Augmenting design patterns with design rationale

Article *in* Artificial Intelligence for Engineering Design Analysis and Manufacturing · April 1997

DOI: 10.1017/S089006040000189X · Source: DBLP

Augmenting Design Patterns with Design Rationale

Feniosky Peña-Mora[1] and Sanjeev Vadhavkar[2]

Abstract

Present-day software applications are increasingly required to be "reuse-conscious" in terms of the operating platforms, topology and evolutionary requirements. Traditionally, there has been much difficulty in communicating specialized knowledge like design intents, design recommendations and design justifications in the discipline of software engineering. This paper presents a methodology based on the combination of design rationale and design patterns to design reusable software systems. Design rationale is the representation of the reasoning behind the design of an artifact. Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. The paper details the use of an explicit software development process to capture and disseminate the specialized knowledge (i.e., intents, recommendations and justifications) that augments the description of the cases in a library (i.e., design patterns) during the development of software applications by heterogeneous groups. The importance of preserving and using this specialized knowledge has become apparent with the growing trend of combining the software development process with the product (i.e., software code). The importance of codifying corporate memory in this regard is also important considering the changing nature of the workplace, where more people are on contract. The information on how and why a software code was developed becomes essential for efficient and smooth continuity of the present software project as well as for reusing the code in future projects. It has become essential to capture the design rationale to develop and design software systems efficiently and reliably. The software prototype developed using the combined methodology will be used as a part of an integrated design environment for reusable software design. This environment supports collaborative development of software applications by a group of software specialists from a library of building block cases represented by design patterns.

1 Introduction

The software development process can be construed as a collaborative-iterative decision-making activity performed by a group of software professionals and non-professionals who have a stake in the final artifact. This activity is organized to conceive the idea, prepare the description, determine the plan of action by which resources are converted, and produce the end result. Here, one of the key characteristics of the process is its iterative nature. This iterative characteristic results from an inherently limited understanding of the design issues at the conception of the design process.

Design of reusable software involves the application of a variety of kinds of knowledge about one software system to another software system. Such a design methodology results in considerable saving in both time and cost, while developing, running and maintaining the current software system. The reused knowledge includes concepts such as domain/context knowledge, development experience, design decisions, design history, code and documentation. It has been argued that at present, the software development process is artifact oriented. 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 maintainers' resources are taken up with reverse engineering designs, in order to make appropriate changes [Shum, 1996]. The emphasis in a software project

[1]Assistant Professor, 77 Massachusetts Avenue, MIT Room 1-253, Intelligent Engineering Systems Laboratory, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: feniosky@mit.edu, Phone: (617) 253-7142, Fax: (617) 253-6324, WWW: http://ganesh.mit.edu/feniosky.html

[2]Research Assistant, 77 Massachusetts Avenue, MIT Room 1-270, Intelligent Engineering Systems Laboratory, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: sanjeev@iesl.mit.edu, Phone: (617)253-6232, Fax: (617)253-6324, WWW: http://ganesh.mit.edu/sanjeev/home.html

is on generating and tracking design artifacts like software requirements, design specifications, software prototype, documentation and the final code, throughout the life cycle of the process. However, the process by which these artifacts are designed and the reasoning underlying the design decisions remains implicit and is often forgotten until the time of reuse. This information 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. Until recently, most research in providing computer support for software design has focused on issues concerned with the synthesis and development of reusable software components [Smith, 1990]. The synthesis of reusable software components (i.e., design patterns) involves the difficult and time-consuming process of composing and combining parts of code to form a whole library of reusable code [Weide et al., 1996]. It is being realized that effective software reuse requires more than building an easy to browse, well cataloged, convenient software components [Shaw, 1990] and [Lubars, 1991]. Methodologies combining catalogs of standardized software components and corresponding retrieval tools with models that capture and retrieve relevant design rationale need to be formalized. The goal of reuse research should be to establish a software engineering discipline based on such methodologies.

The following sections cover in details the ideas put forward to address these challenges. Section 2 presents the research goals of this paper. Results from a set of case studies are given in Section 3. Section 4 provides a survey and comparison of research efforts on the capture of design rationale and reusability. An overview of the building blocks is presented in Section 5. In Section 5.1, design patterns are described in details. The Design Recommendation and Intent Model (DRIM) is presented in Section 5.2. The use of DRIM and design patterns for software reusability is explored in Section 6. A brief presentation on the software tools developed with an example to show their applicability in integrated software design environment, is given in Section 7. Section 8 presents a summary of the research findings and the contributions made in the present work. Finally, some ideas for future research are given in Section 9.

2 Research Goals

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 [Nierstrasz, 1993]. The present research has a more refined goal, which is to explore the role of design rationale in intelligent software classification and retrieval for reuse purpose.

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 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.

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. Each design pattern describes a problem which occurs over and over again in the software environment, and then describes the core of the solution to that problem, in such a way that the basic solution can be used many times over, without ever doing it the same way twice. Design patterns have made a significant impact on object-oriented development methods, as they offer an exciting new form of re-use of design structures. Combination of patterns can create powerful software architectures and frameworks. A more detailed overview of the design patterns plus their role in reusable software development is provided in later sections of the paper.

Specifically, the research goals translate into the following main objectives:

1. To use an object model integrating reusable software libraries with explicit schemes of design rationale capture and retrieval.

2. To develop prototype using the above mentioned object model as its base and test them in an industrial setting for use as an integrated design tool for software developers working in the domain of reusable software engineering.

The proposed framework combining design rationale and design patterns will allow the conception, development and testing of a new methodology that adheres to a software development process and allows the recording and easy retrieval of valuable design rationale information and is able to record and present the knowledge gained during the collaboration. The prototype is being developed and tested on software developed from the MIT Intelligent Engineering Systems Laboratory as well as Air Traffic Control Software, Satellite Design Software and Hostile Missile Counter Attack Software developed at Charles Stark Draper Laboratory.

3 Case Study

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. Case studies have been undertaken for the representation and active computer support for capturing design rationale [Peña Mora et al., 1995b] [Peña Mora and Vadhavkar, 1996]. The design case studies shed light on the 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 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 artifact 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 study 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 some inference mechanism or a set of established design patterns or standards, in contrast to passive support in which the computer just records information entered by the designers.

In the case studies, 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. Designers emphasized that the capture of design rationale must be unobtrusive since they are paid to deliver a product and not to document a process. They explained that they cannot afford to spend a lot of time documenting their work even though it is required since they have to perform their work under heavy time constraints. 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.

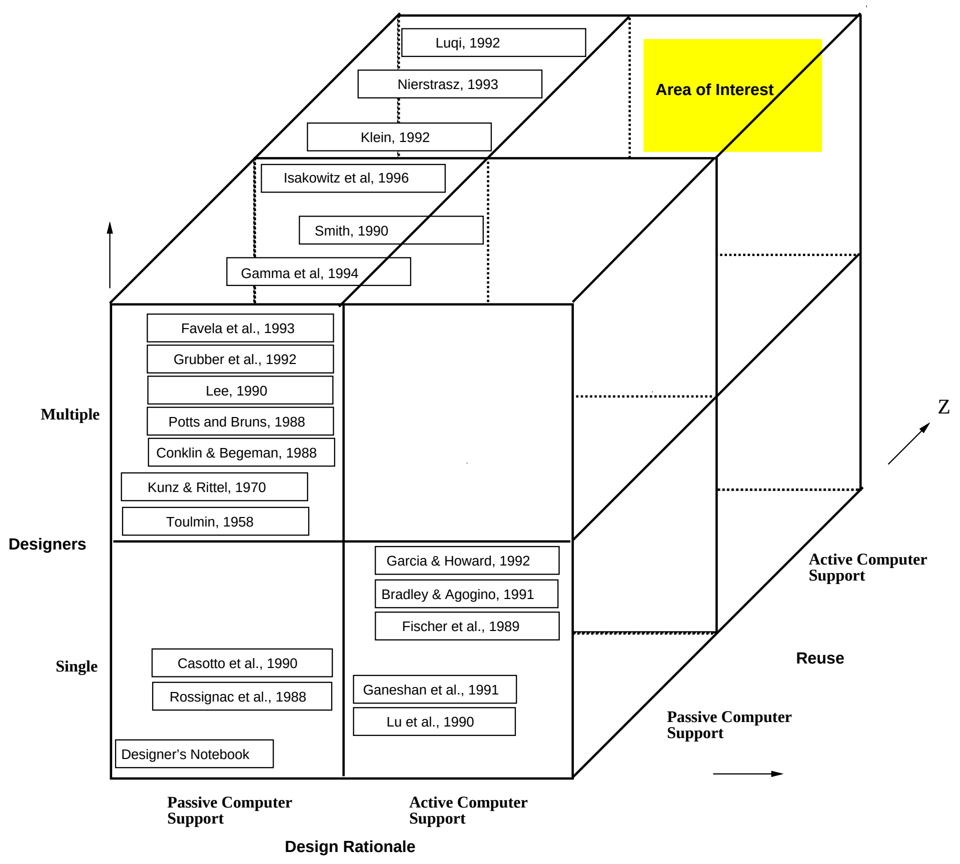


Figure 1: Comparison of research efforts in capturing design rationale for reuse

4 Survey of Current Models or Systems that Capture Multiple Design Rationale for Reusability

Any large scale software engineering system involves the expertise and knowledge of numerous software developers, engineers and programmers. A large scale involvement of such a nature, results in an interaction of different ideas and views, which invariably leads to a difficulty for reusing software components developed by other software designers. This difficulty arises mainly from one group's lack of information about the previous group's thinking behind accepting a particular software component or a solution for a design requirement, i.e. the design rationale is not carried forward as the design process goes on.

Capturing rationale has been a research topic for several decades [Peña Mora et al., 1995a]. There have been a number of models and systems developed by researchers in different application areas ranging from discourse [Toulmin, 1958], to engineering design [Garcia and Howard, 1992]. Figure 1 shows a classification scheme based on the role of design rationale in reusable software design.

In Figure 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 components 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 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 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 [Rossignac et al., 1988] and Cassotto *et al.*'s VOV [Casotto et al., 1990] 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, its the position of this research that design intents are more than operations. They 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 [Toulmin, 1958] and Kunz and Rittel's Issue Based Information System (IBIS) in the Urban Planning domain [Kunz and Rittel, 1970]. This quadrant also includes systems catering to the reusable software engineering domain. For example, Potts and Bruns' Model [Potts and Bruns, 1988]; Conklin and Begeman's Graphical Issue Based Information System (gIBIS) [Conklin and Begeman, 1988]; Lee's Design Representation Language (DRL) [Lee, 1990]. Gruber *et al.*'s SHADE [Gruber et al., 1992] system is an example of a design rationale capture system in the mechanical engineering domain for multiple designers, but providing passive computer support only. A similar example in the Civil and Construction engineering domain include Favela *et al.*'s CADS [Favela et al., 1993]. 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. 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 physical entity. This 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, first principles, etc) since they have different characteristics and are used different by the designers. Section 3 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. These systems take designers' options, evaluate them, and help the designers select the best option. The design patterns effort [Gamma et al., 1994] 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 [Smith, 1990] provides a semi-automatic program development system. Isakowitz *et al.*'s ORCA and AMHYRST [Isakowitz and Kauffman, 1996] 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 [Nierstrasz, 1993] and Luqi's CAPS project [Luqi and Royce, 1992]. These research efforts concentrate on computer-aided rapid prototyping systems. These systems help in rapid prototyping in well-defined and well-understood domains. 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 [Kim and Lochovsky, 1989]. 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.

5 Overview of building objects

A design pattern describes a solution to a recurring design problem in a systematic and general way. Design patterns capture expertise in building object-oriented software. A brief overview of the design patterns described in [Gamma et al., 1994] is given in Section 5.1.

To capture the design experience in a form that others can use effectively at a later stage and to use the concept of design rationale in a software development scheme, the Design Recommendation and Intent Model has been developed. The Design Recommendation and Intent Model is described in Section 5.2. The Design Recommendation and Intent Management System provides a method by which design rationale information from multiple designers can be partially generated, stored and later retrieved by a computer system [Peña Mora, 1994]. It uses domain knowledge, design experiences from past occasions and interaction with designers to capture design rationale.

5.1 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. A design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. They 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. Design patterns in software can be considered similar to the architectural patterns that exist in building complexes and communities [Gamma et al., 1994]. Specific objectives of design patterns in reusable software engineering are:

1. Identify good design that maps the solution to the implementation.
2. Explicitly specify how reusable classes relate to the design.

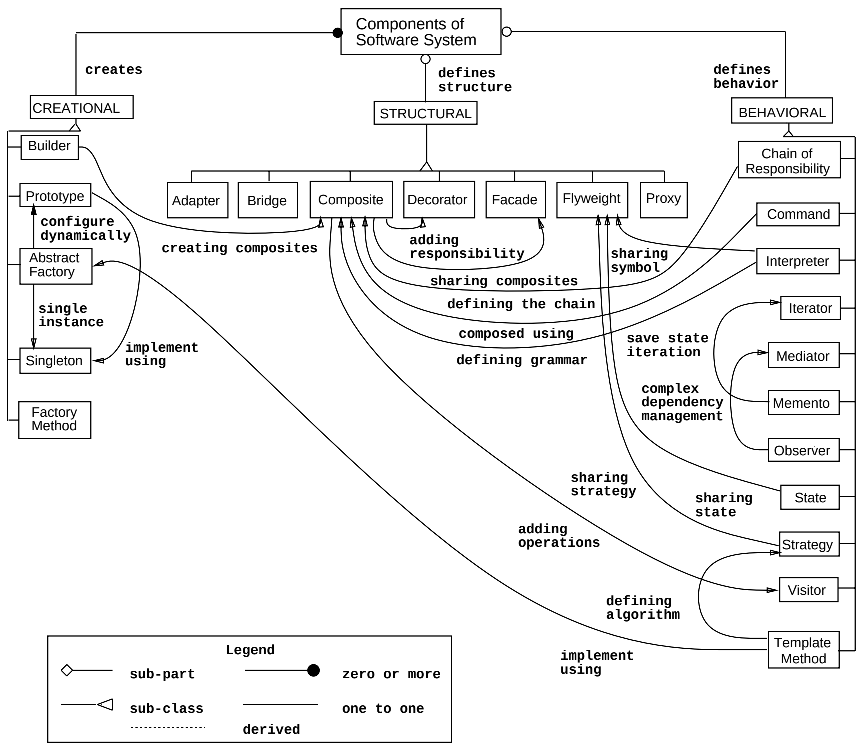


Figure 2: Design Patterns Relationships

3. Define the context in which the design patterns are valid.

4. Explicitly specify key issues, assumptions, constraints and dependencies in prior designs.

The catalog of design patterns as described in [Gamma et al., 1994] contains 23 design patterns. Although the present research concentrates more on the design pattern described in [Gamma et al., 1994], the catalog is by no means exhaustive. Design patterns are evolving at a fast pace thanks to a growing interest in the object-oriented software community. Design pattern relationships are shown in Figure 2. From the Figure, it is clear that the 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 (i.e., design rationale).

5.2 An Overview of DRIM

In Figure 3, the Design Recommendation and Intent Model (DRIM) is presented [Peña Mora, 1994]. The DRIM uses the Object-Oriented Modeling Technique (OMT) described in [Rumbaugh et al., 1991]. The 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 objective of the software project, the constraints involved, the function considered or the goal 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. A given proposal may consist of sub-proposals. A proposal may provide a response to an existing proposal by either supporting, contradicting or changing the ideas put in the existing proposal. A project proposal includes the designer's recommendation and the justification of why that particular proposal is recommended. 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 entities that need to

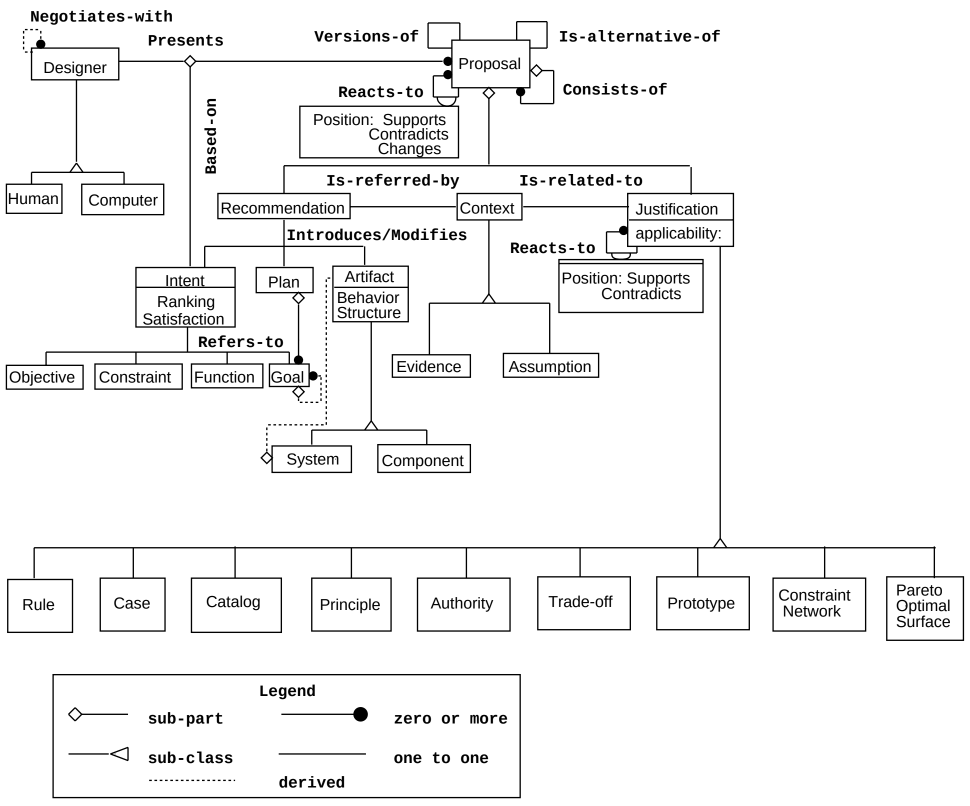


Figure 3: Design Recommendation and Intent Model

be satisfied in order to achieve the design intent. With a plan, more goals are brought into picture. The artifact denotes the 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, 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.

6 Combined methodologies 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 (For example, see Figure 7). 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 and design patterns has been implemented. The following sections cover in details the possibility of using design patterns/augmented code and DRIM for software reusability. First, an overview of how design patterns help in code reuse is presented in Section 6.1. In Section 6.2, the use of DRIM in design reuse is documented. DRIMER (Design Recommendation and Intent Model extended to reusability) is outlined as a means for achieving software reusability in Section 6.3.

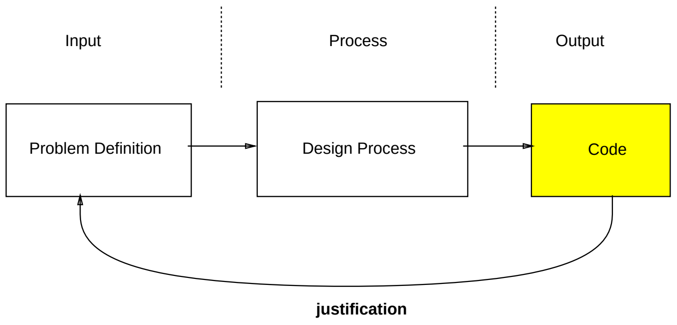


Figure 4: Software Development Scheme

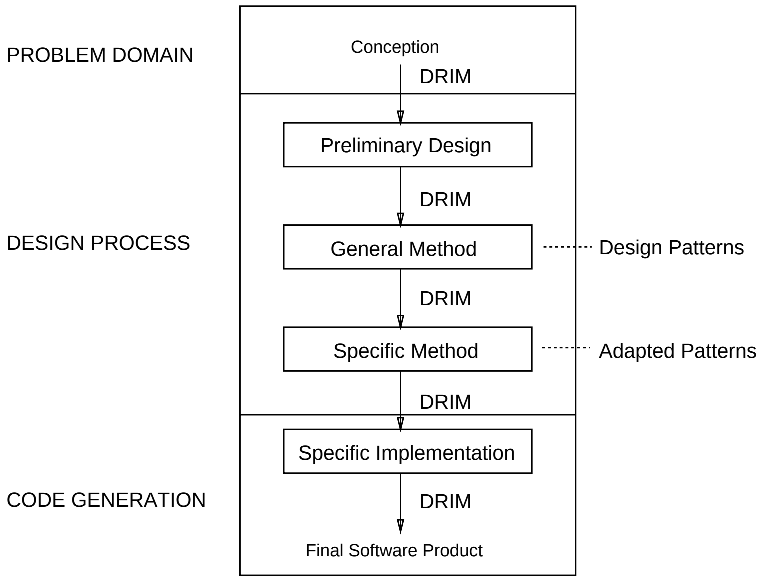


Figure 5: Basic cycle in reusable software development

Figure 4 shows the entire software development scheme. The *problem domain* 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 domain* is generated during the design process.

As shown in Figure 5, the basic cycle for the software process 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 domain* mentioned earlier.

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 this stage by providing an insight into similar design cases.

3. From the preliminary design, 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.

4. From the *general method* and taking into consideration the design issues pertaining to the system at

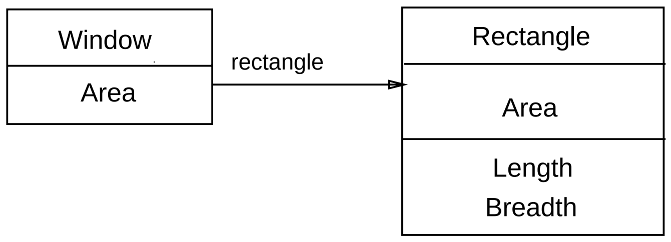


Figure 6: Delegation

hand, the designers select 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.*

5. 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 DRIM.

6. 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 domain.*

6.1 How Design Patterns help in putting Reuse Mechanisms to work

There are basically three types of reuse: white-box, black-box and delegation reuse. Reuse by subclassing is often referred to as **white-box reuse**. The white-box reuse technique uses class inheritance for reusing functionality in object-oriented systems. As an alternative, new functionality can be achieved by assembling or composing objects, i.e. by object composition. This style is known as **black-box reuse**. Most design patterns use **delegation** as a means of achieving reuse. In delegation, two objects are involved in handling a request, a receiving object delegating operations to it's delegates. For example, in figure 6, instead of making *Window* a class of *Rectangle*, the Window class might reuse the behavior of *Rectangle* by keeping a *Rectangle* instance variable and delegating *Rectangle* specific behavior to the *Window.* That is the area of the *Window* can be obtained by delegating it's area operation to a *Rectangle* instance. The main advantage of delegation is that it is easy to compose behavior at run-time. *Windows* can be made circular, by replacing the *Rectangle* instance with *Circle* instance.

Maximizing 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: This commits the designer to a particular implementation, creational design patterns like *Abstract Factory*, *Factory Method* and *Prototype* [Gamma et al., 1994] create objects indirectly.

2. Dependence on specific operations: Instead of specifying a particular operation, behavioral design patterns such as *Chain of Responsibility* and *Command* [Gamma et al., 1994], provide an easier way to satisfy a request. These design patterns either chain the receiving objects or encapsulate a request as an object.

3. Algorithmic dependencies: Algorithms are often extended or replaced during reuse. Objects depending on an algorithm need to be changed when that happens. Creational design patterns such as *Builder* [Gamma et al., 1994], avoid that by separating the construction of an object from it's representation. Behavioral patterns such as *Strategy* [Gamma et al., 1994], 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* [Gamma et al., 1994] use techniques like abstract coupling and layering to promote loosely coupled systems.

5. Extending functionality: Object composition and delegation offer flexible alternatives to inheritance for combining behavior. Design patterns such as *Bridge*, *Chain of Responsibility* and *Composite* [Gamma et al., 1994] introduce functionality by defining a subclass and composing its instances with existing ones. Design pattern *Observer* [Gamma et al., 1994], extends functionality by defining a one-to-many dependency between objects.

6. Inability to change classes conveniently: In some cases a class cannot be modified because the code is not available or may involve altering many subclasses. Design pattern *Adapter* [Gamma et al., 1994], 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. 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.

6.2 Using DRIM for software reusability

To capture the design experience in a form that others can use effectively at a later stage and to use the concept of design rationale in a collaborative environment, the Design Recommendation and Intent Model is suggested. The Design Recommendation and Intent Management System 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.

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.

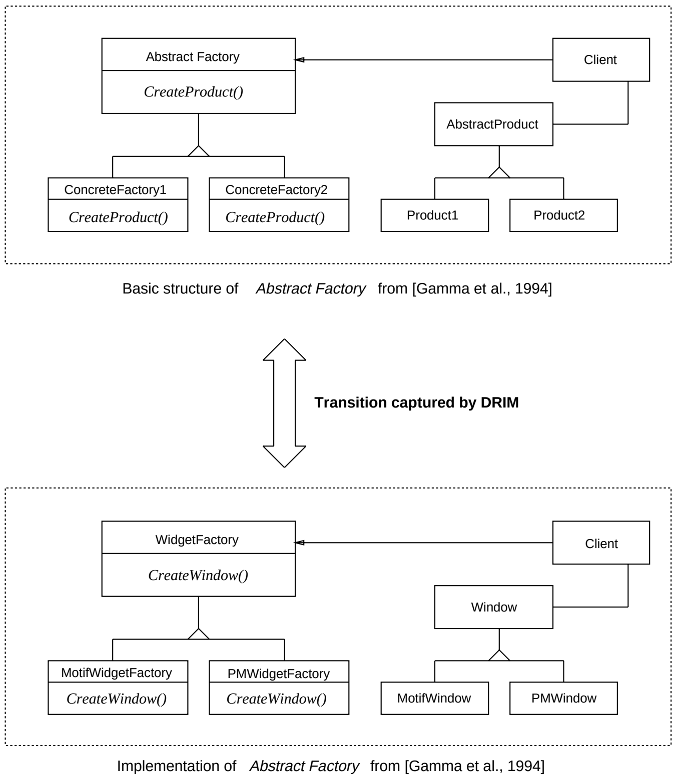


Figure 7: Structure and Implementation of *Abstract Factory*

Design patterns need validation by experience rather than by conventional testing. Conventional testing involves standard code testing techniques like black-box and white-box testing. These testing schemes are useful when the code to be tested is small and the domain is well-understood. Design patterns are usually validated by periodic patterns reviews [Schmidt, 1995]. In these pattern reviews a software expert usually validates the usefulness of the design pattern before it is documented and indexed for later use. The mechanics of implementing design patterns is left to the programmer in the individual domain. The transition from a predefined general structure of a design pattern to a specific implementation in a working environment, needs to be validated and captured in a form understandable at a later stage. Consider the following example from [Gamma et al., 1994] describing the structure and motivation of the design pattern *Abstract Factory* (Figure 7). The example has been simplified to point the use of DRIM in validating the design patterns. The first box represents the OMT [Rumbaugh et al., 1991] notation of the various classes 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. 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.

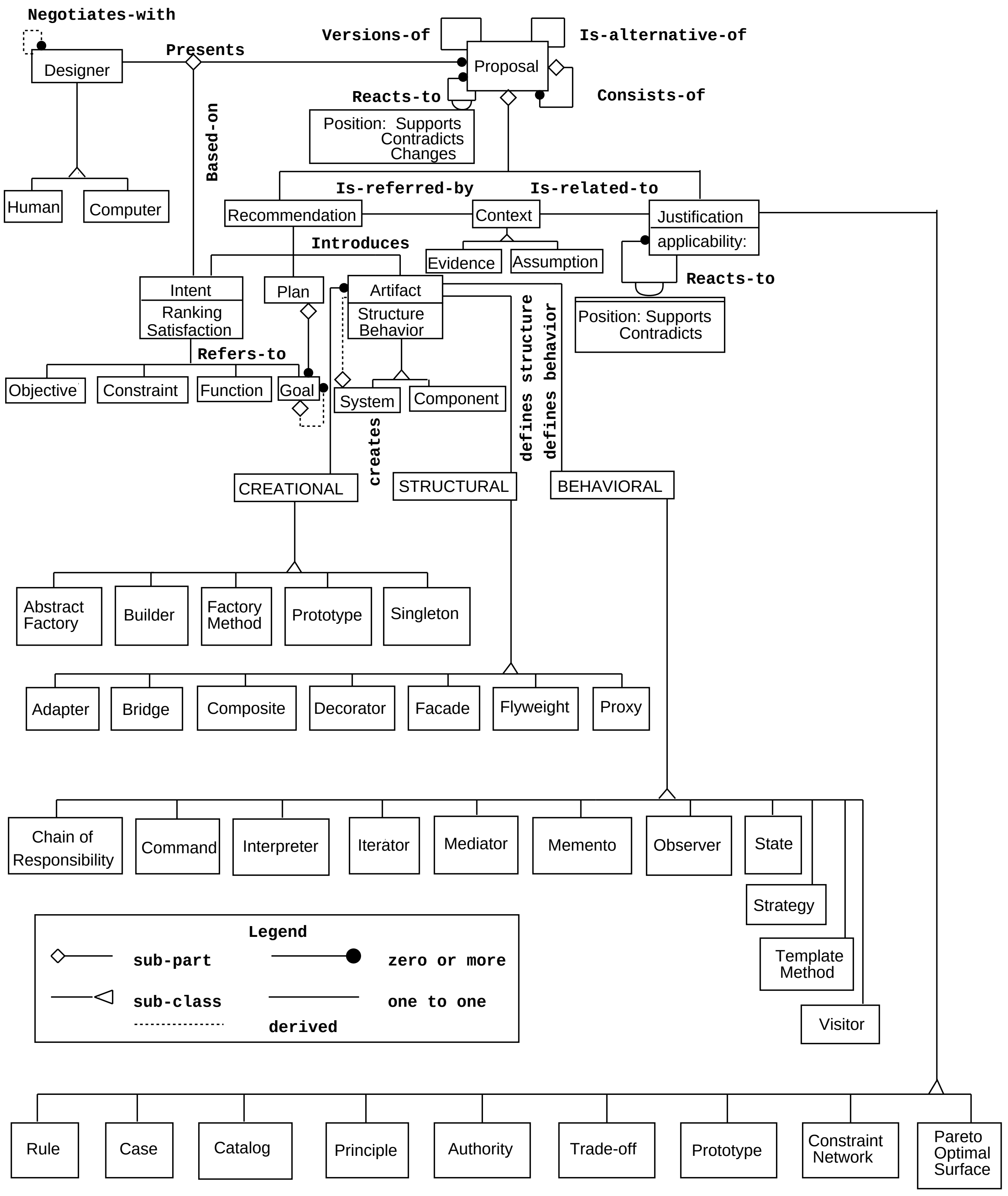


Figure 8: Design Recommendation and Intent Model Extended to Reusability

6.3 Design Recommendation and Intent Model Extended to Reusability

Figure 8 represents the Design Recommendation and Intent Model Extended to 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 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 [Schmidt, 1995]. 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 design and code reuse. The combined approach leads to the "patterns-by-intent" approach. The "patterns-by-intent" approach refers to the process of selecting patterns based on their initial intents and then refining the choice of the pattern by specific constraints. 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.

7 Software Prototype

7.1 Implementation

Based on the methodology described earlier, software prototype using Internet as the backbone was developed[3]. The web offers an excellent front end for this system. The web provides for linkages across different platforms and allows for information sharing between geographically separate designers. At present the database includes 2 design patterns and 6 augmented codes. The design patterns are adapted with the publisher's permission from Gamma's Design Patterns book [Gamma et al., 1994]. Augmented codes are software libraries along with DRIM wrappers that contain DRIM constructs like intent, justification and recommendation attached to them. The code libraries required for the augmented codes were obtained from free domains on the Internet and from the Air Traffic Control Software from Charles S. Draper Laboratory. The entire system is being implemented using C++, PERL, X-Windows/Motif, *Emacs* Macros, *OODesigner*^{TM} and *Netscape*^{TM} on a SUN-SPARC 10 workstation.

7.2 Examples

Figure 9 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 project. As a final stage, the designer tries to add the completed code to the code database.

To illustrate the mechanism of searching the database using the web interface, consider a software designer searching the database for design patterns handling interfaces. An interface denotes the general boundary

[3]The URL for the early prototype is http://ganesh.mit.edu/sanjeev/dp.html/

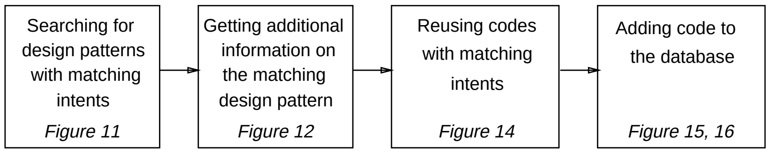
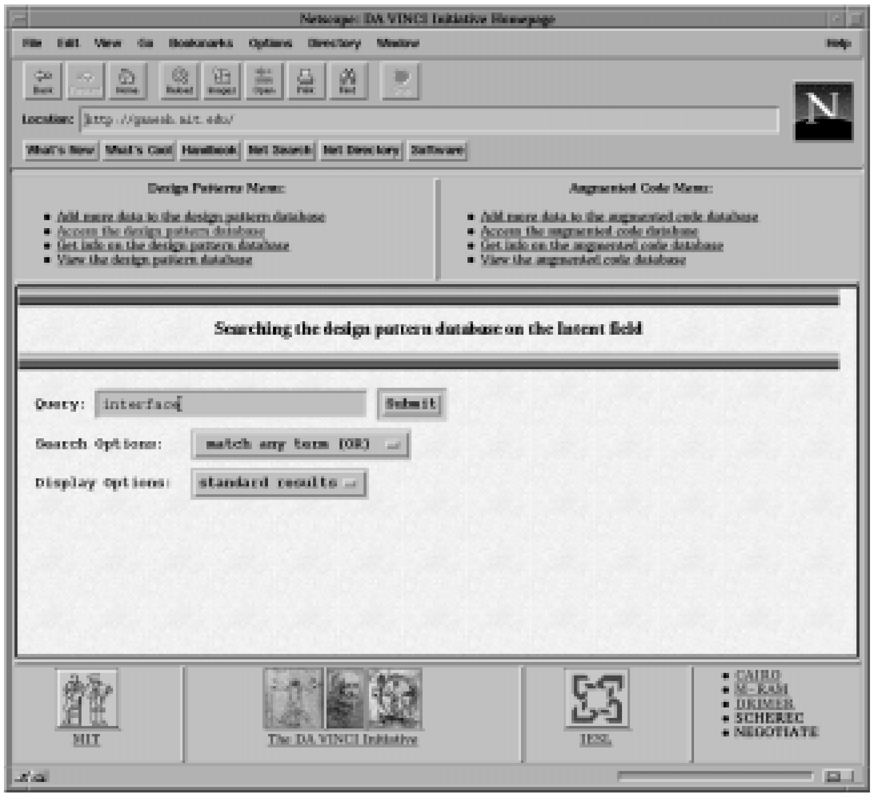

Searching for design patterns with matching intents
Figure 11
Getting additional information on the matching design pattern
Figure 12
Reusing codes with matching intents
Figure 14
Adding code to the database
Figure 15, 16

Figure 9: Flow control for the prototype


Netscape: DA VINCI Initiative Homepage
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
What's New What's Cool Handbook Net Search Net Directory Software
MIT
The DA VINCI Initiative
IESL

Location: http://pasesh.mit.edu/

Design Patterns Menu:

- Add more data to the design pattern database
- Access the design pattern database
- Get info on the design pattern database
- View the design pattern database

Augmented Code Menu:

- Add more data to the augmented code database
- Access the augmented code database
- Get info on the augmented code database
- View the augmented code database

Searching the design pattern database on the Intent field

Query: interface

Search Options: match any term (OR)

Display Options: standard results

- CAIRO
- M-RAM
- DRIMER
- SCHEREC
- NEGOTIATE

Figure 10: Searching the Intent Field

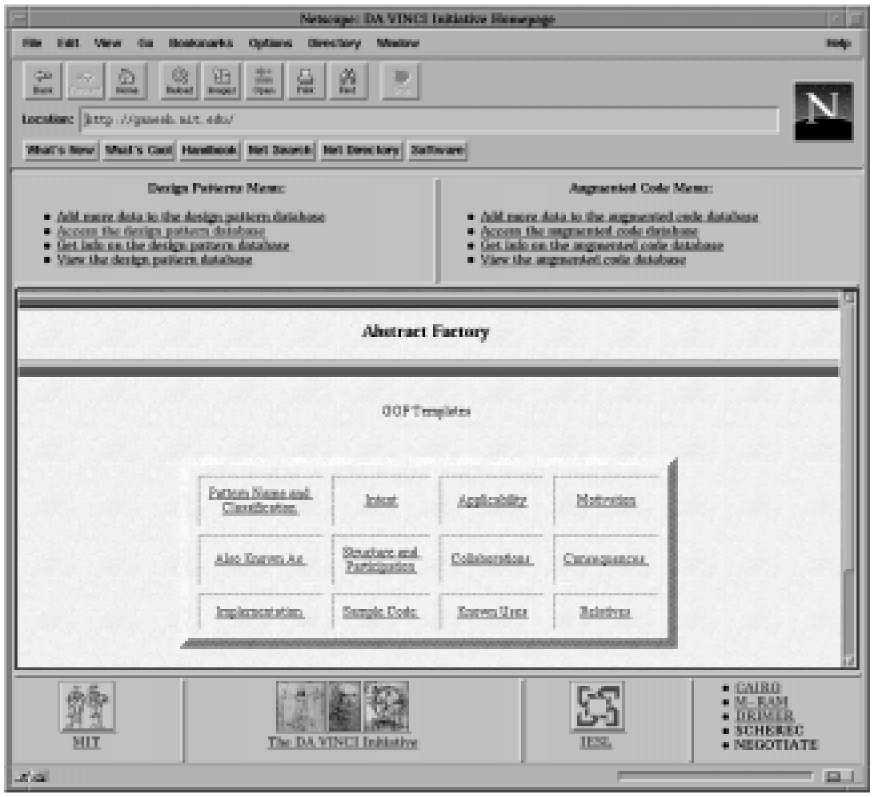

Netscape: DA VINCI Initiative Homepage
File Edit View Go Bookmarks Options Directory Window Help
Location: http://guanah.mit.edu/
What's New What's Cool Handbook Net Search Net Directory Software
Design Patterns Menu:
Add more data to the design pattern database
Access the design pattern database
Get info on the design pattern database
View the design pattern database
Augmented Code Menu:
Add more data to the augmented code database
Access the augmented code database
Get info on the augmented code database
View the augmented code database
Abstract Factory
OOFTemplates
Pattern Name and Classification
Intent
Applicability
Motivation
Also Known As
Structure and Participation
Collaborations
Consequences
Implementation
Sample Code
Known Uses
Relatives
MIT
The DA VINCI Initiative
IESL
CAIRO
M-RAM
DRIMER
SCHEREC
NEGOTIATE

Figure 11: Abstract Factory home page

separating two classes in an object-oriented programming model. From the home page he/she searches the database on the intent field. He/she submits the required query on interfaces to the database search engine as shown in Figure 10. The search engine returns 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 11. The information on the design pattern is classified according to the widely accepted templates described in Gamma's book [Gamma et al., 1994]. 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. A similar search on the name field can be made. Design patterns are becoming popular in the object-oreiented software community. To cater to software developers who have had some exposure to design patterns and augmented code, the option of searching on the name field has been provided.

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 was integrated with a freely available CASE tool named $OODesigner^{TM}$ [Kim, 1994]. The $OODesigner^{TM}$ uses the OMT (Object Modeling Technique) described in [Rumbaugh et al., 1991] to automatically generate C++ skeleton code. The $OODesigner^{TM}$ source code was modified to reflect the documentation format required for the web prototype. After the designer finishes drawing the OMT model, the skeleton C++ code is generated as shown in Figure 12. Previous reuse efforts

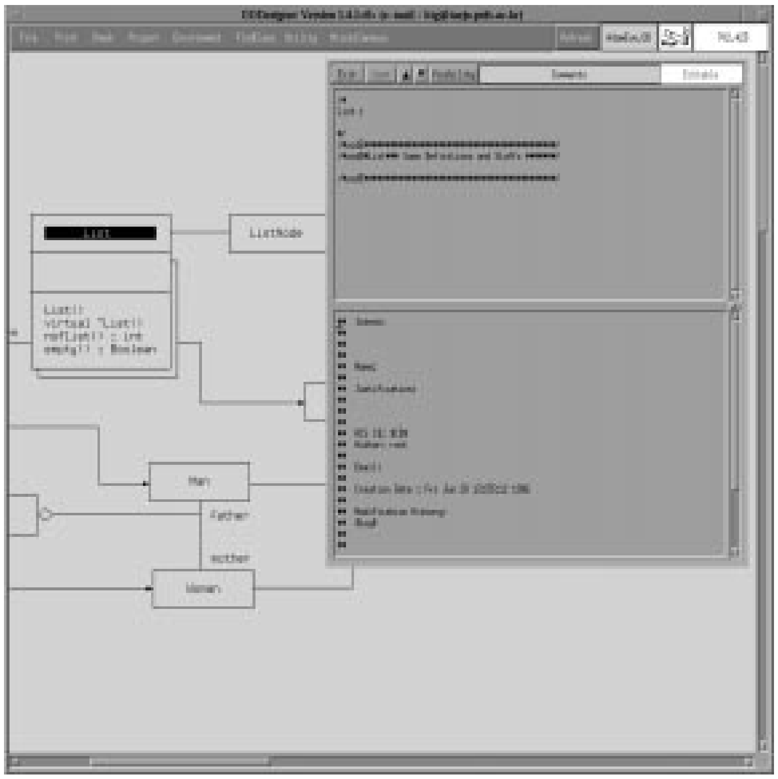


Figure 12: C++ skeleton code generation in OODesigner

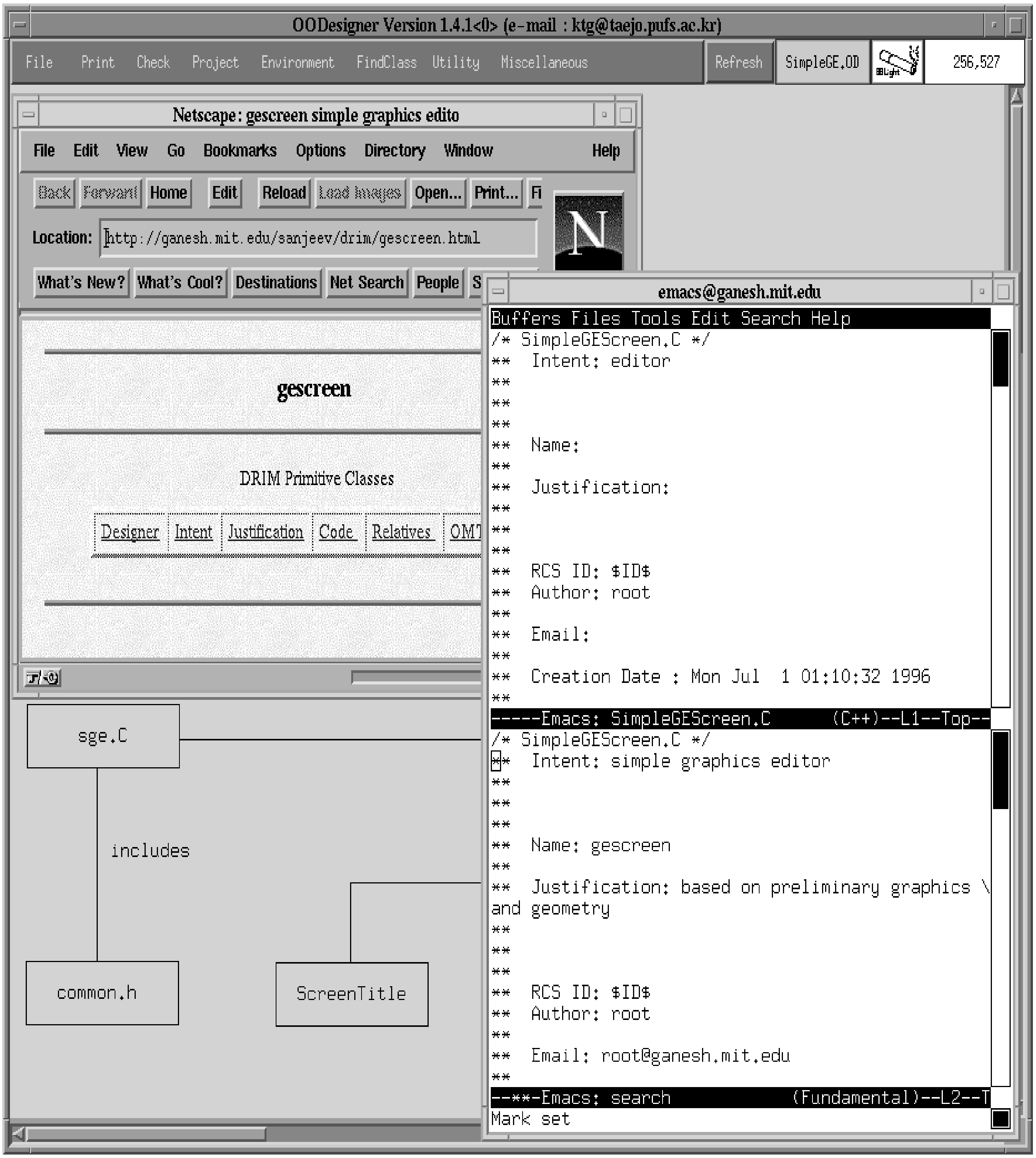
have emphasized strong documentation skills. Hence, *Emacs* macros were written to bring the feature of database addition and database access to a text editor. It is possible to search for code in the database directly from the *Emacs* buffer. When the search engine finds a match, it returns the code with the matching intent as a separate *Emacs* buffer for the designer to peruse. A web browser program is also invoked pointing to more detailed information on the code (Figure 13). These utilities ensure that the reuse effort takes only a fraction of the time taken from actual coding.

The following example illustrates the applicability of the software tool to add more information to the augmented code database. Consider a software developer who has developed some code and who wants to add his/her code to the database. From the home page, the software developer can choose 5 ways to submit his effort. 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. The interface parses the information from the source code to the various DRIM wrappers and make HTML files out of them. Figure 14 shows a designer adding the software code and information on various DRIM wrappers to the Database.

As shown in Figure 15, *Emacs* macros were written which make it possible to add the code to the database directly from the *Emacs* window.

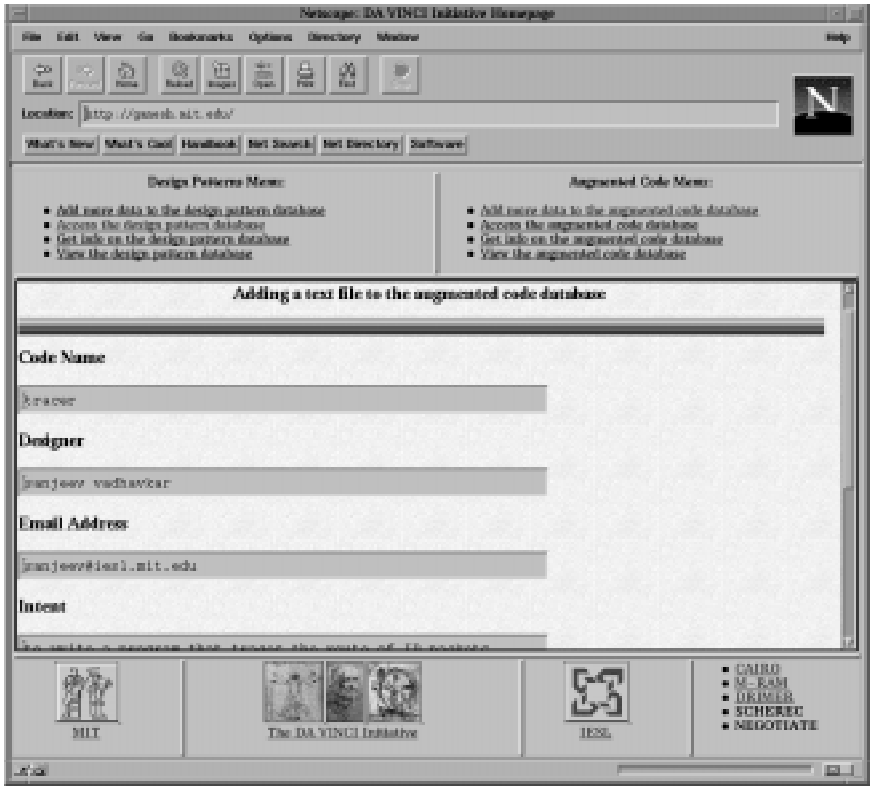
8 Conclusions

In summarizing this work, the contributions can be divided into two parts. First, a model for representing design rationale was developed. This model (DRIM) provides primitives for representing design knowledge


OODesigner Version 1.4.1<0> (e-mail : ktg@taejo.pufs.ac.kr)
File Print Check Project Environment FindClass Utility Miscellaneous
Refresh
SimpleGE.OD
256,527
Netscape: gescreen simple graphics edito
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Load Images Open... Print...
Location: http://ganesh.mit.edu/sanjeev/drim/gescreen.html
What's New? What's Cool? Destinations Net Search People
gescreen
DRIM Primitive Classes
Designer Intent Justification Code Relatives OMT
sge.C
includes
common.h
ScreenTitle
emacs@ganesh.mit.edu
Buffers Files Tools Edit Search Help

```
/* SimpleGEScreen.C */
**  Intent: editor
**
**
**
**  Name:
**
**  Justification:
**
**
**
**  RCS ID: $ID$
**  Author: root
**
**  Email:
**
**  Creation Date : Mon Jul  1 01:10:32 1996
**
-----Emacs: SimpleGEScreen.C       (C++)--L1--Top--
/* SimpleGEScreen.C */
**  Intent: simple graphics editor
**
**
**
**  Name: gescreen
**
**  Justification: based on preliminary graphics \
and geometry
**
**
**
**  RCS ID: $ID$
**  Author: root
**
**  Email: root@ganesh.mit.edu
**
--**-Emacs: search            (Fundamental)--L2--T
Mark set
```

Figure 13: Results from the search routines


Netscape: DA VINCI Initiative Homepage
File Edit View Go Bookmarks Options Directory Window Help
Location: http://ganesh.mit.edu/
What's New What's Cool Handbook Net Search Net Directory Software
MIT
The DA VINCI Initiative
IESL
CAIRO
M-RAM
DRIMER
SCHEREC
NEGOTIATE

Design Patterns Menu:

- Add more data to the design pattern database
- Access the design pattern database
- Get info on the design pattern database
- View the design pattern database

Augmented Code Menu:

- Add more data to the augmented code database
- Access the augmented code database
- Get info on the augmented code database
- View the augmented code database

Adding a text file to the augmented code database

Code Name

tracer

Designer

sanjeev vadhavkar

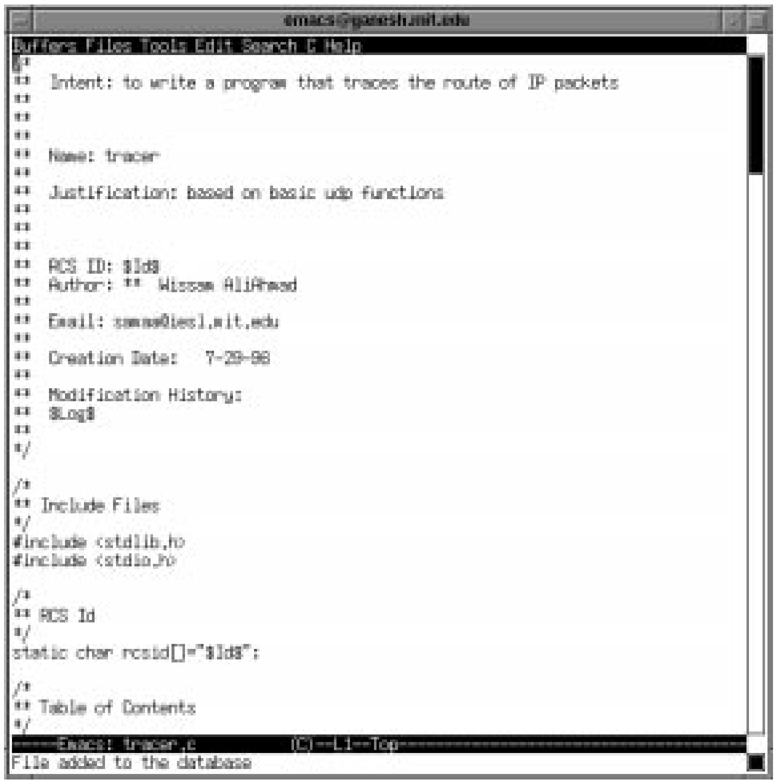
Email Address

sanjeev@iesl.mit.edu

Intent

to write a program that traces the route of IP packets

Figure 14: Adding to the augmented code database



```
/*
**  Intent: to write a program that traces the route of IP packets
**
**
**
**  Name: tracer
**
**  Justification: based on basic udp functions
**
**
**
**  RCS ID: $Id$
**  Author: **  Wissam AliAhmad
**
**  Email: samaa@iesl.mit.edu
**
**  Creation Date:   7-29-98
**
**  Modification History:
**  $Log$
**
*/

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

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

/*
** Table of Contents
*/
-----Emacs: tracer.c          (C)--L1--Top-----------------------------------
File added to the database
```

Figure 15: The Emacs Macros

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. The model allows human designers interacting with computers to record the rationale of their design.

Second, the proposed framework combining Design Recommendation and Intent Model with design patterns, 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 a software design tool that facilitates software reuse by: 1] Using an exhaustive library of tested software components. 2] Recording and allowing easy retrieval of decisions made during the software design process. 3] Providing economic methods for systems by providing a context for design modifications when the requirements change over their life time.

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.

9 Future Work

The prototype described in Section 7 needs to be 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. To make the tool more versatile, a more general search and index mechanism based on Case-Based Reasoning [Kolodner, 1993] needs to be implemented. 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 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 inferences can't be made. Learning occurs as a natural consequence of reasoning where novel procedures applied to problems are indexed in memory. By integrating Case-based reasoning into the present methodology, the following advantages are envisaged:

1. The system will be able to propose initial solutions quickly, based on similar cases existing in the system's database.
2. In case-based reasoning systems, learning occurs as a natural consequence of learning [Kolodner, 1993]. Thus, the system will be able to evaluate future software cases even in the absence of algorithmic methods.

Acknowledgments

The authors would like to acknowledge the support received from the Charles Stark Draper Laboratory (CSDL). Funding for this project comes from CSDL, award number DL-H-4847757. The authors gratefully acknowledge the support received from Professor Taegyun Kim of Pusan University of Foreign Studies, Pusan, Korea the creator of $OODesigner^{TM}$.

References

[Casotto et al., 1990] Casotto, A., Newton, A., and Sangiovanni-Vincentelli, A. (1990). Design Management based on Design Traces. In *27th ACM/IEEE Design Automation Conference*, pages 136– 141, Orlando, FA. IEEE.

[Conklin and Begeman, 1988] Conklin, J. and Begeman, M. (1988). gIBIS: A Hypertext Tool for Exploratory Policy Discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331.

[Favela et al., 1993] Favela, J., Wong, A., and Chakravarthy, A. (1993). Supporting Collaborative Engineering Design. *Engineering with Computers*, 9(4):125–132.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns.* Addison-Wesley Publishing Company, Reading, Massachusetts.

[Garcia and Howard, 1992] Garcia, A. and Howard, H. (1992). Acquiring Design Knowledge Through Design Decision Justification. *AI EDAM*, 6(1):59–71.

[Gruber et al., 1992] Gruber, T., Tenenbaum, J., and Webber, J. (1992). Toward a Knowledge Medium for Collaborative Product Development. In Gero, J., editor, *Artificial Intelligence in Design '92*, pages 413–432. Kluwer Academic Publishers, London, England.

[Isakowitz and Kauffman, 1996] Isakowitz, T. and Kauffman, R. (1996). Supporting Search for Reusable Software Objects. *IEEE Transactions on Software Engineering*, 22-6:407–424.

[Kim, 1994] Kim, T. (1994). *OODesigner FAQ.* Pusan University of Foreign Studies, Pusan, Korea.

[Kim and Lochovsky, 1989] Kim, W. and Lochovsky, F. (1989). *Object-Oriented Concepts, Databases and Applications.* ACM and Addison-Wesley Publishing Company, Reading, Massachusetts.

[Kolodner, 1993] Kolodner, J. (1993). *Case Based Reasoning.* Morgan-Kaufmann, .

[Kunz and Rittel, 1970] Kunz, W. and Rittel, H. (1970). Issues as Elements of Information Systems. Institute of Urban and Regional Development Working Paper 131, University of California, Berkeley, Berkeley, CA.

[Lee, 1990] Lee, J. (1990). SIBYL: A Qualitative Decision Management System. In P. Winston and S. Shellard, editor, *Artificial Intelligence at MIT: Expanding Frontiers*, chapter 5, pages 104–133. MIT Press, Cambridge, MA.

[Lubars, 1991] Lubars, M. (1991). The ROSE-2 Strategies for supporting high-level software design reuse. In Hahn, F., editor, *Automating Software Design*, pages 184–190. AAAI Press and The MIT Press, Cambridge, Massachusetts.

[Luqi and Royce, 1992] Luqi and Royce, W. (1992). Status Report: Computer-Aided Prototyping. *IEEE Software*, 19-5:77–81.

[Nierstrasz, 1993] Nierstrasz, O. (1993). Composing Active Objects. In G. Agha, P. Wegner and A. Yonezawa, editor, *Research Directions in Concurrent Object-Oriented Programming*, pages 151–171. MIT Press.

[Peña Mora, 1994] Peña Mora, F. (1994). *Design Rationale for Computer Supported Conflict Mitigation during the Design-Construction Process of Large-Scale Civil Engineering Systems.* PhD thesis, Massachusetts Institute of Technology.

[Peña Mora et al., 1995a] Peña Mora, F., Sriram, D., and Logcher, R. (1995a). Conflict Mitigation System for Collaborative Engineering. *AI EDAM – Special Issue on Concurrent Engineering*, 9(2):101–124.

[Peña Mora et al., 1995b] Peña Mora, F., Sriram, D., and Logcher, R. (1995b). Design Rationale for Computer Supported Conflict Mitigation. *Journal of Computing in Civil Engineering*, 9(1):57–72. Best Paper of 1995 published in the Journal.

[Peña Mora and Vadhavkar, 1996] Peña Mora, F. and Vadhavkar, S. (1996). Design Rationale and Design Patterns in Reusable Software Design. In Gero, J., editor, *Artificial Intelligence in Design '96*. Kluwer Academic Publishers, London, England.

[Potts and Bruns, 1988] Potts, C. and Bruns, G. (1988). Recording the Reasons for Design Decisions. In *Proceedings of the 10th International Conference on Software Engineering*, pages 418–427. IEEE.

[Rossignac et al., 1988] Rossignac, J., Borrel, P., and Nackman, L. (1988). Interactive Design with Sequences of Parameterized Transformations. In *Intelligent CAD Systems 2: Implementational Issues*. Springer-Verlag, New York, NY.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

[Schmidt, 1995] Schmidt, D. (1995). Experience using Design Patterns to develop reusbale object-oriented communication software. *Communications of the ACM*, 38:65–74.

[Shaw, 1990] Shaw, M. (1990). Towards higher level of abstractions for software systems. *Data and Knowledge Engineering*, 5:119–128.

[Shum, 1996] Shum, S. B. (1996). Design Argumentation as Design Rationale. In A.Kent and J.G. Williams, editor, *The Encyclopedia of Computer Science and Technology*. Marcel Dekker Inc, New York.

[Smith, 1990] Smith, D. (1990). KIDS: A Semi-Automatic Program Development System. *IEEE Transactions on Software Engineering*, 16:1024–1043.

[Toulmin, 1958] Toulmin, S. (1958). *The Uses of Argument*. Cambridge University Press, Cambridge, England.

[Weide et al., 1996] Weide, B., Edwards, S., Heym, W., Long, T., and Ogden, W. (1996). Characterizing Observability and Controllability of Software Components. In *Proceedings 4th International Conference on Software Reuse*, pages 62–71. IEEE.