.

**Known Uses**

Examples of patterns found in real applications help increase the design pattern’s credibility in the user community.
Related Patterns

In many cases, during the implementation of design patterns, it is noticed that the design pattern is related to other design patterns. By grouping the closely related design patterns together, designers are given a chance to compare and contrast the design patterns and understand the primary features of each one of them.
Appendix B

OODesigner

CASE tools provide active computer support to software designers working in the area of object-oriented programming by providing automated techniques to generate pseudo code from object models. To make the prototype function as an integrated design environment it was necessary to integrate the Web based prototype with a commercial CASE tool. Towards that end, the following commercial CASE tools were assessed: DevelopmentAssistant\textsuperscript{TM} by Ristanovic CASE, CRCTool\textsuperscript{TM}Version1.2 by Stan Mitchell, OOTherOOTool\textsuperscript{TM}Version1.6f by Roman M Zielinski and OODesigner\textsuperscript{TM} by Taegyun Kim. The OODesigner\textsuperscript{TM} was chosen for the following reasons: ease of integration with existing prototype, availability of source code and expected functionality in a UNIX environment.

Currently, OOD has following primary functions:

- A general graphics editor (with limited functionality)
- An object diagram layout based on the OMT notation
- C++ code skeleton generation (header file + source file). The comments and codes for individual member functions can be documented, or edited within OOD directly\textsuperscript{1}. The C++ code generator supports inheritance.

\textsuperscript{1}The original OODesigner\textsuperscript{TM} source code was altered, with proper permissions, to make Emacs the default text editor.
• Reverse engineering to make sure that OO methodology is a seamless and iterative process.

This part of the thesis is devoted to describe the various functions of *OODesigner*™ [12]. Figure B-1 shows the use of *OODesigner*™ for drawing complex object models using the Object Modeling Technique (OMT)[29].

Figure B-1: OMT Models in *OODesigner*™
Figure B-2 shows the $OODesigner^TM$ being used to generate pseudo C++ code and the associated header files for the classes drawn using OMT.

Figure B-2: Pseudo Code Generation in $OODesigner^TM$
In Figure B-3, the software designer updates the various member functions of the class to create a more detailed OMT model.

Figure B-3: Advanced OMT Modeling in OODesigner™
Additional code documentation can then be inputted directly by a software designer using the OODesigner™ as shown in Figure B-4.

Figure B-4: Code Documentation in OODesigner™
Bibliography


