Article

An expert system for the selection of software design patterns

Gary P. Moynihan, Abhijit Suki and Daniel J. Fonseca
Department of Industrial Engineering, The University of Alabama, Box 870288, Tuscaloosa, Alabama 35487-0288, USA
E-mail: Gmoynihan@coe.eng.ua.edu

Abstract: *This paper describes the development of a prototype expert system for the selection of design patterns that are used in object-oriented software. Design patterns provide one method of software reuse, which supports the goal of improved software development productivity. The prototype system represents an initial step towards providing an automated solution regarding the design pattern application problem, i.e. leading a designer to a suitable design pattern which is applicable to the problem at hand. The feasibility of using expert system technology to aid in the selection problem is demonstrated.*

Keywords: expert systems, software design, design patterns, object-oriented

1. Introduction

The relative portion of effort spent on various activities during the software development process has been identified in the literature (Vliet, 2001). Design decisions have a significant impact on the quality of the final product. These decisions directly affect 65% of the work done in subsequent activities. Software design is an iterative process, such that the requirements are transformed into a model for constructing the software. Frequently, the design is first represented at a very high level of abstraction. At this level, the design can be directly traced to specific data, functional and behavioral requirements. Being an iterative process, the design often undergoes changes over the duration of the design phase. Eventually, the design representations are expressed at a much lower level of abstraction. At this level, the connection between the requirements and the design is less obvious.

An object-oriented design differs from these procedural software design methods (e.g. McGlaughlin, 1991). The object-oriented design achieves a number of distinct levels of modularity. Modularity can be explained as dividing software into separately named and addressable components. These modules can be integrated together to form the complete system. 'The unique nature of object-oriented design lies in its ability to build upon four important software design concepts: abstraction, information hiding, functional independence, and modularity. All design methods strive for software that exhibits these fundamental characteristics, but only object-oriented design provides a mechanism that enables the designer to achieve all four with less complexity and compromise' (Pressman, 1997). Although object-oriented software design is usually easier than alternative approaches, designing software is a difficult task, regardless of technique. Jones *et al.* (1998) point out that 'the promise of the object-oriented approach to the systems analysis hinges on correctly partitioning the problem domain into essential classes and objects. Most developers agree that this is no easy task.'

A good reusable and flexible design is difficult to achieve on the first attempt, particularly if it incorporates capabilities of reuse and modification. Despite these difficulties, experienced object-oriented designers produce good designs that are hard to achieve for a novice designer. There is a remarkable difference between the approaches of experienced and inexperienced designers towards solving a software design problem (e.g. Jones *et al.*, 1998). Some of the expert solutions to these problems have proven to be good over a period of time. Thus, there are recurring patterns of classes and communicating objects. These patterns are applicable to specific design problems and make object-oriented designs more flexible, elegant and ultimately reusable. The patterns allow designers to reuse successful designs by basing new designs on prior experience.

A design pattern book by Gamma *et al.* (1995) presents a catalog of 23 design patterns culled from numerous object-oriented systems. It is regarded as the accepted baseline for this type of approach (e.g. Budinsky *et al.*, 1996). Each of these 23 patterns can be applied under certain circumstances, so it has its own applicability criteria. Each pattern is associated with its consequences. Some patterns are similar in purpose. Some are related to others. Application of one pattern may make the application of relevant patterns beneficial to the overall design. This large amount of knowledge is associated with all 23 design patterns. Thus, expertise is needed to determine the suitable patterns for selection.

Design patterns are a valuable tool for the practising software professional. As the benefits of design patterns are becoming more and more evident, the use of design patterns to develop elements of reusable object-oriented software has become an emerging trend. Research done in this area is now being used for commercial applications. Some of these applications, and their associated advantages, are discussed by Cline (1996). However, of the 23 design patterns that can be used for developing reusable object-oriented software, only a few of the patterns can be adopted while developing a single application. A software designer should not have to master all 23 patterns to use them for a specific application. The expert's knowledge used in the selection of design patterns can be analyzed, and heuristics can be formed to develop an expert system. The software designer can then focus on design patterns that are specifically applicable to his/her project.

In this paper, we present the results of developing and validating an expert system for choosing software design patterns. The prototype system guides the designer through the pattern selection process via targeted inquiry regarding the nature of the specific design problem. The system also provides a means of browsing patterns and viewing the relationships between them.

2. Survey of related research

Budinsky *et al.* (1996) developed a computer-based tool that automates the implementation of design patterns. For a given pattern, the user of the tool inputs application-specific information, from which the tool generates all the pattern-prescribed code automatically. The tool incorporates a hypertext rendition of design patterns to give designers an integrated online reference and developmental tool. This automatic code generation enhances the utility of design patterns. Though this tool provides no help in the selection of design patterns, once a pattern is selected, it can automate its implementation.

Eden *et al.* (1997) have presented a prototype tool that supports the specification of design patterns and their realization in a given program. The prototype automates the application of design patterns without obstructing modifications of the source code text from the programmer. The programmer may edit the source code text at will (Eden *et al.*, 1997). The authors have described a prototype that supports the application of pattern specification language routines by the principles of the metaprogramming approach. The automating tool maintains a library of routines and supports their modification, application and debugging. The tool also recognizes the need of the programmer for manual editing of the object program's source code.

In re-engineering legacy code, design patterns are introduced frequently. This improves clarity in the design and helps in future developments. Cinneide and Nixon (1999) have automated the transformations required to introduce design patterns in re-engineering legacy code. They have presented a methodology for the design pattern transformations and have constructed a prototype software tool, called DPT (Design Pattern Tool), that applies design pattern transformations to Java programs. Their methodology has been applied successfully to structure-rich patterns, such as Gamma *et al.*'s (1995) creational patterns (Cinneide & Nixon, 1999).

All of these tools are helpful in automating the application and implementation of design patterns. The tools allow the user to automatically generate code, to a certain extent, once the user decides to apply a specific design pattern. However, the tools are not applicable to the selection of design patterns suitable to the problem at hand. As the number of patterns discovered is increasing, it is difficult for a novice designer to master all of them. A computer-based tool is required that can reduce the number of patterns for consideration to a few suitable ones.

There are very few applications of expert systems for the selection of design patterns. One reason for this could be the relatively new nature of the design pattern concept. Kramer and Prechelt (1996) have developed a system to improve maintainability of existing software. They have incorporated the structural design patterns identified by Gamma *et al.* (1995). In their approach, Kramer and Prechelt (1996) note that design information is extracted directly from C++ header files and stored in a repository. The patterns are expressed as PROLOG rules and the design information is translated into facts (Kramer & Prechelt, 1996). A single PROLOG query is then used to search for all patterns. This tool demonstrates an artificial intelligence approach for discovering design patterns in an existing software. A similar approach can be applied to selection of design patterns for their application during the design stage of new software.

Khriss *et al.* (2000) have presented a pattern-based approach to the correct stepwise refinement of UML static and dynamic design models by application of refinement schemas. They have also described an approach that allows for the construction of intelligent computer-aided software engineering tools that can provide automatic support for the selection, management and application of design patterns for the refinement of design models. This paper substantiates a need for, and also supports the feasibility of, a knowledge-based approach towards the selection of design patterns.

From the cited literature, it follows that a knowledge-based system is an appropriate tool for the automation of routine designs. Although there are quite a few tools to automate the implementation of design patterns, there is a need for further automation in this area.

3. Development of the prototype expert system

A prototype expert system was developed to select suitable design patterns from a pool of candidates, based upon inputs characterizing the problem at hand. Required output includes a guideline on patterns that are most suitable to the problem, as well as additional guidelines on the patterns that can be applied in combination with the suitable design patterns in order to achieve an overall improved design.

3.1. Development approach

The 23 design patterns, identified by Gamma *et al.* (1995), form the basis for this effort. The authors highlighted eight patterns (i.e. abstract factory, adapter, composite, decorator, factory method, observer, strategy, and template method) within this group, because of their simplicity and frequent occurrence in many object-oriented designs. They noted that for a novice designer or a student of design patterns, these patterns provide a good beginning. The resulting prototype expert system focuses on determining the knowledge associated within these eight design patterns and factors that drive the process of applying design patterns.

Accepted expert system methodology identifies three primary steps: knowledge engineering, system development, and system verification and validation. During the initial knowledge engineering phase, the key concepts, relationships and heuristics were identified. The literature (e.g. Budinsky *et al.*, 1996) provides guidance on the selection of design patterns and the path followed by expert designers in the selection. This was the primary source of knowledge for forming heuristics. This baseline was finetuned through a series of interviews that were conducted with the domain experts in the field. These domain experts (who were primarily university faculty) provided guidance regarding the details of the design pattern knowledge, the key concepts in the design pattern domain, the search strategy to be utilized, and the form of the results to be displayed.

During the development phase, the acquired knowledge was organized. Harmon and Sawyer (1990) outline some of the classical strategies for knowledge representation. They note that 'there are five different ways to encode the facts and relationships that constitute knowledge: semantic networks, object–attribute–value triplets, frames, logical expression, and rules. Each method has advantages and disadvantages.' The expert system for the selection of design patterns uses a rule-based scheme of knowledge representation. The selection of a rule-based system was felt appropriate after considering these other schemas. As noted by Landauer (1990), desired properties of a rule-based scheme include 'local specification of the relation by means of rules, which allows for the decoupling of different kinds of knowledge' and 'making domain knowledge explicit'. This implied simplicity of understanding, as well as the associated ease of rule implementation, facilitates later explanation of the knowledge base to the domain expert and external evaluators. Hence, the verification and the validation phase are expedited. Any subsequent modification of the rule-based system is also considered easier (Landauer, 1990).

The environment chosen for the development was object-oriented. Encapsulation, inheritance and polymorphism allow data hiding, sharing of attributes in a common class, and varying implementation of operations depending on the calling object. Inference rules and objects were constructed and formalized. These rules were incorporated as attributes of the objects, thus combining the benefits of both approaches.

During the programming phase, these formal knowledge representations were translated into computer code. Use of the Level5 Object expert system shell facilitated system construction. Selection of this specific shell was based on an investigation of functionality, installation and integration characteristics, as well as compatibility with existing hardware, software and communications assets. Level5 Object is an object-oriented shell that uses expert knowledge and distributed data access to facilitate applications development (Information Builders, 1996). It utilizes Microsoft Windows to provide a flexible, intuitive and expandable environment for delivering knowledge-based systems. The system was developed and delivered on an IBM compatible microcomputer with a Pentium main processor.

3.2. *System architecture*

As noted previously, this project employs the rule-based method of knowledge representation within an object-based environment. By using this approach, we can represent objects, their attributes and associated values, while at the same time we can still use rules for our inferencing procedures. The rules tend to be more intuitive, and are far more transparent than other modes of knowledge representation, thus permitting better understanding of the system and the knowledge base. They are easy to modify compared to other forms of representation. In particular, additions, deletions and

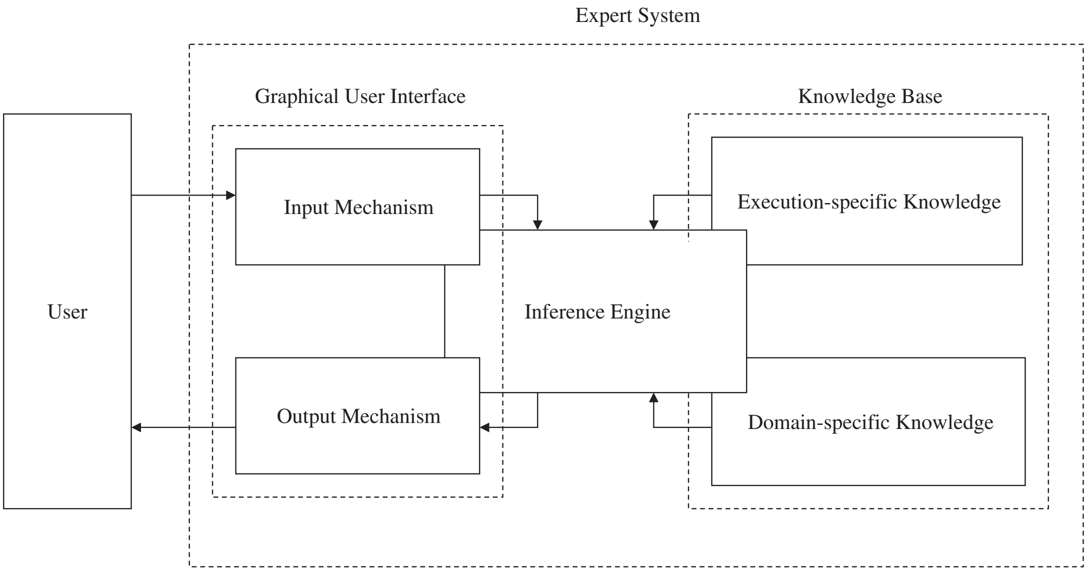


Figure 1: *System architecture.*

revisions to rule bases are relatively straightforward processes.

Consistent with most expert system applications, the prototype system for the selection of design patterns is composed of three primary components: the knowledge base, the inference engine and the user interface (see Figure 1). Although there is a single knowledge base that supports the system, it is partitioned into specific lobes. These reflect two broad categories: domain-specific knowledge and execution-specific knowledge. Domain-specific knowledge pertains to the knowledge about design patterns. This knowledge was captured from the literature baseline, as well as domain expert interviews, during the knowledge acquisition phase. The domain-specific knowledge was further divided into four sub-categories: knowledge on approaches, knowledge relevant to applicability and consequences testing, knowledge to search for a better similar pattern, and knowledge to search for related patterns. These knowledge lobes are triggered in a sequential order during the execution of the system.

The inference engine serves as the processing and control mechanism for expert systems. There are primarily three types of inference engines that can be used to conduct the search: forward chaining, backward chaining and hybrid chaining. In backward chaining, the knowledge base is examined to see if it can establish a value for the goal attribute by moving backwards toward the initial data state (Jackson, 1999). For this project, a backward-chaining search was used. As a first step, the goals and sub-goals were identified. Then, working backwards based on the evaluated numerical calculations, the conclusions and recommendations were drawn.

Figure 2 gives the high level flowchart of the search process. As shown, the system initially attempts to determine the approach of the user towards the selection of design patterns. The knowledge relevant to approaches allows the system to reduce the search of suitable patterns to a more limited subset. The applicability and consequences testing knowledge lobe contains the characteristics of the design patterns. It has the knowledge that is required to judge whether or not a design pattern is suitable to the problem at hand. It consists of eligibility criteria for all the design patterns that are in the scope of this study. This knowledge base also contains consequences associated with each design pattern such as the effects of using a composite pattern.

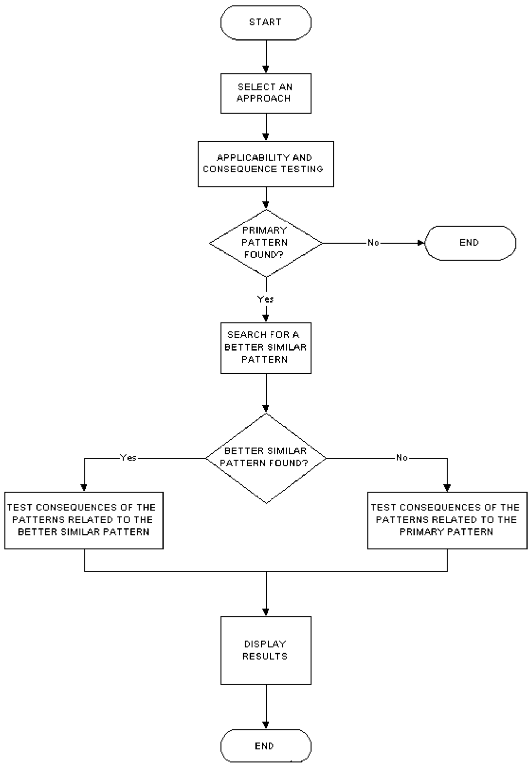

START
SELECT AN APPROACH
APPLICABILITY AND CONSEQUENCE TESTING
PRIMARY PATTERN FOUND?
No
END
Yes
SEARCH FOR A BETTER SIMILAR PATTERN
BETTER SIMILAR PATTERN FOUND?
Yes
No
TEST CONSEQUENCES OF THE PATTERNS RELATED TO THE BETTER SIMILAR PATTERN
TEST CONSEQUENCES OF THE PATTERNS RELATED TO THE PRIMARY PATTERN
DISPLAY RESULTS
END

Figure 2: *Overall search methodology.*

The knowledge lobe for searching for a better similar pattern contains knowledge related to the relationships between various patterns, i.e. the knowledge regarding the circumstances under which one pattern is preferred over the other. This knowledge lobe allows the system to determine a better similar pattern after one pattern is found to be suitable. The final piece of domain-specific knowledge is the related pattern knowledge. All design patterns are linked to each other by various relationships. The selection of one pattern can trigger the application of other supporting patterns. This knowledge allows the system to determine the suitability of related patterns once a given pattern is selected. The related pattern knowledge is used in the final stages of the pattern search.

All the design patterns are related to each other, i.e. the selection of one pattern triggers application of other supporting patterns. This knowledge allows the system to determine the suitability of related patterns once a given pattern is selected. The related pattern knowledge is used in the final stages of the pattern search.

The execution-specific knowledge base captures all the knowledge that is not domain specific. This knowledge facilitates the execution of the system. Some examples of this type of knowledge can be the established goals, the relationship between the user interface and the knowledge elements, and the relationship between various screens. As the knowledge domain increases, the execution-specific knowledge overhead also increases.

The inference engine facilitates the search mechanism. The intelligent behavior of the system is due to the capability of the inference engine to link the knowledge base to reach a goal. The inference engine interacts with the user via user interfaces. This interaction allows the inference engine to understand the problem that the user is trying to solve. The predefined goals drive the search process of the inference engine. In the case of the expert system for the selection of design patterns, the inference engine tries to achieve three goals by completion of the execution cycle. The goals are executed in a sequential order. The goals are interrelated, and the outcome of one goal can affect the execution of the other goals. The goals guide the direction of the search for the inference engine. Depending on the goals, the chain is formed in the knowledge base. When the input of the user is required, the inference engine utilizes the user interfaces to interact.

The user interface facilitates the contact between the system and the user. Depending on the functionality, the user interfaces are divided as input screens and output screens. The input screens are driven by goals that are being investigated. The expert system for the selection of design patterns uses dynamic input screens to display question prompts on a generalized screen. The dynamic nature of the input screen also provides real-time information on the status of the search.

4. Use of the system

The resulting prototype system executes in a Windows environment and utilizes a graphical user interface to promote ease of use. System initiation displays a banner screen. On the screen, there are three buttons to choose from. The 'about the system' option provides information on the system's purpose and scope. The 'exit' button on the banner screen allows the user to quit the execution of the program. The user can then click on the 'continue' button to proceed with system processing. This presents a subsequent display to allow the user to choose one of the approaches for design pattern selection, consistent with the methodology of Gamma *et al.* (1995). The use of radio buttons ensures that only one approach is selected at a time. There are two buttons on the choice screen. Throughout the system, context-sensitive help and explanation screens are available for user support.

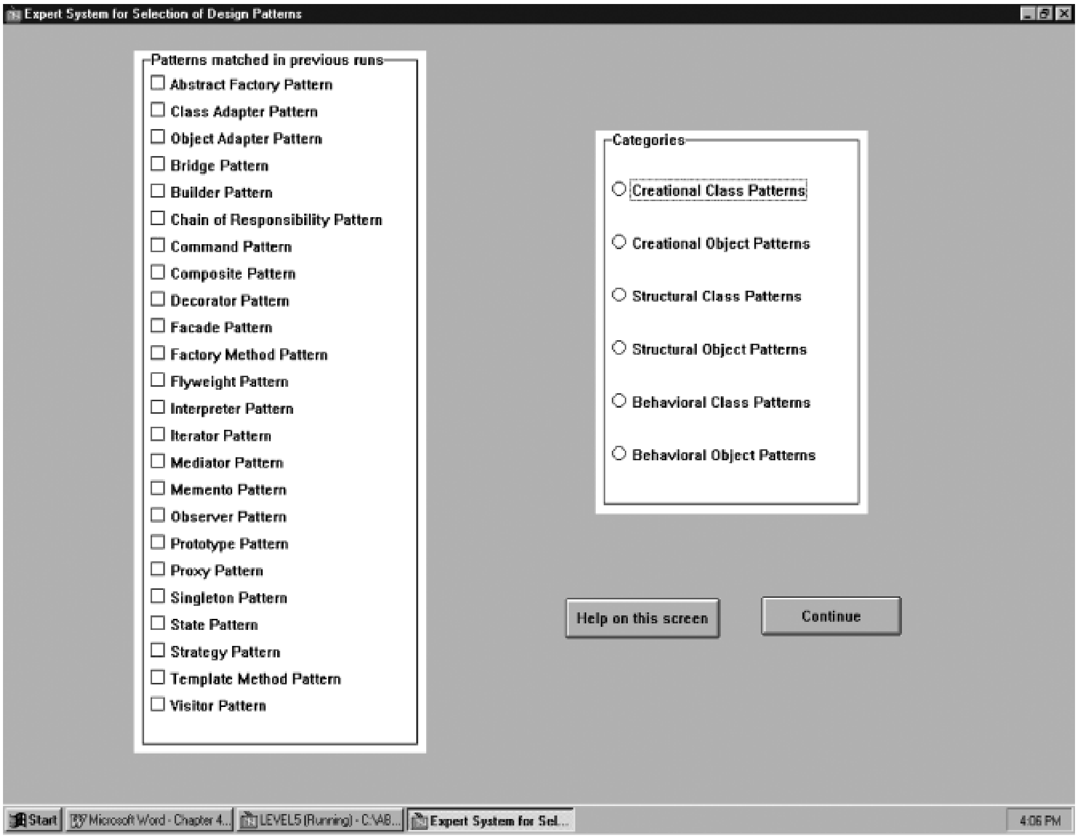
The execution proceeds through a series of display transformation and heuristic activation to obtain the desired goals. Level5 Object utilizes 'when-needed' and 'when-changed' methods to represent aspects of the underlying heuristics (Information Builders, 1996). The when-needed methods allow the system to acquire data for the attributes for which a value is undetermined. They are used to monitor the value of an attribute. When the value of a specific attribute is changed, the method is fired.

The when-changed methods are attached to the buttons on these screens, and facilitate the screen transition. The choice made at the approach screen directs subsequent flow of the system. Depending upon the approach chosen by the user, the when-changed methods display the relevant screens. The user makes several more choices in the screens that follow, which trigger the appropriate demons. These demons reduce the pattern search area to a subset. These candidate patterns become the target for the first goal. The when-changed method, in the meantime, triggers the transition of the display to the applicability and consequences screen. The first goal is set to search for a primary pattern match. This goal drives the chaining of the rules in the applicability and consequence testing knowledge base. When the backward chain is formed, values of some of the attributes are in an 'undetermined' state. These values are needed in order to evaluate the rules. The when-needed methods are used to accommodate this functionality. These methods generate the question prompts, and are displayed using the proper screens to accept response from the user.

The system follows a different sequence of screens depending upon the choice the user makes on the design patterns approach screen. If the 'categorized approach' is chosen, the next screen is the one shown in Figure 3. The patterns are categorized into six categories. On the screen, these categories are shown on the right-hand side. After choosing a category, the user responds to a series of screen prompts to determine a design pattern that is suitable to the problem at hand. There might be more than one pattern, in the same category, that is considered suitable. To accommodate this possibility, the system should be executed for multiple iterations. Previously selected patterns can be eliminated from the search by identifying them with the check boxes on the left-hand side of the screen.

If the user clicks on the 'continue' button, the screen in Figure 4 is displayed. This is used for all of the question prompts that are relevant to applicability and consequences criteria (consistent with the sequence depicted in Figure 2). The questions are displayed in the scrollable text box at the center of the screen. On the right of the text box there is a list of patterns. The patterns under current consideration are highlighted. The user is expected to choose between true and false as a response to these prompts.

After determining various elements of the design problem at hand, and linking back to the desired goal, the inference engine performs numerous iterations. At the end of the expert search, the conclusion on the goal is reached. At this point, the inference engine determines whether the primary pattern is found or not. Depending upon the conclusion of the search, the when-changed methods are fired, which reduce the solution space to a subset of patterns. The output display screen is also modified to reflect the search results. However, the output screen is not displayed to the user until the end of the execution cycle.


Expert System for Selection of Design Patterns

Patterns matched in previous runs
- ☐ Abstract Factory Pattern
- ☐ Class Adapter Pattern
- ☐ Object Adapter Pattern
- ☐ Bridge Pattern
- ☐ Builder Pattern
- ☐ Chain of Responsibility Pattern
- ☐ Command Pattern
- ☐ Composite Pattern
- ☐ Decorator Pattern
- ☐ Facade Pattern
- ☐ Factory Method Pattern
- ☐ Flyweight Pattern
- ☐ Interpreter Pattern
- ☐ Iterator Pattern
- ☐ Mediator Pattern
- ☐ Memento Pattern
- ☐ Observer Pattern
- ☐ Prototype Pattern
- ☐ Proxy Pattern
- ☐ Singleton Pattern
- ☐ State Pattern
- ☐ Strategy Pattern
- ☐ Template Method Pattern
- ☐ Visitor Pattern

Categories
- ○ Creational Class Patterns
- ○ Creational Object Patterns
- ○ Structural Class Patterns
- ○ Structural Object Patterns
- ○ Behavioral Class Patterns
- ○ Behavioral Object Patterns

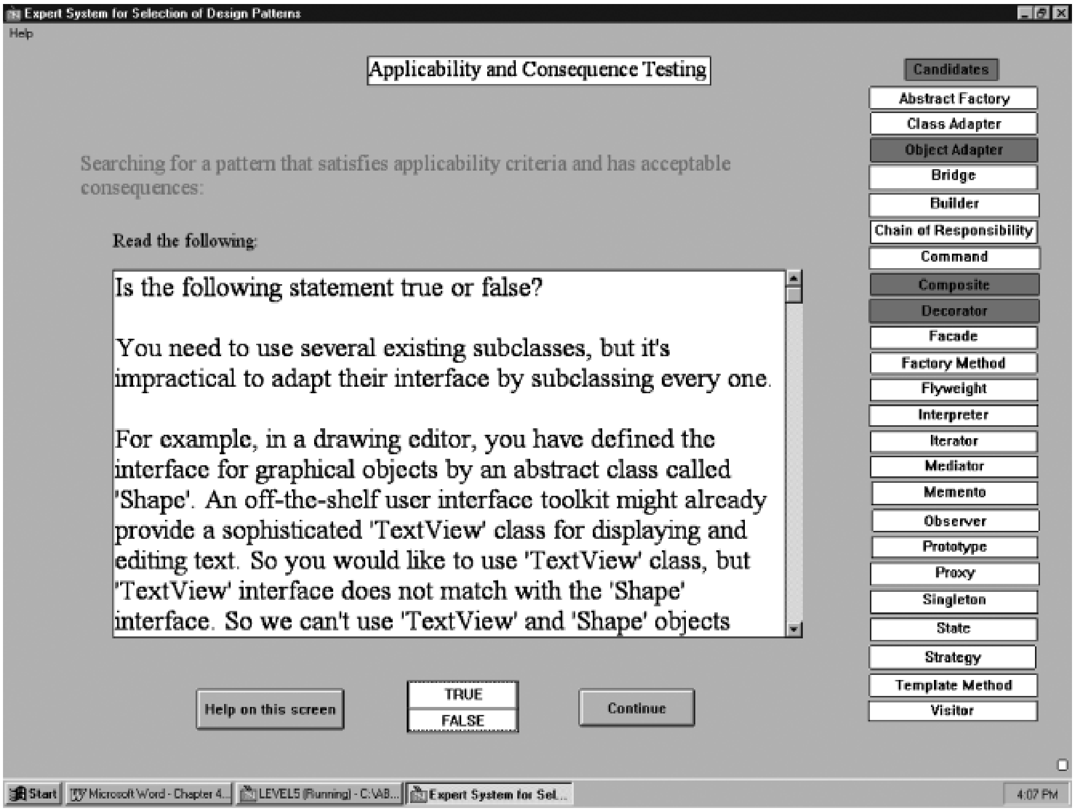
Figure 3: *Design pattern selection approach screen.*

The next goal is established for the search of a better similar pattern. The process for searching a primary pattern match is repeated for this search as well. Once a primary pattern is selected, the better similar pattern screen is displayed. The layout of this screen is similar to the applicability and consequences testing screen (Figure 3). The primary pattern that has already been selected is displayed on the screen above the text box. This screen prompts the user with questions related to the selection of a better similar pattern. The procedure for responding to these question prompts is similar to that of the applicability and consequences testing screen.

The previously mentioned sequence is executed when the user chooses the categorized approach. If the user selects the causes of redesign approach, then the screen depicted in Figure 5 is displayed. The applicable causes of redesign are listed on the right. The application of design patterns avoids these causes of redesign. The check boxes allow the user to choose one or more causes. Depending upon the causes selected, a subset of patterns become candidates for activation. These patterns are tested for applicability and consequences first, and after that point the same logic path is followed by the system.

The screen in Figure 6 represents the solve specific design problem approach. Commonly occurring design problems are listed on the right. Check boxes associated with each design problem facilitate multiple selection. Functionality is similar to that followed in previous approaches.

Gamma *et al.* (1995) both provide, and recommend using, a graphical representation of the relationship between various design patterns.


Expert System for Selection of Design Patterns
Help
Applicability and Consequence Testing

Searching for a pattern that satisfies applicability criteria and has acceptable consequences:

Read the following:

Is the following statement true or false?

You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one.

For example, in a drawing editor, you have defined the interface for graphical objects by an abstract class called 'Shape'. An off-the-shelf user interface toolkit might already provide a sophisticated 'TextView' class for displaying and editing text. So you would like to use 'TextView' class, but 'TextView' interface does not match with the 'Shape' interface. So we can't use 'TextView' and 'Shape' objects

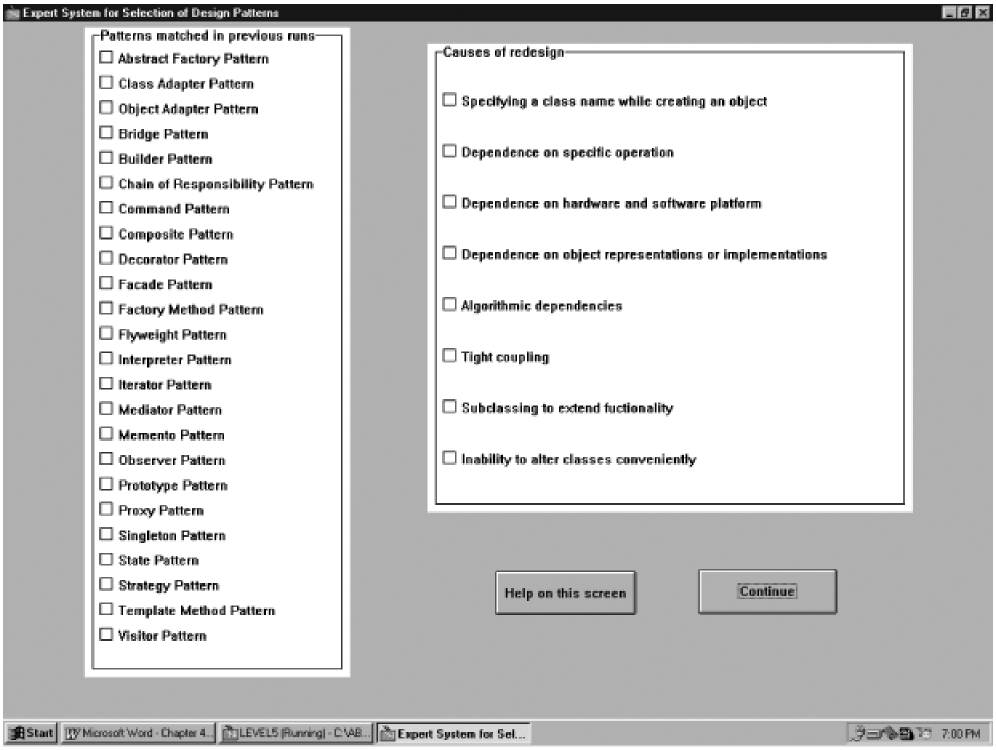
Figure 4: *Applicability and consequences testing screen.*

The screen in Figure 7 is such a graphical representation of the relationship between these design patterns, which are within the scope of the system. By studying the relationship between the design patterns, the user can test the suitability of a pattern by clicking on the button for the pattern of interest. This provides an alternative means of pattern search, starting with the applicability and consequences testing.

If the user chooses to follow the study pattern's intent approach, a display analogous to Figure 4 appears. The text window in the center of the screen displays the intents of all the patterns as stated by Gamma *et al*. (1995). A list of all the patterns is displayed on the right. After reviewing the intent of a pattern, it can be tested for its suitability by clicking on the pattern's name. The system also searches for a better similar pattern and related patterns, if a primary pattern is found suitable.

Once the related pattern goal is attained, the system displays the conclusion screen. This output screen displays the results of all the three expert searches. The primary pattern selected is the one that can solve the problem at hand. At the end of the pattern search, the system concludes that the applicability criteria for this pattern are met. That is, the problem under consideration has the characteristics which satisfy the applicability criteria for this design pattern. However, every pattern has consequences associated with its application. The system concludes that these consequences are acceptable to the user. The application of the primary design pattern can solve the problem at hand without adversely affecting the overall design.

The pattern listed as a better similar pattern provides an alternative for consideration. Under certain situations, one of the patterns is better suited to the problem under consideration than the other. If a better similar pattern is found



Expert System for Selection of Design Patterns

Patterns matched in previous runs
- ☐ Abstract Factory Pattern
- ☐ Class Adapter Pattern
- ☐ Object Adapter Pattern
- ☐ Bridge Pattern
- ☐ Builder Pattern
- ☐ Chain of Responsibility Pattern
- ☐ Command Pattern
- ☐ Composite Pattern
- ☐ Decorator Pattern
- ☐ Facade Pattern
- ☐ Factory Method Pattern
- ☐ Flyweight Pattern
- ☐ Interpreter Pattern
- ☐ Iterator Pattern
- ☐ Mediator Pattern
- ☐ Memento Pattern
- ☐ Observer Pattern
- ☐ Prototype Pattern
- ☐ Proxy Pattern
- ☐ Singleton Pattern
- ☐ State Pattern
- ☐ Strategy Pattern
- ☐ Template Method Pattern
- ☐ Visitor Pattern

Causes of redesign
- ☐ Specifying a class name while creating an object
- ☐ Dependence on specific operation
- ☐ Dependence on hardware and software platform
- ☐ Dependence on object representations or implementations
- ☐ Algorithmic dependencies
- ☐ Tight coupling
- ☐ Subclassing to extend fuctionality
- ☐ Inability to alter classes conveniently

Figure 5: *Causes of redesign.*

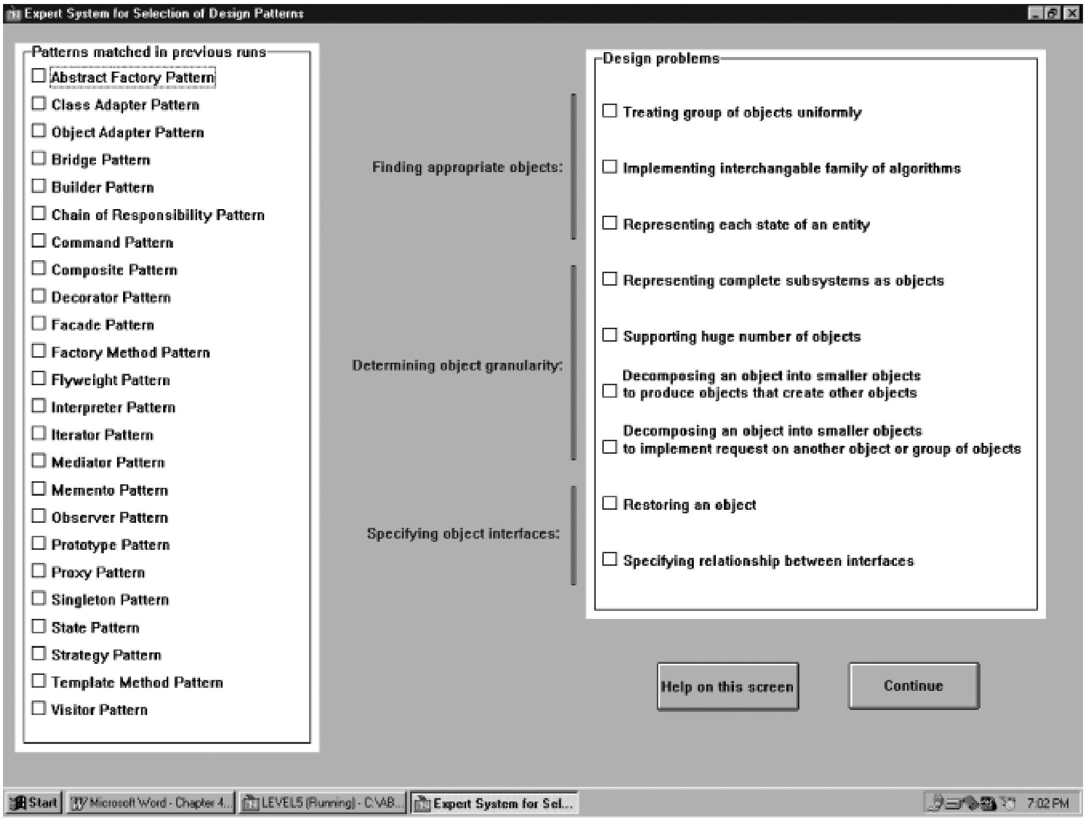
during the search, it will be displayed within this category on the output screen. During a search, a better pattern may not be found. In such cases, no pattern is listed under the category.

The final category listed indicates the related patterns. These are often used in conjunction with the primary pattern. If a better similar pattern is present, then these patterns are related to the better similar pattern. The application of the primary pattern or the better similar pattern can solve the design problem at hand. However, the related patterns used together with the primary pattern or the better similar pattern can improve the design even further.

5. System verification and validation

Verification determines correctness of the expert system, i.e. whether the product satisfies the specification standards set at the beginning of the project. In the expert system's context, verification ensures that the compile time and runtime errors are eliminated. The debugging utilities of the Level5 Object expert shell were utilized throughout the development cycle to ensure error-free execution of the pattern search. Individual programs and modules were evaluated individually by executing a series of predetermined test cases (Laudon & Laudon, 2001). The complete system was then verified to ensure that the integrated modules behaved as expected. Verification aspects common to knowledge-based processing, as noted by Medsker and Liebowitz (1994), were also applied.

Validation determines whether the built product is the right product, i.e. 'the process of evaluating software at the end of the development process to ensure compliance with software requirements' (Institute of Electrical and Electronics Engineers, 1983). The product that satisfies verification is a correctly built product. However, the product may not deliver the expected results. Validation has its focus on the


Expert System for Selection of Design Patterns
Help on this screen
Continue
Start
Microsoft Word - Chapter 4...
LEVEL5 (Running) - C:\AB...
Expert System for Sel...
7:02 PM

Patterns matched in previous runs

- ☐ Abstract Factory Pattern
- ☐ Class Adapter Pattern
- ☐ Object Adapter Pattern
- ☐ Bridge Pattern
- ☐ Builder Pattern
- ☐ Chain of Responsibility Pattern
- ☐ Command Pattern
- ☐ Composite Pattern
- ☐ Decorator Pattern
- ☐ Facade Pattern
- ☐ Factory Method Pattern
- ☐ Flyweight Pattern
- ☐ Interpreter Pattern
- ☐ Iterator Pattern
- ☐ Mediator Pattern
- ☐ Memento Pattern
- ☐ Observer Pattern
- ☐ Prototype Pattern
- ☐ Proxy Pattern
- ☐ Singleton Pattern
- ☐ State Pattern
- ☐ Strategy Pattern
- ☐ Template Method Pattern
- ☐ Visitor Pattern

Design problems

Finding appropriate objects:
- ☐ Treating group of objects uniformly
- ☐ Implementing interchangable family of algorithms
- ☐ Representing each state of an entity

Determining object granularity:
- ☐ Representing complete subsystems as objects
- ☐ Supporting huge number of objects
- ☐ Decomposing an object into smaller objects to produce objects that create other objects
- ☐ Decomposing an object into smaller objects to implement request on another object or group of objects

Specifying object interfaces:
- ☐ Restoring an object
- ☐ Specifying relationship between interfaces

Figure 6: *Solve specific design problem screen.*

logical errors in the system, and was accomplished by applying the accepted techniques of face validation and predictive validation (e.g. Libertore & Stylianou, 1993).

Face validation (essentially having a domain expert evaluate the system at 'face value') is a common approach. The prototype system and associated documentation were reviewed by external evaluators. This validation against expert performance was conducted by 11 design pattern practitioners with a variety of levels of expertise, from students to professional software developers. These evaluators were self-selected by responding to a request on the pattern-discussion electronic mailing, hosted at the University of Illinois. Those who replied to the message, and expressed their interest in reviewing the system, were sent the program and relevant documentation. Feedback was collected and analyzed. The overall evaluation indicates that the system is generating the correct results and will provide a useful tool in assisting in the selection and implementation of design patterns. Several reviewers also provided suggestions for improving the system. These included screen design, displaying status of the search, and error messages. The suggestions resulting from the face validation were carefully considered and the system was modified accordingly.

Predictive validation involved comparing results from test cases in the literature with the actual system output. These test cases resulted in execution and outcomes consistent with the proven results cited by Gamma *et al.* (1995) and Vlissides (1998).

Test case 1 In the structural object pattern category, the system looked for the abstract factory pattern's applicability and then considered its consequences. When the abstract factory pattern was found suitable, the next step

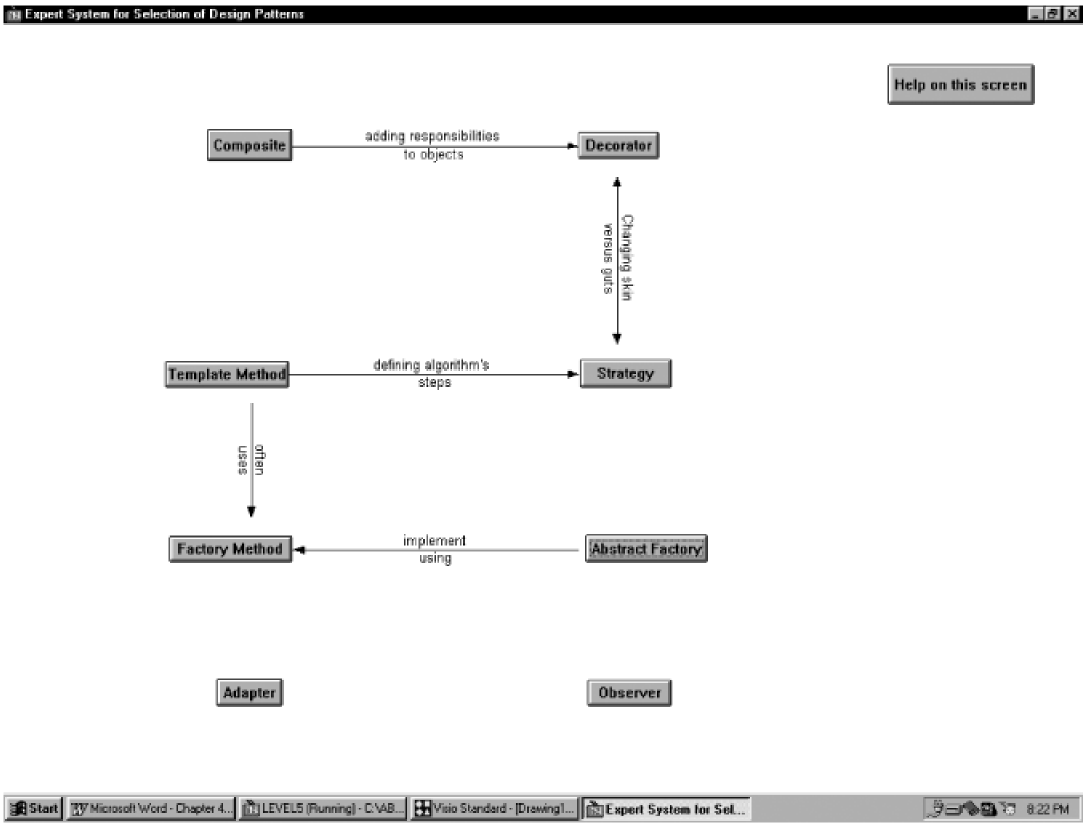

Expert System for Selection of Design Patterns
Help on this screen
Composite
adding responsibilities to objects
Decorator
Changing skin versus guts
Template Method
defining algorithm's steps
Strategy
often uses
Factory Method
implement using
Abstract Factory
Adapter
Observer
Start
Microsoft Word - Chapter 4...
LEVEL5 (Running) - C:\AB...
Visio Standard - [Drawing1...
Expert System for Sel...
8:22 PM

Figure 7: *Relationship between patterns screen.*

was to find a better similar pattern. Since there were no similar purpose patterns for the abstract factory pattern, the system did not search for a better similar pattern. The abstract factory pattern classes are often implemented with the factory method pattern. Thus, the system considered the consequences of the factory method pattern. When the consequences of the factory method pattern were found suitable, the system suggested application of the abstract factory pattern as a primary pattern and the application of the factory method pattern to support implementation of the abstract factory pattern.

Test case 2 To solve a specific design problem such as 'Treating a group of objects uniformly', the system looked for the applicability of the composite pattern. After the applicability was verified, the system considered consequences of the composite pattern. When the composite pattern was found suitable as a primary pattern, the system evaluated the decorator pattern as a better similar pattern. When the decorator pattern was found not suitable as a better similar pattern, the system searched for patterns related to the composite pattern. Since the decorator pattern supports the implementation of the composite pattern, the system tested the decorator pattern for its application as a related pattern. When the decorator pattern was found suitable, the system suggested application of the composite pattern as a primary pattern and application of the decorator pattern to support the implementation of the composite pattern.

Test case 3 The problem under consideration required the following attributes in software design: uniform document structure, implementation of formatting algorithms, and provision for single user protection. The approach taken to search for a pattern to provide uniform document structure was the 'solving specific design problems' approach. The composite pattern was found suitable, with the decorator pattern as a related pattern, to support implementation of the composite pattern. To search for a pattern to provide implementation of formatting algorithms, the intent approach was chosen. The strategy pattern was found to be suitable to provide this functionality. The categorized approach was chosen to carry out the pattern search for the provision of single user protection. The system tested behavioral class patterns, but they were found to be not suitable. The test for behavioral object patterns yielded the template method pattern. The test for the strategy pattern as a similar purpose pattern failed, and the template method pattern was found as the best-fit pattern.

6. Conclusions and further research

The software reuse approach has been widely adopted to improve software productivity and quality. Compared to other approaches that attack the same problems (e.g. computer-aided software engineering), it shows the greatest potential. Among various software reuse techniques, pattern-based reuse has the best flexibility and can be generally applied. It is less specific and is applicable in different languages and different domains. The developed expert system automates the process of selecting and implementing design patterns. Domain knowledge from human experts is represented in the system, and can be easily accessed by users through the user interface. The resulting prototype provides users with detailed guidance on which design pattern or which combination of patterns is suitable to solve the specified problem. Thus far, the prototype has been used for instructional purposes in a computer science course at this university. Expanded instructional use is planned.

The scope of this research was limited to a small portion of design patterns that have been discovered. The prototype expert system includes five design patterns, which are considered to be a frequently encountered subset of patterns (Gamma *et al.*, 1995). A reasonable extension of this research is to enlarge the group of candidate design patterns. The increased number of design patterns will also raise the complexity of the knowledge base exponentially. Special techniques, e.g. partitioning of the knowledge base, may need to be applied. As suggested by some system reviewers, a knowledge base maintenance facility can also be integrated. This facility will allow users to modify the knowledge base by correcting existing rules or adding new rules.

References

Budinsky, F., M. Finnie, J. Vlissides and P. Yu (1996) Automatic code generation from design patterns, *IBM Systems Journal*, **35** (2), 151–171.

Cinneide, M. and P. Nixon (1999) A methodology for the automated introduction of design patterns, in *Proceedings of the IEEE International Conference on Software Maintenance*, New York: IEEE, 463–472.

Cline, M. (1996) The pros and cons of adopting and applying design patterns in the real world, *Communications of ACM*, **39** (10), 47–49.

Eden, A., J. Gil and A. Yehudai (1997) Automating the application of design patterns, *Journal of Object-oriented Programming*, **10** (2), 44–46.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995) *Design Patterns: Elements of Reusable Object-oriented Software*, Boston, MA: Addison-Wesley.

Harmon, P. and B. Sawyer (1990) *Creating Expert Systems for Business and Industry*, New York: Wiley.

Information Builders (1996) *Level5 Object for Microsoft Windows*, New York: Information Builders.

Institute of Electrical and Electronics Engineers (1983) IEEE Standard Glossary of Software Engineering Terminology, *IEEE Standard 729*, New York: IEEE.

Jackson, P. (1999) *Introduction to Expert Systems*, 3rd edn, New York: McGraw-Hill.

Jones, C., T. Hilton and C. Lutz (1998) Discovering objects: which identification and refinement strategies do analysts really use?, *Journal of Database Management*, **9** (3), 3–14.

Khriss, I., R. Keller and I. Hamid (2000) Pattern-based refinement schemas for design knowledge transfer, *Journal of Knowledge-based Systems*, **13** (6), 403–415.

Kramer, C. and L. Prechelt (1996) Design recovery by automated search for structural design patterns in object-oriented software, *Proceedings of the Third Working Conference on Reverse Engineering*, New York: IEEE, 208–215.

Landauer, C. (1990) Correctness principles for rule-based expert systems, *Expert Systems with Applications*, **1** (3), 291–316.

Laudon, K. and J. Laudon (2001) *Management Information Systems: New Approaches to Organization and Technology*, 5th edn, Upper Saddle River, NJ: Prentice Hall.

Libertore, M. and A. Stylianou (1993) The development manager's advisory system: a knowledge-based DSS tool for project assessment, *Decision Sciences*, **24** (3), 953–976.

McGlaughlin, R. (1991) Some notes on program design, *Software Engineering Notes*, **16** (4), 53–54.

Medsker, L. and J. Liebowitz (1994) *Design and Development of Expert Systems and Neural Networks*, New York: Macmillan.

Pressman, R. (1997) *Software Engineering: A Practitioner's Approach*, New York: McGraw-Hill.

Vliet, H. (2001) *Software Engineering: Principles and Practice*, Chichester: Wiley.

Vlissides, J. (1998) *Pattern Hatching: Design Patterns Applied*, Software Patterns Series, Boston, MA: Addison-Wesley.

The authors

Gary P. Moynihan

Gary P. Moynihan is a professor in the Department of Industrial Engineering, University of Alabama, USA. He received BS and MBA degrees from Rensselaer Polytechnic Institute, and a PhD from the University of Central Florida. His primary area of specialization is the application of expert systems to manufacturing. Prior to joining the faculty of the University of Alabama, Dr Moynihan held positions in the aerospace, computer and chemical processing industries.

Abhijit Suki

Abhijit Suki received his BS in production engineering from the University of Mumbai, and an MS degree in industrial engineering from the University of Alabama. He recently accepted a position as a systems analyst with Capital One Services.

Daniel J. Fonseca

Daniel J. Fonseca is at present an assistant professor with the Department of Industrial Engineering at the University of Alabama. He previously taught at the Monterrey Institute of Technology campuses in Mexico City and Tampico. Dr Fonseca received BSIE and MSIE degrees from the University of Alabama, and both MS and PhD degrees in engineering science from Louisiana State University. His areas of research interest include manufacturing systems, artificial intelligence and systems simulation.