Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2001), *15*, 173–187. Printed in the USA.
Copyright © 2001 Cambridge University Press 0890-0604/01 $12.50

Component-based software development for integrated construction management software applications

FENIOSKY PEÑA-MORA,[1] SANJEEV VADHAVKAR,[1] AND SIVA KUMAR DIRISALA[2]

[1] Intelligent Engineering Systems Laboratory, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
[2] Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065, USA

(RECEIVED February 23, 1999; ACCEPTED August 30, 2000)

Abstract

This paper presents a framework and a prototype for designing Integrated Construction Management (ICM) software applications using reusable components. The framework supports the collaborative development of ICM software applications by a group of ICM application developers from a library of software components. The framework focuses on the use of an explicit software development process to capture and disseminate specialized knowledge that augments the description of the ICM software application components 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 application code. There are three main components in the framework: design patterns, design rationale model, and intelligent search algorithms. 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 extended in the current research effort to capture the specific implementation of reusable software components. DRIM provides a method by which design rationale from multiple ICM application designers can be partially generated, stored, and later retrieved by a computer system. To address the issues of retrieval, the paper presents a unique representation of a software component, and a search mechanism based on Reggia's setcover algorithm to retrieve a set of components that can be combined to get the required functionality is presented. This paper also details an initial, proof-of-concept prototype based on the framework. By supporting nonobtrusive capture as well as effective access of vital design rationale information regarding the ICM application development process, the framework described in this paper is expected to provide a strong information base for designing ICM software.

Keywords: Component-based Software; Design Patterns; Design Rationale; Integrated Construction Management Applications; Intelligent Search Mechanisms; Process Planning; Software Framework

1. INTRODUCTION

1.1. Component-based approaches to ICM

Integrated Construction Management (ICM) software applications refer to a suite of software applications handling: scheduling for process planning, inventory management, financial and accounting, resource/contract management, equipment management, and procurement. Integrating these essential back-office software applications into one integrated suite results in substantial savings in cost and reduces end-user complexity. However, 85% of this integrated applications industry bears a strong resemblance to many industries of the early 19th century—virtually all of the products are monolithic, expensive, hand-crafted, error prone, and one of a kind. Individual pieces of the system cannot be easily reused (except in some instances in the same company) because of the close cooperation required between the process planners, designers, programmers, and the systems integrators. By most estimates, between a quarter to half of all application development projects in the construction industry end in failure—after all the time and money is spent, the product still cannot be used operationally. To counter this, the software application development industry, as a whole, has been undergoing a paradigm shift

Reprint requests to: Sanjeev Vadhavkar, Intelligent Engineering Systems Laboratory, Department of Civil and Environmental Engineering, MIT, Room 1-270, 77 Massachusetts Avenue, Cambridge, MA 02139, USA. E-mail: vada@mit.edu

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

concerning the way in which software is designed, developed, integrated, and distributed. Large single-vendor systems that are designed to address an entire application-level suite of functionality are gradually being replaced. Software applications are instead being divided into smaller units or components that may be combined in various ways by software integrators, or even by the end-user, into a unified product. The software components that are being developed may be relatively useless out of context. However, in combination with other components and with an application framework, the software components provide the end-user with the functionality that is directly tailored to and useful in the setting of his or her work.

The component-based approach offers significant promise in creating ICM software applications. The systems required in this setting are large and complex; necessarily customizable and adaptable to individual institution and user needs, and to the changing needs of each; and, ideally, ubiquitous, encompassing the whole enterprise concerned with satisfying the needs of managers. These system characteristics pose daunting challenges to the construction industry that has traditionally been unable to cope effectively with the full spectrum of needs of project managers. This in turn leads to high prices for inadequate systems that address only a small subset of user needs. This ongoing shift to a component-based ICM software application industry is also the result of increasing pressures from the consumer for larger, more complex, more reliable, and more customizable application software.

A variety of challenges to the systems designer for ICM software systems of the future stems from the capabilities demanded by ubiquitous computing—in particular, the extended enterprise environment in which these computing facilities are to be installed and used. The first step in planning component-based ICM software systems architecture lies in the analysis of current and expected future needs. The aim is to provide computing capabilities that are highly integrated with the work that is done in a normal business setting, and to provide these capabilities without imposing unacceptable cognitive load on the project manager and other system developers. Thus, these capabilities must be based on the way in which work is already done, must track changes in the nature of that work as it occurs, and should not redefine the nature of the work itself.

1.2. Importance of component-based approaches to ICM

Current practices in the software application industry emphasize the generation and tracking of design artifacts such as 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. The *design rationale* exists in the form of domain/context knowledge, development experience, design decisions, design history, code, and documentation. However, this information on design rationale is stored in forms like design notebooks, minutes of design reviews, or designer's memory. Consequently, it is difficult to recover and reuse this vital information. A Hewlett-Packard study (Griss, 1993) showed that for application development, using components with explicit design rationale resulted in a 57% increase in software productivity and a 42% decrease in project development time. Conservatively assuming that development of a reusable component would double its cost and that each reuse would cost a quarter as much as redeveloping the component, an average reuse of 10 times would lower overall costs by 60%. Data for the entire software application industry indicates, with even a 5% market share, in the United States alone, the savings in software application development today from reuse alone would exceed $2 billion annually. International Data Corporation (IDC) expects the software component market alone, $100 million in 1995, to reach the $1 billion mark by 2000. Gartner Group Inc., an IT advisory firm, projects that the overall market for commercial components, models, and templates will exceed $2 billion by 2001 and will generate $7 billion in related services. A survey by the Standish Group, an IT research firm specializing in software reuse issues, found that 8% of the corporate IS departments surveyed plan to deploy software components this year, and another 32% plan to deploy software components in 1998. This data, though for the entire software applications industry, shows that there is significant economical potential for adopting a component-based approach for commercial systems and in-house software development. This is typical of how ICM software is currently developed in engineering and construction companies such as Bechtel, Kajima, Flour Daniel, ABB, Bovis, and others (NIST, 1997).

1.3. Broad research goals

To mitigate the types of problems identified above, the research targeted three focus areas:

- The creation of an object model integrating reusable software components with explicit schemes of design rationale capture and retrieval, to *reduce the direct involvement of software experts in custom development*.
- The use of design patterns as high-level repositories of *cross-domain* and *enterprise-wide* software components.
- Representation of the software components and *retrieval of multiple, but minimum, components whose combination gives as much of the required functionality as possible*.

To provide a "complete solution" to the problem of reusable application development is a massive research under-

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

taking, dealing with methodological issues, development tools and environments, software classification and retrieval, and project management tactics and strategy (Nierstrasz, 1993). The current research had a more refined objective, which was to integrate principles from the domains of design rationale, design patterns, and intelligent search mechanisms to create a framework for component-based ICM applications. Specifically, the research focused on the following main goals:

- To use an object model integrating reusable software libraries with explicit schemes of design rationale capture and retrieval.
- To develop and test a framework that combines design rationale, design patterns, and intelligent search principles. The new framework adheres to a software development process and allows the recording and easy retrieval of valuable design rationale information.
- To develop a prototype using the above-mentioned framework as its base for use as an integrated design tool for collaborative application development. The prototype was able to record and present the knowledge gained during the collaboration between ICM application designers separated across geographic space (different locations) and time (different projects).

The following sections cover in detail the ideas put forward to address these challenges. Section 2 presents a background of the research domain concentrating on reusability issues as well as experiences on using design patterns in other domains. An overview of the building components in the new framework is presented in Section 3. The use of the Design Recommendation and Intent Model (DRIM), design patterns, and intelligent search principles for software reusability are explored in Section 4. DRIMER (Design Recommendation and Intent Model Extended for Reusability) is outlined as a means for achieving software reusability in Section 4.3. Section 5 details the initial proof-of-concept prototype built using the framework developed in Sections 3 and 4. Finally, Section 6 presents a summary of the findings and the research contributions.

2. BACKGROUND

There has been a proliferation of artificial intelligence techniques for engineering design and management activities. Prior research has addressed traditional problems such as resource allocation, optimized design, and process management using artificial intelligence techniques such as neural networks, knowledge-based expert systems, and genetic algorithms (Fischer & Kunz, 1995; Maurer, 1996; Smith et al., 1996; Petrie et al., 1998; Clayton et al., 1999). Consider for example, the Procura model (Goldmann, 1996), which has been developed as a framework for the distributed application of generic engineering services. The increasing focus of all these research activities is in creating interdependent subsets of components that can interoperate and provide reusable functionality. These research efforts have concentrated on creating artificial-intelligence-based solutions geared toward solving specific activities in design and management of engineering artifacts. The work presented in this paper more involves providing the computer support for combining these solutions in a component-based architecture for building integrated construction management software.

Increasingly it is being realized that planning of software development efforts is a dynamic endeavor requiring knowledge about the process itself as well as the implicit design rationale (Dellen & Maurer, 1996; Maurer, 1996). Effective software reuse requires more than building easy to browse, well-cataloged, convenient software components (Shaw, 1990; Lubars, 1991). Surveys of current models that capture design rationale for reusability by Peña-Mora and Vadhavkar (1997) and Vadhavkar (1997) showed the lack of models handling active design rationale capture for multiple application designers. Reusable software can best be accomplished with knowledge of not only what the software application is and what it does, but also why it was put together in that way and why other design alternatives were discarded. The difficulties in reusability at the primary code level are outlined in Section 2.1. Earlier attempts to achieve reusability by using classifying schemes to software are discussed in Section 2.2.

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 (Biggerstaff & Perlis, 1989):

- *Mechanisms for identifying components*: How can one determine what components are generally useful and adaptable to many different projects?
- *Methods 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 (Shaw, 1990)?
- *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*: With the assumption that there will be a large number of software components, how does one represent the components so that a group of them can be retrieved for reuse by composing them?

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

2.2. Classification of software components

Various means of overall improvement in each of the above mentioned questions and the experiences and quantitative measures are present by Biggerstaff and Perlis (1989). Efforts have been made to provide better languages that can support high levels of abstraction and interface between modules and templates. These methods help in writing reusable components. However, the ability to reuse software components does not stop with the ability to write reusable components. The main problem is how to select the appropriate components for reuse in a given context. Hence, attempts in retrieving the appropriate software components based on the context have been done by applying the classifying schemes (Prieto-Diaz, 1987). The classification is based on functionality and environment. Functionality is represented as a ⟨*function, object, medium*⟩. A component is represented as a sextuple. An ICM example is ⟨*manage, activities, plan, schedule, PERT chart, organization*⟩.

This type of classification can only capture the description of smaller components with little functionality. In addition, the limitation with this approach is that every reusable component needs to be described in six words. Another way of categorizing software components is using a set of attributes to the component. However, this approach is limited and cannot classify all the software components because of the various ways a component can be classified and various properties a component can have. A fixed number of predetermined attributes cannot cover all the necessary features to classify software components from various domains. Hence, a list of attribute-value pairs alone cannot classify software components. One should be able to describe a software component by a set of features provided by the component instead of a set of attribute-value pairs. The advantage with this approach is that it will not limit the power to describe a reusable component in terms of predetermined attributes because they will not be sufficient to describe all the components. However, certain properties like the language of implementation, the operating system in which the component can be used, support for concurrency, and version number may be used as attributes. These attributes limit the use of these components based on factors that are external to the functionality of the components and are applicable for classifying all software components.

3. OVERVIEW OF BUILDING COMPONENTS

To satisfy the goals of this research in terms of an integrated framework for component-based ICM applications, 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 (Gamma et al., 1994) have been 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 (Peña-Mora, 1994) is used in the proposed framework to capture the specific implementation of reusable software systems. This framework uses a suitable representation of software components for retrieval of multiple components using Reggia's setcover algorithm (Siva Kumar, 1998).

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 application designers. A brief overview of the design patterns described in Gamma et al. (1994) is given in Peña-Mora and Vadhavkar (1997) and Vadhavkar (1997).

DRIM (Peña-Mora, 1994) 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. DRIM is described in Peña-Mora and Vadhavkar (1997) and Vadhavkar (1997).

The requirements of a complex system will be many and varied such that a single existing component cannot have all the required functionality. However, a set of existing components could provide all the required functionality or a major portion of it. Hence, a retrieval system based on keyword matching for individual components will not provide solutions where a combination of components is required. Search based on intent field, name of the code, or some other form of indexing as suggested in Peña-Mora and Vadhavkar (1997) will only give individual components. It is possible to retrieve components specific for each functionality. Such a simple retrieval mechanism will result in the retrieval of several individual components with a poor degree of relevancy with each component satisfying one or a few features. This is because the mechanism to compose a minimum number of components to give the maximum desired functionality is not considered. This framework uses a suitable representation of software components for retrieval of multiple components using Reggia's set-

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

cover algorithm (Siva Kumar, 1998). The combined functionality of all these components will give most of the required functionality and hence increase the degree of relevancy to the actual requirements. Software with the required functionality can then be built by suitably composing the existing components with smaller functionality. Reggia's setcover algorithm is used to minimize the search by giving a minimum set of components that satisfy most of the requirements for the software.

4. COMBINED METHODOLOGIES FOR SOFTWARE REUSABILITY

4.1 Development Process for Reusable Software

With the three core building components described above, this section delves into a combined architecture for creating component-based ICM applications. A design pattern only describes a solution to a particular design problem. Most software application 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 re-implementation, because different design choices in the design pattern can lead to vastly different final code. To address these needs, a framework combining design rationale, design patterns, and intelligent search principles is proposed. The following sections cover in detail the possibility of using design patterns, DRIM, and intelligent search principles for software reusability. DRIMER (Vadhavkar, 1997), which combines design patterns and DRIM, is outlined as a means for achieving software reusability in Section 4.1. The role of intelligent search mechanisms in software reusability is outlined in Section 4.2. A contemporary survey of similar approaches is presented next in Section 4.3 to highlight the case for the technologies used in the current endeavor.

Figure 1 shows a simplified process-oriented 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 product of the *design process*. The *justification* for the *code* satisfying the *problem definition* is generated during the design process.

Figure 2 shows the envisioned basic cycle for the software process using the proposed framework. The steps of the framework are as follows:

- The design process starts with the *conception* of various goals, objectives, and functions of ICM software applications. This is a part of the *problem definition* mentioned earlier in Figure 1.
- Based on the requirements, the designers come up with a *preliminary design* satisfying some intents and taking into consideration some constraints. DRIMER helps in capturing the transition from the *conception* to the *preliminary design* of ICM software.
- Next stage involves the *retrieval* of software designs from the database. Reggia's setcover method for retrieval of components provides solutions by considering the combination of the components for achieving the required software functionality for ICM.
- After *retrieving* relevant components, 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. DRIMER helps in capturing the relationship between the *preliminary design* and the *general methods* to be implemented in ICM software.
- 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 the view of the application, thereby giving the whole software design process the characteristic iterative property. DRIMER helps in capturing the design rationale in the evolution of the *general method* to the *specific method*.
- After finalizing the *specific method*, the designers go ahead with the *specific implementation* of ICM software applications. The various issues in the *specific implementation* of the software code can be studied and if the new component is generic enough, a new

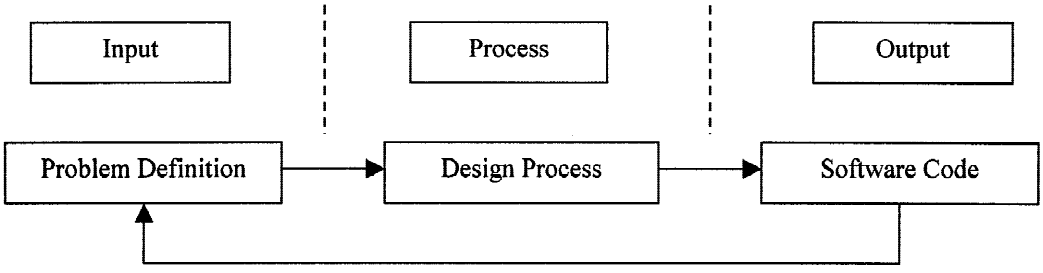


Fig. 1. Software development scheme.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

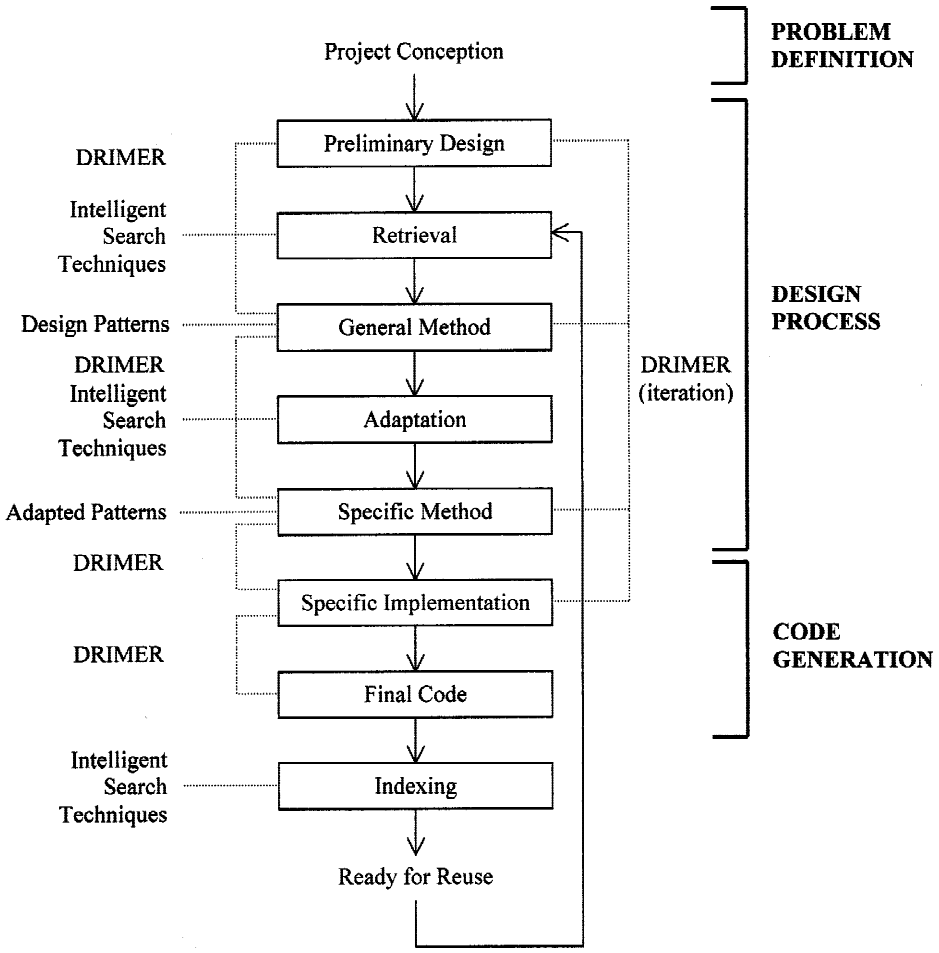


Fig. 2. A generic reusable software development process.

software component with higher functionality can be added to the component database for later use.

- 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. DRIMER explicitly presents the justification that the *code* generated satisfies the requirements of the *problem definition*. The evolution of the ICM application is captured in the above mentioned steps.
- To reuse the newly generated code requires representing it with a set of most useful features and constraints. The new component will have higher functionality and, hence, if a similar component is required in the future, it can be directly used without the need for composing it from smaller components. However, it should be noted that the higher the functionality a component has, the more specific it will be. Such specific components will have less reusability. Hence, the user has to decide on storing the new component in the database depending on how generic the component is.

4.2. Design Recommendation and Intent Model Extended to Reusability

Figure 3 shows the Design Recommendation and Intent Model Extended for Reusability. In DRIMER, the *artifact* component in a software design context represents the components in a software application. A software *designer* presents a *proposal*, which includes a *recommendation* and a *justification*. The *recommendation* introduces or modifies the components in a software application. The design patterns either create these components (i.e., Creational Patterns), or define their structure (i.e., Structural Patterns), or define their behavior (i.e., 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, making it difficult to motivate strategic design choices by other groups within a project or organization (Schmidt, 1995). Identifying, documenting, and reusing useful design

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

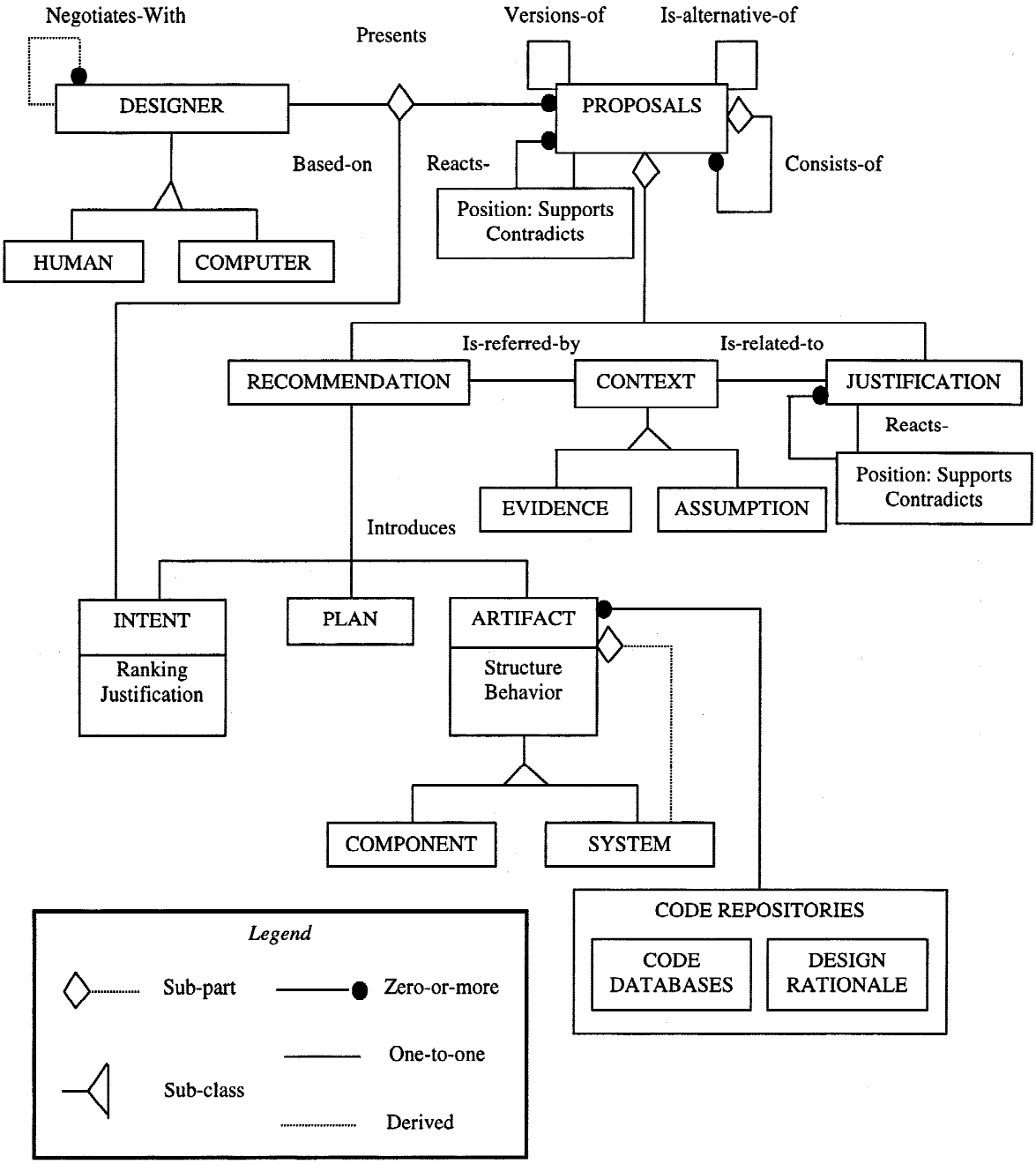


Fig. 3. Design Recommendation and Intent Model Extended for Reusability.

patterns requires concrete experience in a domain. 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.

4.3. Role of intelligent search techniques for software reusability

Searching for components without considering the effect of combination of the components will result in several components each with a poor degree of relevance. However, retrieving the components intelligently, by considering the

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

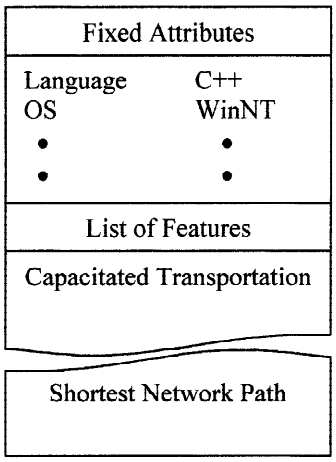


Fig. 4. Software component representation.

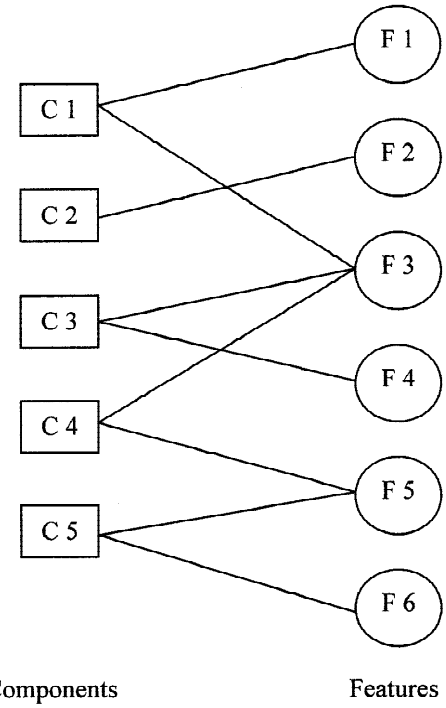


Fig. 5. Software components and features represented as bipartite graph.

effect of combination of the components to cover all the required features will give solutions with a high degree of relevance. The main intent of Reggia's setcover algorithm can be stated in terms of software components and software features as finding the combination of a minimum number of components with the required features. A software component is represented as shown in Figure 4. It has a set of fixed attributes and a list with avariable length of features. The fixed attributes act as constraints to limit the search space, while the list of features describe the component.

For example, suppose the user wants features F1, F3, and F5 (see Fig. 5). If the combination of the components is not considered, the user will be presented with the components C1, C3, C4, and C5. In this case, the degree of relevance will be low because the ratio of the number of features a component can cover to the total required features is low. However, if the combination of the components is considered to cover all the given features, the user will be presented with the solutions (C1 C4) and (C1 C5). This will give a higher degree of relevancy and an immediate idea of which components can be composed for the required features. A software component having a set of features may have additional constraints like the language in which the component is implemented and the operating system in which it can be used. Hence, although a software component has a particular feature, it may be implemented in a language different from what the user wants. In such a case, this software component should not be considered to cover the corresponding features. To continue with our example, suppose components C3 and C5 are implemented in C++ and C1 and C4 are implemented in Java. Then for the same query as above, that is, components with features F1, F3, and F5 together with the constraint that the implementation language should be Java, only C1 and C4 will be given as the possibility.

5. SOFTWARE PROTOTYPE FOR THE SYSTEM

This section is devoted to the software prototype of Peña-Mora and Vadhavkar (1997) and Vadhavkar (1997) of the proposed framework described in earlier sections. The section concentrates more on the implementation issues as well as a running example to illustrate the use of the proposed framework to design component-based process planning software. A detailed description of the system architecture and actual working can be found in Vadhavkar (1997). Charles Stark Draper Laboratory (CSDL) provided the initial funding to develop and test an Internet based prototype (http://ganesh.mit.edu/sanjeev/dp.html) using the framework described earlier. 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.

Figure 6 shows a typical use of the prototype developed on the framework discussed in earlier sections. The process begins with an ICM application 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. Section 5.1 explains the implementation issues of the prototype. An example in Section 5.2 shows how the prototype works. The internals of the prototype are described in Section 5.3.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

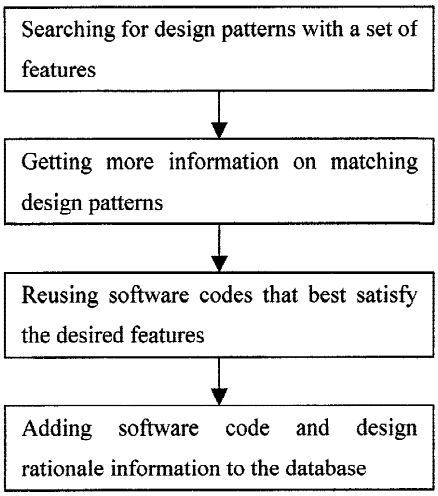


Fig. 6. Flow control for the prototype.

5.1. Implementation

At present the code database of the prototype includes 4 design patterns and 10 augmented codes. The database is by no means comprehensive and is used primarily to illustrate the underlying technologies of the framework. The design patterns were 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 Planning Software from CSDL. The entire system has been implemented using C++, Java™, PERL, X-Windows™/Motif, Emacs Macros, OODesigner™, and Netscape™ on a SUN SPARC 10™ workstation. Figure 7 shows the sys-

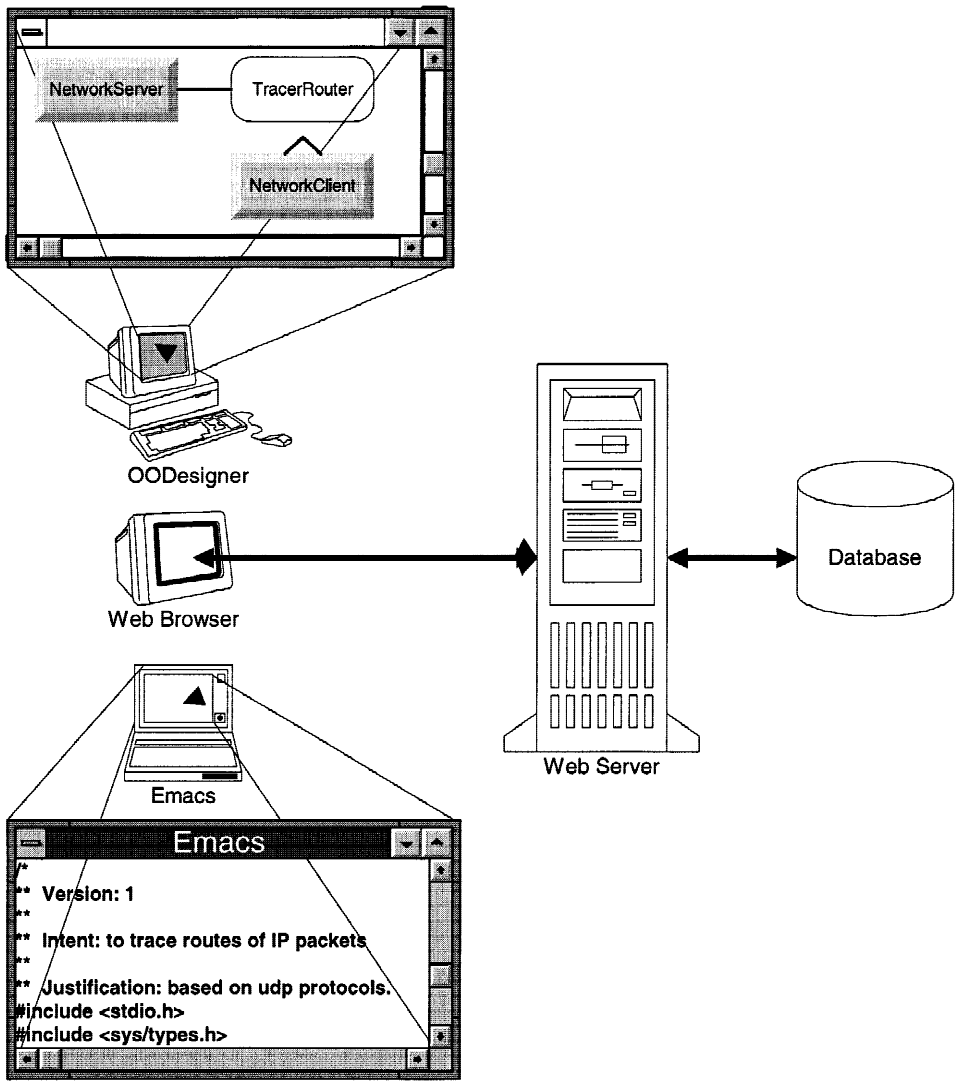


Fig. 7. System architecture.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

tem architecture that would help peers in replicating the prototype described here.

5.2. Example

To illustrate the mechanism of searching the database using the web interface, consider an ICM application designer searching the database for design patterns handling interfaces. An interface denotes the general boundary 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 (Peña-Mora & Vadhavkar, 1997; Vadhavkar, 1997). The search engine returns the design pattern with matching intent (in this case, *Abstract Factory*). The application developer can access more information on the design pattern by going to the web page of the pattern (Peña-Mora & Vadhavkar, 1997; Vadhavkar, 1997). 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 ICM application 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. To cater to 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 ICM application designers working in the domain of reusable software, it was necessary to integrate this web-based prototype with the CASE tools used for ICM application development. Combining the above-mentioned web-based prototype with a commercial CASE tool resulted in an integrated design environment for reusable software creation. To fully and beneficially implement reuse, the documentation effort essential for reuse has to be made as simple as possible for the ICM application designer. As a first step in this direction, the web-based prototype was integrated with a freely available CASE tool named OODesigner^{TM} (Kim & Shin, 1998). The OODesigner uses the OMT (Object Modeling Technique) described in Rumbaugh et al. (1991) to automatically generate C++ skeleton code. 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. Previous reuse efforts have emphasized strong documentation skills. Hence, Emacs macros were written to bring the feature of database addition and database access to a

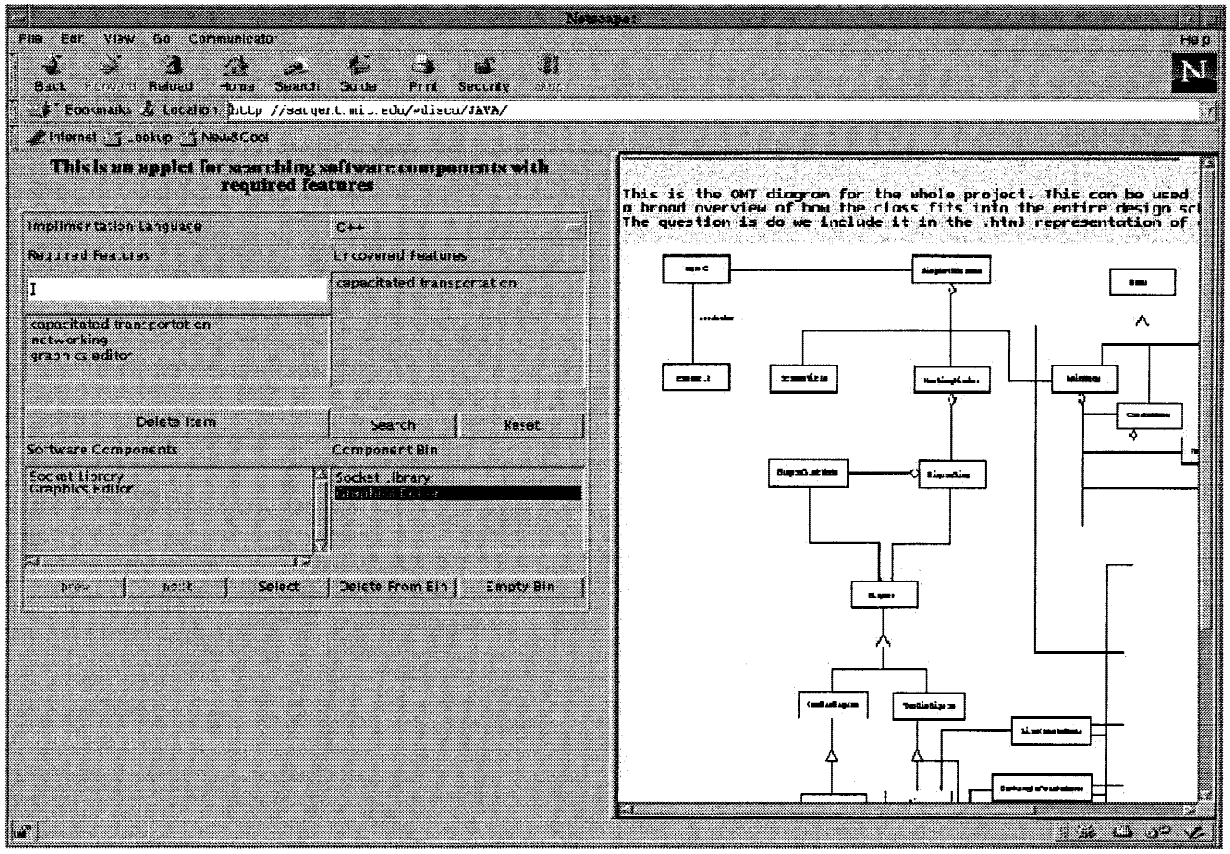


Fig. 8. Search based on features and constraints.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

text editor. It is possible to search for code in the database directly from the Emacs buffer. Additional utilities in the prototype ensure that the reuse effort takes only a fraction of the time taken from actual coding.

The prototype can also be used effectively to add more information to the augmented code database (Peña-Mora & Vadhavkar, 1997; Vadhavkar, 1997). Consider an ICM application developer who has developed some code and who wants to add his/her code to the database. From the home page, the developer can choose five ways to submit his effort. He/she can either send a text file or an e-mail 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 DRIMER wrappers, he/she submits the formatted file. The prototype provides Emacs macros to add the code to the database directly from the Emacs window.

The above examples show how to retrieve components based on simple keywords defining the name, intent, or an index. The user can also specify a set of features, given additional constraints (for example, language of implementation, operating system) and get a minimum set of components satisfying the features as well as the constraints. In the present prototype, language is the only constraint specified for the components. This search process is shown in Figure 8. In this case, the language of implementation of the component should be C++. One of the features is not covered and two components are suggested for the rest of the features. It is possible to see the design details (see Figure 9) and implementation (see Figure 10) of the retrieved components. If the language constraint is removed as shown in Figure 8, then all the features are covered and three components are suggested.

5.3. Working

While adding code to the database from the web, the prototype uses CGI scripts written in PERL to parse documents from the HTML form to plain ASCII. While adding code to the database from Emacs, the prototype uses a combination of Emacs LISP macros and PERL scripts to parse documents (Peña-Mora & Vadhavkar, 1997; Vadhavkar, 1997). Since it is necessary to maintain a database mirror on the web for browsing from different sources, the prototype uses CGI scripts written in PERL to parse programs written using Emacs to standard HTML template. The parser

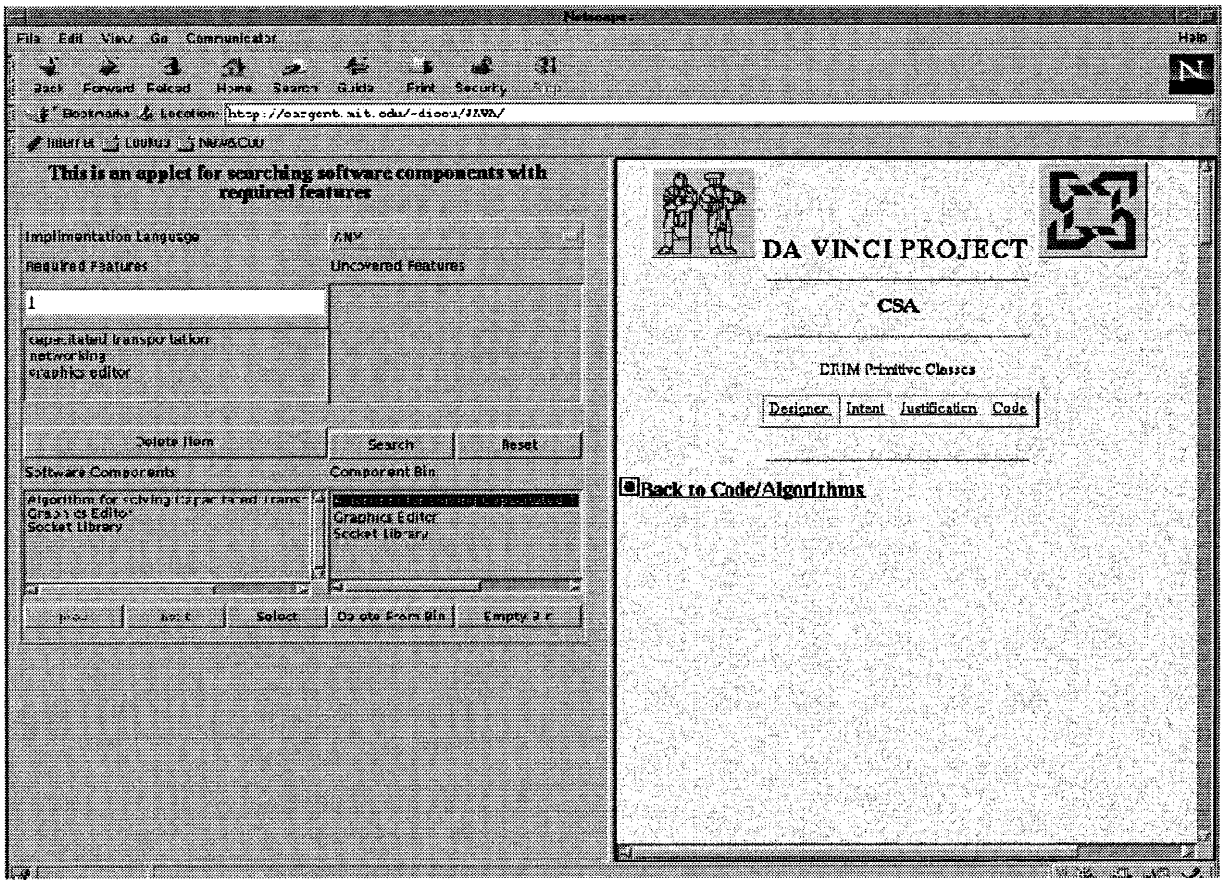


Fig. 9. Online browsing of retrieved components.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

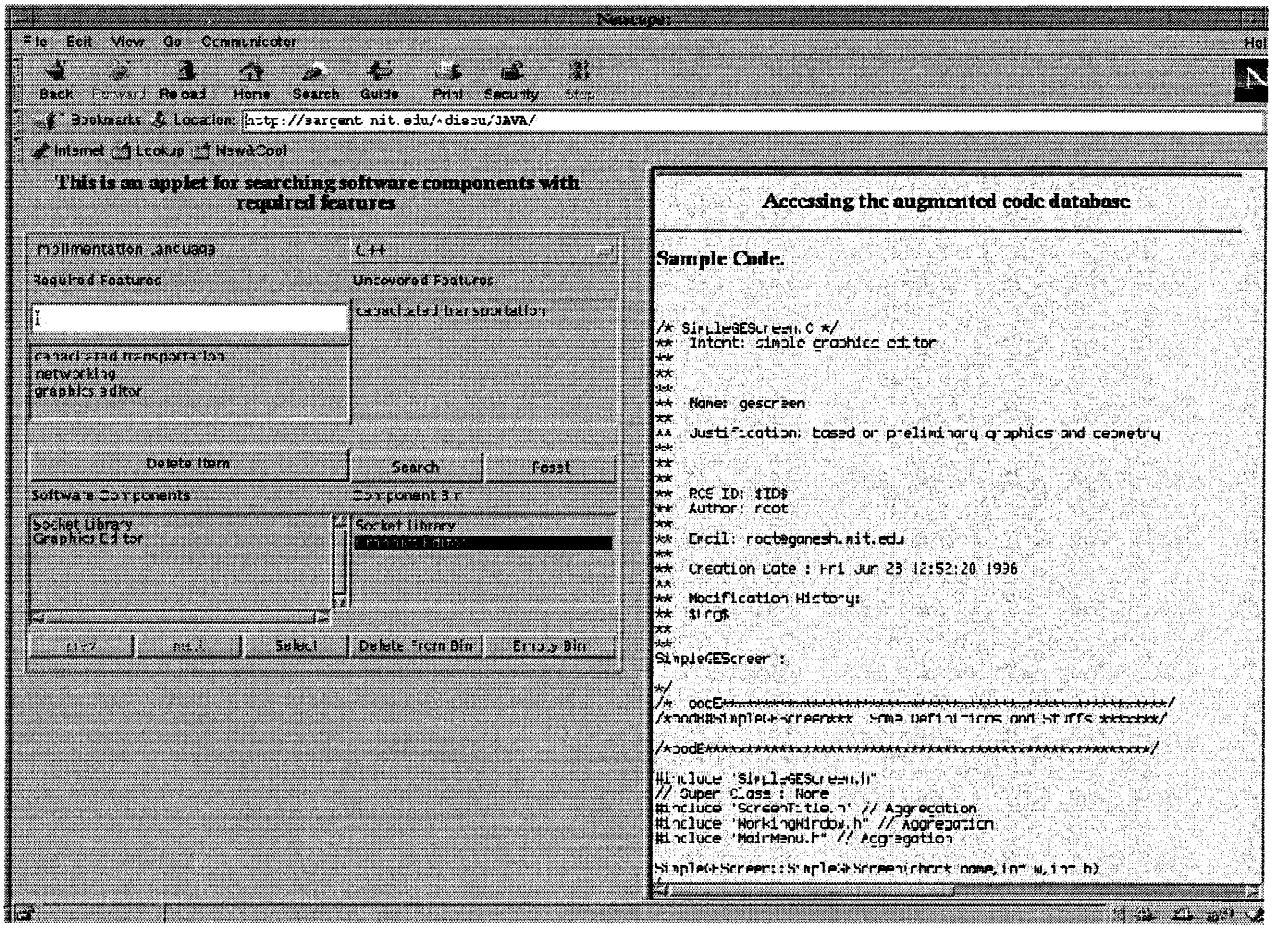


Fig. 10. Removing constraints by selecting ANY option.

also captures relevant design rationale information like *intent* and *justification* plus context information from the code documentation and saves them into flat files. The search engine searches on the intents and names from the flat files and retrieves the appropriate source code from the code database. The prototype also allows web browsing for additional information on the retrieved code. The appropriate OMT diagram is retrieved from the code database as well (see Fig. 11).

6. CONCLUSIONS AND FUTURE WORK

The DRIMER model and the initial prototype based on this model represent an initial attempt in creating a framework and building a prototype environment that can offer varying degrees of assistance to an ICM application designer for component-based software design. By combining the fast evolving area of design patterns with design rationale principles, significant benefits above current ICM application design practices are envisaged. This section 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 application developers in designing component-based ICM software applications efficiently and reliably. The proposed framework allows for active computer support to multiple designers in the area of active design rationale capture for reusability during ICM application development. The approach is based on abstracting the design as well as the process involved in designing a software system and reusing them to design a new system. The framework encourages AI researchers in the domain of engineering design, manufacturing, and construction to develop software components that can be reused in a number of ICM software applications. In addition, the prototype functions as an assistant that delivers the relevant knowledge to a designer so that the designer may make informed decisions during the design of ICM software applications.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

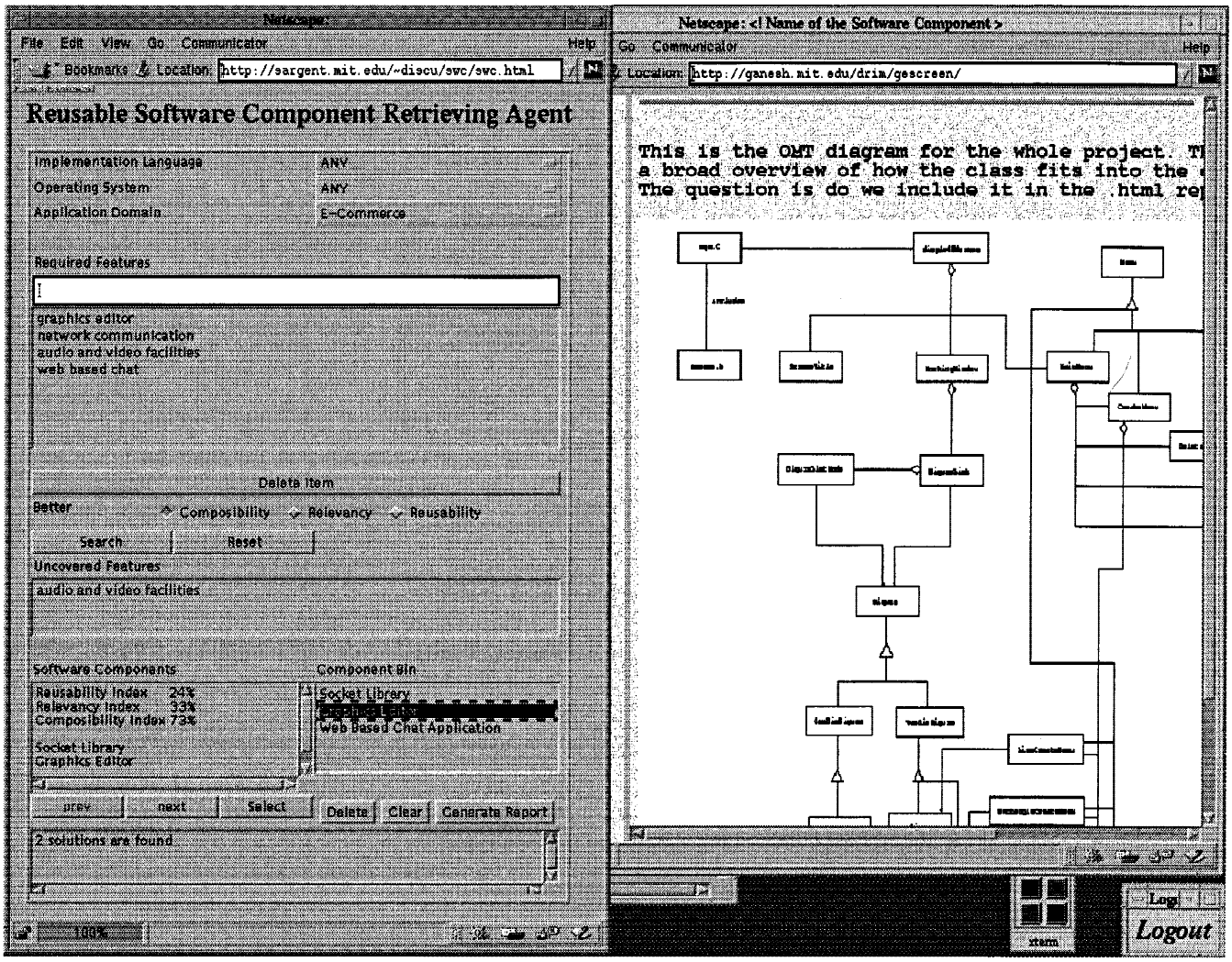


Fig. 11. Results from the search routines.

The research described herein has significant technological and economic benefits to the modern ICM application design process. The project envisions a paradigm shift from the specify-build-then-maintain life cycle assumed in the past to one of component-based software. Component-based software offers an economic relief to the change activity associated with modern ICM 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 that 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 DRIMER and intelligent search mechanisms offers active assistance to ICM application designers in designing reusable ICM software systems. Although the framework emphasizes the importance of documenting the ICM application development 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.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

- Providing economic methods for systems by providing a context for design modifications when the requirements change over their lifetime.
- Providing a strong information base for understanding ICM software applications.

6.2. Future work

This section concentrates on the steps that will be taken to refine the underlying theories on design rationale and design patterns as well as improve the prototype described earlier. A design rationale is the representation of the reasoning that has been invested in the design. The prototype described in Section 5 shows how computer support tools could augment the design rationale capture process by 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 a particularly open style of design and accountability within an organization that may be considered by some to be idealistic. Design rationale faces the harsh realities of industrial settings where there is a certain degree of distrust between ICM application developers and programmers making the process of capturing design rationale 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 the design-specific memory of the entire organization. It would also be interesting to carry out research in producing a design rationale model that filtered and integrated design rationale information from a wide range of sources across different ICM projects.

The prototype mentioned in Section 5 needs to be refined and tested in an industrial setting with 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 a large amount of ICM software code. The present number of design patterns and augmented code used for testing the prototype does not match the amount of ICM software code written in any average size ICM application project. For an implementation of the prototype in an industrial setting, the back end of the prototype will need to be upgraded to a database management system (DBMS).

REFERENCES

Biggerstaff, T., & Perlis, A. (1989). *Software Reusability: Volume 1 Concepts and Models*. Reading, MA: ACM and Addison-Wesley Publishing Company.

Clayton, M., Teicholz, P., Fischer, M., & Kunz, J. (1999). Virtual components consisting of form, function, and behavior. *Automation in Construction 8*, 351–367.

Dellen, B., & Maurer, F. (1996). Integrating planning and execution in software development processes. *Proc. WETICE-96*, Stanford, California, pp. 170–176.

Fischer, M., & Kunz, J. (1995). Data sharing and control in AEC software integration. *International Journal of Construction Information Technology 3(2)*, 77–90.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns*, pp. 197–183. Reading, MA: Addison-Wesley Publishing Company.

Goldmann, S. (1996). Procura: A project management model of concurrent planning and design. *Proc. WETICE-96*, Stanford, California.

Griss, M. (1993). Software reuse: From library to factory. *IBM Systems Journal 32(4)*.

Kim, T., & Shin, G. (1998). Restructuring OODesigner: a CASE Tool for OMT. *Proc. 1998 International Conference on Software Engineering*, Kyoto, Japan, 449–451.

Lubars, M. (1991). The ROSE-2 strategies for supporting high-level software design reuse. In *Automating Software Design* (Hahn, F., Ed.) pp. 184–190. Cambridge, MA: AAAI Press and The MIT Press.

Maurer, F. (1996). Project coordination in design processes. *Proc. WETICE-96*. Stanford, California, pp. 191–198.

Nierstrasz, O. (1993). Composing active objects. In *Research Directions in Concurrent Object-Oriented Programming* (Agha, G., Wegner, P. and Yonezawa, A., Eds.), pp. 151–171. Cambridge, MA: The MIT Press.

Peña-Mora, F. (1994). *Design Rationale for Computer Supported Conflict Mitigation During the Design Construction Process of Large-scale Civil Engineering Systems*. Ph.D. Thesis, Massachusetts Institute of Technology.

Peña-Mora, F., & Vadhavkar, S. (1997). Augmenting design patterns with design rationale. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 11(2)*.

Petrie, C., Goldmann, S., & Raquet, A. (1998). Agent-based process management. In *Proc. Int. Workshop on Intelligent Agents in CSCW*, Deutsche Telekom, Dortmund. 1–17.

Prieto-Diaz, R. (1987). Classification of reusable modules. *IEEE Software 4(1)*, 6–16.

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

Schmidt, D. (1995). Experience using design patterns to develop reusable object-oriented communication software. *Communications of the ACM 38*, 65–74.

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

Siva Kumar, D. (1998). *Automating Design Intent Capture for Component-based Software Reusability*. M.Sc. Thesis, Massachusetts Institute of Technology.

Smith, I., Stalker, R., & Lottaz, C. (1996). Creating design objects from cases for interactive spatial composition. *Artificial Intelligence in Design '96*, pp. 97–116, Kluwer.

Vadhavkar, S. (1997). *Augmenting Design Patterns with Design Rationale*. M.Sc. Thesis, Massachusetts Institute of Technology.

Feniosky Peña-Mora is an Associate Professor of Information Technology and Project Management in the Civil and Environmental Engineering Department's Intelligent Engineering Systems Group at the Massachusetts Institute of Technology (MIT). Prof. Peña-Mora's current research interests are in information technology support for collaboration, design rationale capture, conflict mitigation, change management, negotiation management, and process integration during design and development of large-scale engineering systems. He is the leader of the Da Vinci Agent Society Initiative at MIT. He is the author of several publications on computer-supported design, computer-supported engineering design and construction, project control, and management of large-scale engineering systems.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press

Sanjeev Vadhavkar is presently a Doctoral Candidate in the Civil and Environmental Engineering Department's Intelligent Engineering Systems Group at MIT. He is currently exploring methodologies to increase effectiveness of geographically distributed engineering teams. As a research assistant at MIT, he has worked in the areas of artificial intelligence, design rationale, design patterns, component-based software, and collaborative software engineering. He graduated with a Master of Science degree from MIT in 1996 and a Bachelor of Technology (Honors) degree from the Indian Institute of Technology in 1995.

Siva Dirisala graduated with a Master of Science degree from MIT and a Bachelor of Technology degree from the Indian Institute of Technology. While at MIT, he was involved in automating design intent capture for component-based software reusability. He is currently a Project Leader in Oracle Corporation. He is presently involved in developing Oracle Exchange, a web-based Business-to-Business Marketplace application. His current interests are creating scalable and interoperable Marketplace/E-business applications using XML messaging, web-based workflows, and content management.

https://doi.org/10.1017/S0890060401152054 Published online by Cambridge University Press