THE UNIVERSITY OF CALGARY

The Application of Design Patterns in Knowledge Inference Engine

by

Dong Pan

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JULY, 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-34985-3

Canada

Abstract

Software design patterns are a literature form to describe successful solutions to common software problems. Design patterns are a valuable technique in the software engineering problem-solving discipline. The design patterns capture experts' successful experience, make implicit design knowledge explicit, and explain the deep structure and rationale of a design. The design pattern community has written and documented many design patterns - however, no paper or book has been written in the knowledge representation domain. Description Logic based systems are knowledge representation and reasoning systems that support a richer representation formalism than standard rule based systems. CLASSIC is a small description logic with a well-defined syntax.

In order to verify that design patterns are applicable to the knowledg representation domain, CLASSIC was chosen as a model for the design of a knowledge inference engine. A number of design patterns were applied to the design and implementation of the system. The results show that design patterns are applicable to the domain. In addition, the use of design patterns makes the system more flexible and extensible.

Acknowledgements

The work presented in this thesis could not have been possible without the support of many people. Thanks to my supervisor, Dr. Rob Kremer, for his timely advice, consultations, encouragement, and criticism throughout the development of this work. Rob motivated this work by first introducing design patterns to me and convincing me the value of patterns. He has been an invaluable source of support and guidance throughout my graduate program.

I would also like to thank people at the Software Engineering Group, Knowledge Science Institute lab, and the Computer Science Department at the University of Calgary. These people have been an endless source of inspiration and good advice throughout this work. They include Mildred Shaw, Brian Gaines, Roberto Flores-Mendez, Saul Greenberg, Pim van Leeuwen, Wuji Yang, and John Frankovich. Thanks also go to the department of Computer Science for providing me with such a friendly environment and for its continuous support.

Thanks to Carlos Marques for the thought-provoking discussions we had during the work. Many thanks to Guoqiang Li, and Dr. Yea-Mow Chen, who offered me much help which made it possible for me to enter the graduate program. Thanks to Andy Kremer, who spent tremendous effort in proof-reading this thesis. I really appreciate it.

I would like to thank my Mom and Dad for their encouragement, understanding, and caring. They are always a source of motivation for me. Without them, I could never accomplish what I had done today.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Aim

The aim of this research is to study software design patterns by designing and implementing a knowledge inference engine based on CLASSIC, and evaluating the applicability of design patterns in knowledge representation systems.

1.2 Motivation

Software reuse has long been considered a way to solve the software crisis problems. Software reuse provides a basis for drastic improvement in software quality and developer productivity. However, reuse is not widely practiced in software organizations for a variety of reasons. One reason may lie in people's misunderstanding of the meaning of reuse - some may think of reuse only in terms of code reuse. While code reuse is one kind of reuse, one should recognize that the effort spent in coding is only a small portion of the effort applied to the whole project. Software developers tend to like to create their own code if the code in a reusable library does not fully satisfy their needs, or they do not fully understand the time and space requirements of the piece of code.

Design patterns are abstract descriptions of a solution to a problem under certain constraints. They abstract the solution from many successful designs and describe the solution in an easily understood format. A design pattern is an abstract solution in that it tells one how a problem can be solved without prescribing how the concrete implementation should be done. Hence, the reuse of design patterns may be easier to achieve. On the one hand, programmers are given the solution in a pattern form. On the other hand, the concrete implementation is not given, so the programmer still has much

freedom to apply his or her creativity in the implementation. Thus, programmers will be more likely to reuse design patterns. This is especially true in a volatile environment where software and hardware platforms are under constant change: only design patterns which capture the expertise of the designers will be reused.

Traditionally, novices learn Object-Oriented programming by first learning basic concepts, then reading others' programs, and then trying to program in the language. Through trial and error, the novice gains experience and learns various design patterns through abstracting from these programs even though design patterns are not used explicitly. If the programs are not well documented, the learning is harder, with more time and effort required. A programmer has to study implementation details: because design patterns are obscured in implementation details, the learner has to look at the programs in-depth in order to understand them. Sometimes, it is even impossible to understand certain design decisions by looking only at the implementation (Pree, 1995). If the programs are well documented and design patterns applied are described explicitly, the learning effort will be much reduced.

For system maintainers or those who join a project in the middle of development, the same problems exist as for novices learning by studying others' programs. These people must study the system to understand the design of the system. With the help of design patterns, people can look at the system at a higher abstraction level. Many design rationales are described by design patterns, so the design can be more easily understood. Controlled experiments show that maintenance work can be done in less time and with fewer error if the system is documented in design patterns format (Prechelt, Unger, Philippsen, and Tichy, 1997).

Knowledge representation and inference is a relatively mature domain. It has undergone many years of evolution and many successful systems have been developed, such as KRS

(Gaines, 1993; Gaines, 1995) and NeoClassic (Patel-Schneider, Abrahams, Resnick, McGuinness, and Borgida, 1996). These systems have been applied to many real applications, such as configuration management (Wright et al, 1993), data mining (Brachman et al, 1993), etc. Though this is a relatively mature domain, and design patterns have been around for several years, the author noted that there is no publication on design patterns in the knowledge inference domain. The author wonders why no one is doing this work. Is it because the domain is so special that design patterns are not applicable to it?

The author himself believes that design patterns should be applicable to knowledge inference software. The research will include a research of a related domain, the implementation of a knowledge inference system, and an examination of whether design patterns are applicable to the system.

1.3 Design Patterns

Software design patterns are a literature form to describe successful solutions to common software problems. They are insightful nuggets of information that capture the essence of a successful family of solutions to recurring problems. Each design pattern is described in a certain format. Most of design patterns are described in a format called Alexandrian form.

In Alexandrian form, the description of each pattern consists of the pattern name, the intent of the pattern, the context where the problem occurs, forces (tradeoffs), solutions (may include the structure of the solution), rationale, examples of using the pattern, and known uses which describe systems in which the pattern has been used. To provide the readers with a concrete feeling of design patterns, the following uses the Proxy pattern (Gamma et al, 1994, pp.207-217) as an example. The description of the pattern is not

intended to be complete. Readers interested in the pattern should refer to the original text for a more detailed description.

Pattern Name:	Proxy
Intent:	Provide a surrogate or placeholder for another object to control access to it (Gamma et al, 1994, p.207).
Context:	In cases when a remote object needs a local representative, or there is the need to control the creation of expensive objects, access to objects, or other additional operations on original object, a proxy object is needed.
Forces/Tradeoffs:	One level of indirection is introduced by the pattern when accessing the original object. Many operations can be added through the level of indirection depending on the kind of proxy.
Solutions:	The pattern consists of three classes: the proxy class, the subject class, and the real object class. The subject class defines a common interface for proxy and real object classes so that a proxy can be used anywhere a real object is expected.
Known Uses:	ET++, NEXTSTEP, etc.

For software developers, design patterns are another valuable method that complements those existing methods (Gamma, et al, 1994, p.353). Patterns capture obscure but important practices and make implicit knowledge explicit. They provide a structural and easily understood form for documenting and sharing successful experience among developers. Patterns help improve communication among developers by providing a common vocabulary which has a higher abstraction level. The use of patterns in system

development enables the reuse of software architecture. Patterns can also help one learn existing systems or teach novices good design. The topic of design patterns will be described further in chapter 2.

The application of design patterns in system design can generate software that is more robust (Gamma, et al, 1994, p. 24). The systems will be more extensible and flexible. In addition, if design patterns in the system are explicitly documented, the maintenance and learning effort will be much reduced because design patterns encompass many design rationale.

1.4 Description Logics and CLASSIC

Description logics are languages tailored for expressing knowledge about concepts and concept hierarchies. They can be seen as variable free first order term languages. In such systems, one starts with primitive concepts and roles, and can use the language constructs (also called *description constructors*, such as intersection, role quantification, etc.) to define new concepts. Concepts can be considered as sets of individuals, whereas roles are binary relations between individuals. The main reasoning tasks are classification and subsumption checking. Subsumption represents the *is-a* relation where the more general concept is the parent of a more specific one.

CLASSIC is a small description logic language. The language is composed of primitive concept, concepts, roles, rules, and individuals. Complex descriptions are built from simple ones by using *description constructors*. CLASSIC defines eight language constructs, such as intersection (**and** constructor), role quantification (**atLeast**, **atMost** constructors), value restriction (**all** constructor), etc. These constructors will be described in more detail in section 3.2.

The knowledge inference domain is a relatively mature one where many systems have been developed and used in a variety of applications. The requirements for the knowledge inference system developed in this research were obtained mainly by studying existing systems, such as KRS (Gaines 1995) and NeoClassic (Patel-Schneider et al, 1996). The major function is to check the coherence of a description, to compute the subsumption relationship, and to classify concepts and individuals.

1.5 Objectives

The primary objective of this thesis is to evaluate the applicability of design patterns in the knowledge inference domain. The method used to achieve the objective is to develop an actual knowledge inference system, and check whether design patterns can be applied in the design and implementation of the system.

Because a knowledge inference system itself is very broad and complex, the work of this research will not focus on developing a full-fledged knowledge inference system. However, the system should support basic knowledge inference functionality. CLASSIC was chosen as the model for the system because it is relatively simple and there exists much literature about it. Though the system is intended to be a test case for evaluating ideas, this intent does not mean that the system cannot be further extended to a fully functioning knowledge inference system. The system should, however, be implemented in a principled way so that it implements basic functions, but it should be able to be extended easily in the future. Extensibility is one requirement for the system design.

The following auxiliary objectives can be derived from the primary objective:

1. Studying software design patterns and developing an in-depth understanding of them;

2. Designing and implementing a knowledge inference engine;

3. Applying design patterns to the design and implementation of the system as appropriate;

4. Documenting the design patterns in the context of the system.

The research work is centered around the objectives discussed above. These objectives are discussed again in chapter 4 when discussing the requirements of the knowledge inference system. The evaluation of whether design patterns are applicable to the knowledge inference domain depends on the design and implementation result. If design patterns are applied in the system design, the conclusion that design patterns are applicable can be drawn. Otherwise, negative conclusions can be drawn.

1.6 Summary

This chapter has briefly introduced design patterns. A design pattern is an abstract description of a solution to a problem under certain constraints. Design patterns capture expert experience and are abstracted from many successful designs. Design patterns can help develop more flexible and maintainable software. In addition, design patterns can help novices learn good programming faster and understand existing systems better.

People in the pattern community have done much work on design patterns. The work that has been done covers nearly every aspects of software development. Many books and papers have been published which document patterns and the experience of using patterns in various domains.

Description logics are languages tailored for knowledge representation. Knowledge inference is one specific application that description logics can be used for. Though the

domain has undergone many years of evolution and design patterns have been around for several years, the author noted that no one has done design patterns related work on the knowledge inference domain.

The primary objective of this research is to evaluate the applicability of design patterns in the knowledge inference domain by implementing such a system based on CLASSIC, a small description logic language. The system should support basic knowledge inference functionality, and be flexible enough to extend to support more complex functionalities.

1.7 Thesis Structure

Chapter 2 describes design patterns in detail, and describes the benefits that design patterns can bring to software development.

Chapter 3 provides background knowledge about description logics, especially CLASSIC, so that the reader of the thesis can understand the work described in the thesis more easily.

Chapter 4 describes the requirements analysis for the knowledge inference engine to be implemented in this research. The requirements fall into two categories: the function of knowledge inference, and robustness (flexibility and extensibility) of the system.

Chapter 5 describes the actual design and implementation of the knowledge inference system. The description is also divided into two parts: the core functionality of the system, and extensibility of the system. Much of the description refers to the design patterns used, and discusses how those design patterns fit into the context of the design.

Chapter 6 concludes the thesis and gives future directions of work.

Chapter 2 Background: Design Patterns

This chapter mainly discusses patterns, especially software design patterns. Every mature engineering discipline has handbooks to describe successful solutions for known problems. Software design patterns are a literature form to describe successful solutions to common software problems. Industrial experience has proven that patterns are a valuable technique in the software engineering problem-solving discipline. Not only do patterns capture successful experience, they also help improve communication among designers. They can help new developers avoid the traps and pitfalls that traditionally have only been learned by costly experience. Patterns do more than just describe solutions, they provide rationale behind the solutions.

Section 2.1 gives a brief description of the origin of patterns. Section 2.2 gives a definition of patterns. Though there is no standard definition to patterns, the section gives a generally accepted description of patterns and the properties a pattern should possess. Section 2.3 discusses the values of patterns. It is these values that motivate many software practitioners working on patterns. Section 2.4 describes what a pattern is composed of. Section 2.5 provides a general description of the current state of pattern community, and the activities of people in the community. Section 2.6 gives two examples of design patterns and their applications in real world projects.

2.1 Origin of Patterns

The concept of pattern has been around for a long time. The current use of the term "pattern" in the software community is derived from the writings of the architect Christopher Alexander. Alexander noted that the ultimate purpose of all design and engineering is to fit human needs and comfort, to improve human conditions. He found

recurring themes in architecture and captured them in descriptions that he called pattern. He uses the term "pattern" to represent the replicated similarity in a design, and in particular the similarity that makes room for variability and customization in each of the elements.

In his books, Alexander describes patterns as:

> "Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing" (Alexander, 1979, p.247).

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander, 1977, p.x).

Software designers found analogies between Alexandrian patterns and software architecture patterns. The vocabulary of software patterns, such as ôforcesö, the term ôpatternö itself, and ôpattern-languageö, comes from Alexander. Software design patterns became popular with the unveiling and wide acceptance of the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson and Vlissides, 1994).

Initially, most of the patterns were software design patterns. But patterns are not restricted to software design (Coplien, 1997b; Appleton, 1997). Patterns appear in all aspects of software engineering, including development organization, software process,

project planning, requirement engineering, etc. There is a body of literature for each kind of pattern. Still, software design patterns seem to be the most popular form.

2.2 Definition of Patterns

Patterns are a new topic emerging from the Object-Oriented community. They are a literary form in the software engineering problem-solving discipline. Patterns have roots in many disciplines, including contemporary architecture, literate programming, and documentation of best practices and lessons learned in all vocations.

Patterns are insightful nuggets of information that capture the essence of a successful family of solutions to recurring problems that arise within a particular domain. They usually involve some kind of architecture or organization of constituent parts to produce a greater whole.

Within the context of software development, patterns can be considered as representing recurring structural or behavioral solutions in software. The software engineering community has been borrowing from the discipline of architecture to help categorize, communicate, and document these problem-solving patterns.

A pattern should clearly describe the forces involved in a problem. Generally, any problem in design requires balancing opposing or contradictory forces, and the same is true in software. Many software patterns deal with common design problems: such as how to design a group of objects to cooperate to achieve some goals in the presence of other considerations such as performance or maintenance.

The goal of the software pattern community is to build a body of literature that will help software developers resolve common difficult problems in design and development. Patterns help create a common vocabulary for communicating insights and experience

about these problems and solutions. The focus of patterns is more on creating a culture to document and support sound design than on technology (Appleton, 1997).

Patterns do not represent someone's new idea of how to solve problems, but are descriptions of solutions that have been proven successful in a number of systems (Coplien, 1997b). The longer a pattern has been applied successfully, the more valuable it tends to be. Patterns do not represent principles or rules one must follow, but are practical advice about how to balance forces so as to work one's way out of difficult design and implementation situations.

The Patterns Definition section of Pattern Home Page gives a clear and concise definition for the term pattern within the context of software development:

> "Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves" (Coplien, 1997b, p.3).

A pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. A pattern does more than just identify a solution, it also explains why the solution is needed.

A good pattern will possess the following attributes (Coplien, 1997b, p.2):

- "It solves a problem: Patterns capture solutions, not just abstract principles or strategies.
- "It is a proven concept: Patterns capture solutions with a track record, not theories or speculation.

- "The solution is not obvious: Many problem-solving techniques, such as software design paradigms or methods, try to derive solutions from first principles. The best patterns generate a solution to a problem indirectly û a necessary approach for the most difficult problems of design.

- "It describes a relationship: Patterns do not just describe modules, but describe deeper system structures and mechanisms.

- "The pattern has a significant human component (minimize human intervention). All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility."

2.3 The Value of Patterns

Industrial experience has shown that patterns have a number of valuable attributes. It is also these values that motivate many of the software practitioners who are writing, mining, and teaching patterns. Patterns capture obscure but important practices and make implicit knowledge explicit. They provide a structural and easily understood form for documenting and sharing successful experience among developers. Patterns help improve communication among developers by providing a common vocabulary which has a higher level of abstraction. The use of patterns in system development enables the reuse of software architecture. Patterns can also help one learn existing systems or teach novices good design.

The following discusses various valuable attributes of patterns one by one in more detail.

- **Capture Expert Knowledge**: By definition, patterns capture successful experience. Every pattern is extracted from existing, working designs and is not

created without experience. Design patterns capture the essence of working designs in a form that makes them reusable in future work. They capture important structures, practices, and techniques that are key competencies in a given field, but which are not yet widely known. As Coplien once wrote in (Coplien 1997a, p.39): "*Pattern's biggest payoff may lie in capturing the great truths that are about to be lost to history*".

- **Knowledge Sharing**: Software developers tend to reuse designs that have worked well for them in the past. As they continue their careers, their repertoire of design experience grows and they become more proficient. Unfortunately, this reuse is usually restricted to individual experience, and there is usually little sharing of design knowledge among developers (Beck, Coplien, Crocker, Dominick, Meszaros, Paulisch and Vlissides, 1996). Design patterns capture design knowledge and provide a mechanism for easily sharing design knowledge among developers. They can be quickly understood by both senior and junior developers. Other approaches have been less successful in bridging this gap.

- **A Common Design Vocabulary**: Pattern names provide a common vocabulary for software developers to use in effectively communicating, documenting and exploring design alternatives. Vlissides considers communication as the biggest payoff of design patterns (Beck et al, 1996). Beck also notes that design patterns solve a limited (but critically important) set of communication problems with team development, and make individuals more productive (Beck et al, 1996). Discussing designs in terms of patterns raises the level of communication. The system will seem less complex because patterns enable one to talk about the system at a higher level of abstraction than that of a design notation, separate classes, or components of program languages. Design patterns also make

communication more precise, more concise, more complete, and less likely misunderstood.

- **Enabling Reuse of Software Architecture**: The underlying operating system and hardware platform of the system will often significantly affect design and implementation decisions. In a volatile environment, reusing design patterns is often the only viable means of leveraging previous development expertise (Schmidt 1995). Even though the operating system and hardware platform change, the patterns themselves can be reused. Only portions of the pattern implementation have to be re-implemented to fit platform characteristics. Thus, project risks and development efforts can be greatly reduced.

- **Learning Aid and Training**: Many large systems use design patterns. If one does not understand the design patterns used in the system, it is difficult to follow the flow of control of the system and understand the system. Learning design patterns can help one understand existing systems faster (Gamma, Helm, Johnson, and Vlissides, 1994). Design patterns provide solutions to common problems and describe the deeper system structures and mechanisms. They often encourage good design not by admonishing against mistakes, but by presenting a positive set of habits. Giving novices the opportunity to learn from positive examples can speed their learning.

2.4 Components of Pattern

Alexander says that every pattern must be formulated in the form of a rule which establishes a relationship between a context, a system of forces which arises in that context, and a configuration, which allows these forces to resolve themselves in that context (Alexander, 1979). In the software community, most patterns are expressed in a

format called Alexandrian form. Though several formats for describing and documenting software design patterns exist, it is generally agreed that a pattern should consist of the following components:

The Pattern Name

Each pattern should have a meaningful name. The name allows one to use a single word or short phrase to refer to the pattern, and the knowledge and structure it encompasses. Good pattern names form a vocabulary for discussing conceptual abstraction.

Problem/Intent

A statement describes the problem the pattern is trying to solve, the goal and objectives it wants to achieve within the given context and constraints of the problem. If designers know the problem the pattern solves, they will know when to apply it.

Context

A pattern solves a problem in a certain context. The context is the set of conditions under which the problem and its solution seem to recur, and for which the solution is desirable. It tells of the pattern's applicability.

Forces/Tradeoffs

This description represents relevant forces and constraints of the problem and how they interact/conflict with one another. Forces make clear the intricacies of a problem and define the kinds of tradeoffs that must be considered. A good pattern resolves one or more forces.

Solutions

A description represents the static structure, dynamic behavior, etc., of the solution. This is often equivalent to explaining how to build the solution. It may consist of diagrams, pictures and prose that identify the pattern's structure, participants, and their collaborations, and shows how the problem is solved.

Examples

This section gives one or more sample applications that use the pattern. Examples help readers understand the application of the pattern.

Resulting Context/Force Resolution

This section describes the state or configuration of the system after the pattern has been applied. It also describes what forces have been resolved, which ones are left unresolved, what other patterns may now be applicable, and how the context is changed by the pattern.

Rationale

This section describes how the pattern actually works, why it works, and why it is good. The solutions section describes the visible structure and behavior of the system, while the rationale provides insight into the deep structure and key mechanisms that lie under the surface of the system.

Known Uses

This section describes known occurrences of the pattern and its application within existing systems. It helps to validate the pattern by verifying that it is indeed a proven solution to a recurring problem.

These are the components that a pattern description should have. While writing patterns some authors may wish to combine several components into one, or separate one component into several components. Thus the format of a pattern may vary from author to author.

2.5 The Pattern Community and Activities

Design patterns is relatively young compared with other disciplines in computer science. It began to be widely accepted only after the first PLoP (Pattern Languages of Programs) conference in 1994. Since then, pattern has gained wide popularity and has become a hot topic in the software community, especially among Object-Oriented developers.

Now there are many ways for people in the pattern community to meet and discuss patterns and other related topics. Many conferences are dedicated to patterns: PLoP, EuroPLoP, ChiliPLoP, OOPSLA, and many more. These conferences provide developers and researchers with the opportunity to present and review patterns related to software design, process or organization. The Internet, with its quick and easy access to information, is widely used by the pattern community to exchange and discuss ideas. There are WEB sites dedicated to patterns, such as the pattern home page at University of Illinois (http://hillside.net/patterns/) where people can get general information about patterns, and the Wiki Wiki Web page which is an editable page where people can gather information on patterns and share ideas on pattern topics. Several mailing lists are set up for pattern discussion. The mailing list is very active: each day there are tens of messages flowing on the mailing list, discussing patterns and issues related to patterns.

The work that people in the pattern community are doing is mainly concentrated on the following three areas:

1. Writing papers and books documenting patterns.

2. Applying design patterns to actual projects and documenting the experience of applying patterns

3. Evaluating the usefulness of patterns

A wide variety of pattern papers and books have been published since 1994. Of these publications the one that has had the most influence is the book *Design Patterns: Elements of Reusable Object Oriented Software* by Gamma et al (Gamma et al, 1994). The book is actually a pattern catalog that contains 23 patterns. These patterns are described in the format given in section 2.4. All the patterns documented in the book are general-purpose patterns that can be applied in any domain of software design. Designers have found the book very helpful in designing flexible and extensible systems. It is a major patterns reference in this research. There are many other publications similar to the above book, such as *A System of Patterns* by Buschmann et al (Buschmann, Meunier, Rohnert, Sommerlad and Stal 1996), *Pattern Languages of Program Design* edited by Coplien and Schmidt (Coplien and Schmidt, eds, 1995). These books and papers are the major source of the patterns that are examined and evaluated for their applicability in the proposed system.

The second area of activity for people working in the pattern community is applying design patterns in actual projects, and documenting the implications of using patterns in a system. Many people have published their experience in applying design patterns, and these publications cover a wide variety of real projects. The application domains that have reported the use of design patterns include graphical user interface design (Gamma

1991), communication software (Schmidt and Stephenson, 1995; Schmidt, 1995a), management information systems (Brown 1996), concurrent systems (Schmidt, 1995b), telephony control prototypes (Duell, 1996), etc. All of these reports suggest that the application of design patterns has had a positive effect on the system. These experiences show that the application of design patterns in the system may result in the system gaining one or more of the benefits discussed in section 2.3 - more flexible system, enabling reuse, easily understood and maintainable code, etc.

Although many people have reported their experience in applying design patterns in different domains of software systems, there is no report concerning about the application of design patterns in the artificial intelligence domain, at least not that the author knows about. Is it because this domain is so different from other software domains that design patterns are not applicable to this area? This question is the main motivation for this research.

The third kind of activity that the people in the pattern community are invloved in is evaluating the usefulness of patterns. The results of experience reports are usually based on observations of practitioners. They are not quantitative analysis of the effects. The papers in this category intend to analyze the effects of using design patterns quantitatively through controlled experiments. There are very few people doing this work. The reason for this is probably that the effort and resources needed to perform such activity is too great - it needs several developers working on at least a medium-sized project for several days. Too small or simple a project cannot actually reflect the benefits of applying design patterns. The experiment done by Prechelt et al (Prechelt, 1997; Prechelt et al, 1997) is an example of such activity. The experiment also suggests that patterns will make the maintenance and modifications to the software systems easier and less error prone.

2.6 Design Patterns and Their Applications

Patterns exist in every aspects of software engineering. There are organization patterns, process patterns, test patterns, etc. Each kind of pattern solves specific problems in that domain. However, the focus of this thesis is on software design patterns. These design patterns capture the essence of software design, and make implicit design knowledge explicit. Some software design patterns are general-purpose patterns, which can used in every domain. Most of the patterns catalogued in the *book Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al, 1994) are general-purpose patterns. Some patterns are domain specific, which are captured in a specific software domain and are applicable to only that domain. The Model-View-Controller pattern (Buschmann et al, 1996), for example, is one of this latter kind of pattern, which is used to build flexible user interfaces for interactive applications.

To illustrate design patterns and their applications in the industry, this section describes two examples. The first example describes the Model-View-Control pattern. The second example illustrates how patterns are applied to develop an extensible and maintainable Object Request Broker (ORB) middleware.

2.6.1 The Model-View-Controller Pattern

The Model-View-Controller (MVC) pattern (Buschmann et al, 1996, pp.125-144) provides a means to build interactive applications with a flexible user interface. In interactive systems, the functionality is often relatively stable, but user interfaces are more prone to change. The MVC pattern divides the system into three components - *model*, *view*, and *controller* - so that changes to the user interface will not have major effects on the application-specific functionality.

User interfaces in the MVC pattern are composed of *views* and *controllers*. The *model* component encapsulates application core data and functionality. The *view* component displays information to users. The *controller* component accepts input, such as keystrokes or mouse clicks, from users and translates this input into service requests to the model or view components. Each model can have multiple views so that the same information can be displayed in different ways. Each view has a controller component to handle user input. Whenever data in a model changes, the model notifies all views associated with it. The views in turn obtain data from the model and update the displayed information.

The class structure (Buschmann et al, 1996, p.129) of the Model-View-Controller pattern is shown below:

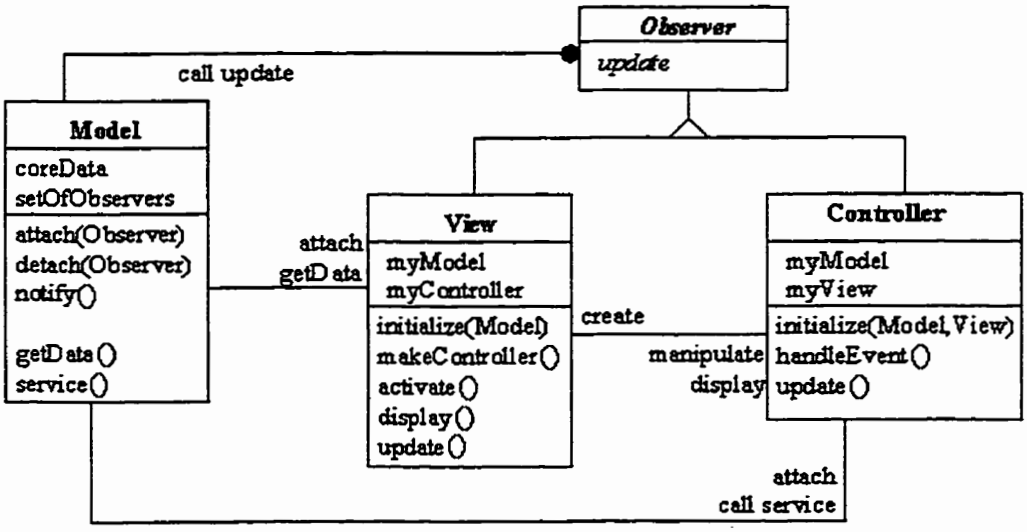


Figure 2.1 Class Structure of Model-View-Controller Pattern Model

Model

Model provides the functional core of the application. It manages views and controllers dependent upon it. When data in it changes, the model notifies dependent components about the data changes.

View

View is responsible for creating and initializing its associated controller. View retrieves data from model and displays information to the user. It must implement an update method to handle data change notifications from model.

Controller

Controller accepts user inputs and translates user inputs into service requests for the model or display requests for the view. If necessary, it will also implement an update method to handle data change notifications from model.

The MVC pattern is probably the best-known pattern for handling the user interface of interactive systems. It was first implemented within the Smalltalk-80 environment (Krasner and Pope, 1988). It has been widely used in a number of software systems or application frameworks, such as MacApp (Apple 1989), ET++ (Gamma, 1991), etc.

2.6.2 Applying Design Patterns to Develop TAO Project - A Case Study

TAO (Schmidt and Cleeland, 1997) project is a real-time endsystem developed by the Distributed Object Computing group at Washington University. TAO stands for The ACE (ADAPTIVE Communication Environment) ORB. The ACE is an object-oriented framework that provides a rich set of components to perform common communication software tasks across a range of OS platforms. ORBs (Object Request Brokers) are the

heart of distributed object computing, which enables the collaboration of local and remote applications in heterogeneous environments.

Design patterns have been used to develop TAO so that the system can be extended and maintained more easily. TAO is designed as a dynamically configurable middleware that overcomes the drawbacks of inflexibility and inefficiency of statically configured ORBs. TAO enables the ORB developers to selectively integrate customized key ORB strategies, such as communication, concurrence, demultiplexing, scheduling, and dispatching. The extensibility of TAO mainly addresses: *extensible to retargeting on new platforms, extensible via custom implementation strategies, and extensible via dynamic configuration of custom strategies* (Schmidt and Cleeland, 1997). This is mainly because of the nature of distributed computing - such systems usually run in a heterogeneous environment.

Eight design patterns have been applied in the development of the ORB architecture for TAO, as described below:

- **The Wrapper Facade pattern**: This pattern is a variant of the Facade pattern (Gamma et al, 1994). It is used to encapsulate low-level stand-alone system mechanisms within type-safe, modular and portable class interfaces.

- **The Reactor pattern**: In order to make ORB implementation independent of any specific event demultiplexing mechanism, and decouple its demultiplexing code from its handling code, the Reactor pattern (Schmidt, 1994) is used in the implementation of TAO. The intent of this pattern is to support synchronous demultiplexing and dispatching of multiple event handlers, which are triggered concurrently from multiple clients.

- **The Acceptor and Connector patterns**: A key responsibility of the ORB core is to manage connections. To support multiple transport mechanisms and allow connection-related behavior to be reconfigured flexibly late in the design phase, the Acceptor and Connector patterns (Schmidt, 1996c) are applied. The intent of these two patterns is to decouple the service initialization from the tasks performed once the service is initialized. The Acceptor pattern is responsible for passive initialization (server side); the Connector pattern is responsible for active initialization (client side).

- **The Active Object pattern**: Concurrency is an important issue in the implementation of ORB. Long-running tasks should not block the processes of other tasks. The Active Object pattern (Lavender and Schmidt, 1995) provides an effective way to support a simple, extensible, and portable concurrency mechanism.

- **The Strategy pattern**: To support transparent interchangeability of multiple ORB strategies, the Strategy pattern (Gamma et al, 1994) is applied. A number of communication, concurrency, demultiplexing, and scheduling algorithms are encapsulated as strategies in the TAO system.

- **The Abstract Factory pattern**: Due to the extensive use of the Strategy pattern, the system contains a large number of strategy classes. In order to simplify the management of a large number of classes and enforce semantic consistency when composing different strategies, the Abstract Factory pattern (Gamma et al, 1994) is applied.

- **The Service Configurator pattern**: In order to enhance the dynamism of TAO, the Service Configurator pattern (Jain and Schmidt, 1997) is applied. This pattern

enables an application to be configured dynamically at run time. Moreover, the pattern can reduce memory consumption of the ORB by dynamically linking only those needed strategies.

The use of design patterns in the development of TAO produces some expected and unexpected improvements in software reusability and maintainability. Compared with ad hoc code, the overall number of lines of code and the McCabe Complexity measure of certain operations are reduced significantly. However, some operations required increase of lines of code because of the use of the Wrapper Facade pattern to encapsulate low-level system calls, and more error checking. Design decisions are expressed with patterns, making the system more easily understood if one knows these patterns. A less complex and more easily understood system requires less effort to maintain. The use of the Strategy pattern, the Abstract Factory pattern, and the Service Configurator pattern make the system more extensible. New strategies can be introduced into the system more easily - even at run time.

2.7 Summary

This chapter has described design patterns as a new problem-solving discipline in software engineering. Design patterns capture successful design experience and document it in an easily understood form. Not only do patterns provide a solution, they provide rationale behind the system. Patterns offer many promising benefits, which have motivate many practitioners to document patterns and "mine" patterns from previous successful systems.

The description of patterns usually follows a format called the Alexandrian form. It is generally agreed that a pattern should have a name, problem and intent, context, forces

and tradeoffs, solutions, examples, resulting context and force resolution, rationale, and known uses.

The pattern community has adopted many means to improve discussion of patterns and pattern related topics. There are pattern forums and conferences where people write, review pattern papers, and share their experience using patterns. The World Wide Web is widely used to exchange ideas. People in the pattern community concentrate mainly on documenting patterns, applying design patterns to actual projects and documenting their experience in using patterns and, to a lesser extent, evaluating the usefulness of patterns.

In order to give readers a concrete idea of patterns, the MVC pattern was discussed briefly, and then a case study in applying patterns was presented. In the study, eight design patterns were applied in order to resolve various forces faced by the project. Compared with ad hoc systems, the complexity of the resulting system is reduced greatly. The maintainability and extensibility is increased significantly by applying design patterns in the system.

Chapter 3 Background: CLASSIC

CLASSIC (Borgida, Brachman, McGuinness, and Resnick, 1989) is a small description logic (DL) based language. *Description logic*, also called *term subsumption*, is a variable-free first order language (Borgida and Kudenko, 1994). Components of such systems are described as terms, and complex terms can be composed of other terms. DL-based systems are knowledge representation and reasoning systems which support a richer representation formalism than standard rule based systems (McGuinness and Borgida 1995). CLASSIC is chosen as the design model for this thesis because it is well defined both in syntax and semantics. This chapter will give a brief description of CLASSIC in order to give readers some background knowledge about the system that this thesis discusses.

Section 3.1 describes the components of CLASSIC. The computation in CLASSIC is based on these components. Section 3.2 describes the *description constructors* defined in CLASSIC. Descriptions in CLASSIC are built up with these *description constructors*. Section 3.3 discusses two actual implementations of CLASSIC.

3.1 Components of CLASSIC

CLASSIC defines description in a compositional manner. Complex knowledge structures are formed using a small set of semantic structures. The basic components of CLASSIC are *individuals*, *roles*, *concepts*, and *rules*. Individuals in a knowledge base are grouped into sets called concepts. Concepts are divided into primitive concepts and normal concepts. Roles are ordinary relations that relate individuals to each other. These components are described below:

- **Primitive concept** is the simplest kind of description that one can form in CLASSIC. The overt definition of primitive concepts is incomplete, and their definition includes something that is beyond the description associated with them (Resnick, Patel-Schneider, McGuinness, Weixelbaum, Abrahams, Borgida, and Brachman, 1996). Some primitive concepts can have further disjointness information associated with them. A disjoint primitive concept is just like a primitive concept, except that all primitive concepts belonging to the same disjoint grouping are disjoint to each other, and their composition is incoherent.

- **Concept** is a named description. Concepts are fully defined in the knowledge base by the description associated with them. Intuitively, concept denotes a collection of individuals.

- **Roles** are ordinary binary relations that relate individuals to each other.

- **Rules** consist of an antecedent and a consequent, which are both descriptions. When the antecedent concept applies to the state of an individual, the rule is "fired" and the consequent concept may also be asserted to apply to the individual.

- **Individuals** are specific instances of concepts. Each individual has a name and variable states. One can make three kinds of assertions about individuals (Borgida and Kudenko, 1994): *assert-member(i, A)* which asserts individual *i* is in the extension of description *A*; *assert-fill(i, r, v)* which establishes that individual *i* is related to individual *v* by role *r*; *asserted-close(i, r)* which asserts that all fillers of role *r* on individual *i* are now known.

Systems based on CLASSIC are composed of these components. Concepts and individuals are defined using concepts and description constructors, which will be discussed in the next section.

3.2 Description Constructors

CLASSIC represents information in terms of descriptions, which are built up from identifiers using *description constructors*. The following *description constructors* are defined in CLASSIC (Resnick et al, 1996):

The **and** constructor

This constructor forms the conjunction of some number of descriptions. For example, a VegetarianPerson may be defined as:

```
(and Vegetarian, Person)
```

This means that VegetarianPerson is someone who is both a Vegetarian and a Person.

The **oneOf** constructor

This constructor enumerates a set of individuals, which are the only possible instances of the description. For example,

```
(oneOf John, Mary, Susan)
```

defines three individuals `John`, `Mary`, and `Susan`. If a concept is defined by the description, the instance of the concept can only be one of the three individuals.

The **all** constructor

An **all** constructor, also called *value restriction*, specifies that all the fillers of a particular role must be individuals described by a particular description. For example, the instances of

```
(all food, meat)
```

must have all their fillers for *food* be instances of *meat*, for example, *beef*.

The **atLeast** constructor

An **atLeast** constructor specifies the minimum number of fillers allowed for a particular role. For example, a parent might be defined to have at least one child:

```
(atLeast 1 child)
```

The **atMost** constructor

An **atMost** constructor specifies the maximum number of fillers allowed for a particular role. For example, an orphan might be defined to have no parent:

```
(atMost 0 parent)
```

The **minimum** constructor

A **minimum** constructor specifies the minimum value of fillers allowed for a particular role. For example, an adult might be defined to be those whose minimum age is 18:

```
(minimum age 18)
```

The **maximum** constructor

A **maximum** constructor specifies the maximum value of fillers allowed for a particular role. For example, an adolescent might be defined to be those whose maximum age is 18:

```
(maximum age 18)
```

The **fills** constructor

A **fills** constructor specifies that a particular role is filled by the specified individuals. For example,

```
(fills sister, Mary, Sandra)
```

specifies that the *sister* role is filled with individuals *Mary* and *Sandra*.

The four constructors, **all**, **atLeast**, **atMost**, and **fills**, form special types of descriptions known as *role restrictions*, which restrict the fillers of a role. They restrict either the type of fillers (**all** constructor), or the number of fillers (**atLeast**, **atMost** constructors), or they specify some or all of the actual fillers (**fills** constructor).

To help readers better understand the syntax of CLASSIC, the following presents a relatively complex example. The example is taken from (Gaines, 1995) where it is represented in graphical format.

```
primitive (US$)
primitive (US$000)
primitive (senior employee)
primitive (senior employee's assistant)
primitive (division,
  (and (all classification,
            oneof(sales, marketing, accounting, manufacturing)),
       (all revenues, and(US$000, integer))))
primitive (employee,
  (and (all salary, and(US$, integer)),
       (minimum salary 15000),
```

```
        (maximum salary 160000),
        (all division, division)))
primitive (foreman,
  (and employee,
        (minimum salary 35000),
        (maximum salary 70000),
        (all division,
           (and (atLeast division 1),
                (atMost division 1),
                (all classification, manufacturing)))
concept (senior foreman,
  (and foreman,
        (minimum salary 45000),
        (all division, (minimum revenues 1000))
rule (senior employee,
   senior foreman,
   (and senior employee,
        (all assistant, senior employee's assistant),
        (atLeast 1 assistant)))
individual (Fred Smith,
  (and foreman,
        (fills salary, 50000),
        (fills division, body works),
        (fills assistant, Sam Jones))
individual (body works,
   (fills revenues, 3000))
```

The first several lines define primitive concepts "US$", "US$000", "senior employee", and "senior employee's assistant". Then, primitive concept "division" is defined whose classification is one of "sales", "marketing", "accounting", or "manufacturing", and whose revenue is integer and "US$000".

Beginning with the next line, an "employee" is defined as primitive concept whose salary is integer and "US$000", and minimum salary is 15000 and maximum salary is 160000, and role division satisfies concept "division". Next, a "foreman" is defined as primitive concept, which is an "employee" with a salary between 35000 to 70000, and who is in a manufacturing division.

Then, a "senior foreman" is defined as a concept which is a "foreman" with a minimum salary of 45000, and whose division must have a minimum revenue of 1000.

Then, a rule "senior employee" is defined whose premise is concept "senior foreman", and consequence is defined to be concept "senior employee" and has at least 1 assistant and all whose assistants are senior employee's assistants.

Finally, an individual "Fred Smith" is defined to be a "foreman" and whose salary is 50000 and who works in "body works" division and who has an assistant "Sam Jones". At last, another individual "body works" is defined to be something with revenue of 3000.

From the definition of individual "Fred Smith", the system can infer that "Fred Smith" is also a "senior foreman" because its definition also satisfies the definition of concept "senior foreman". Then, rule "senior foreman" will be fired on the individual and "Fred Smith" can be further asserted to be a "senior employee", and its "assistant" role filler "Sam Jones" can be asserted to be "senior employee's assistant".

3.3 Reasoning of CLASSIC

Reasoning with descriptions is based on a logic built around the subsumption relationship. Two major relations between descriptions may be computed:

- *Incoherence*: the composition of two descriptions is incoherent if they are logically contradictory.

- *Subsumption*: one description, *D1*, subsumes another description, *D2*, if *D1* is more general than *D2*. In other words, description *D2* logically implies description *D1*. Description logic based systems deduce and maintain subtype hierarchies based on the semantics of description definition, producing a partial order where more general concepts are parents of more specific ones. Subsumption is used to determine a concept's position in the hierarchy.

Based on their definition, individuals can be computed to decide whether they are subsumed by a concept. An individual, *I*, is subsumed by a concept, *C*, if *I* is described by *C*, i.e., *I* satisfies every description on *C*.

One algorithm used to compute the subsumption relationship between descriptions is the structural subsumption method (Borgida and McGuinness, 1996). The algorithm involves two steps: descriptions are first normalized, then are checked to see whether one concept is more general than another.

Normalization

The normalization of description performs two functions. First, it makes explicit all the implied facts, or description by constructing a normal form containing the most specific forms of the different kinds of descriptions included. The normal form contains information from several sources, including rule firing (see requirement 19 discussed in chapter 4), role propagation (see requirement 17 discussed in chapter 4), and inheritance. It will classify and combine the information in a number of ways. Secondly, it will check whether or not there are inconsistencies in the description.

Subsumption

Once descriptions are normalized, determining whether one concept, *C1*, subsumes another concept, *C2* is relatively straightforward. Description *C1* is said to subsume description *C2* if *C1* is more general than *C2*. In order for *C1* to subsume *C2*, for each description on *C1*, there must be an equivalent or more specific description on *C2*.

Descriptions can be conjoined. When two descriptions of the same kind conjoin, the result is a description of the same kind. If two descriptions are of different kinds, the result will be an **and** description. The following table shows how descriptions are conjoined.

Table 3.1 Conjoining of Two Descriptions

Description1	Description2	Conjoined Description
(and d1)	d2 of any kind	(and d1, d2)
(oneOf set1)	(oneOf set2)	(oneOf intersection(set1, set2))
(all r d1)	(all r d2)	(all r (and d1, d2))
(atLeast n1 r)	(atLeast n2 r)	(atLeast max(n1, n2) r)
(atMost n1 r)	(atMost n2 r)	(atMost min(n1, n2) r)
(minimum n1 r)	(minimum n2 r)	(minimum max(n1, n2) r)
(maximum n1 r)	(maximum n2 r)	(maximum min(n1, n2) r)
(fills r set1)	(fills r set2)	(fills r union(set1, set2))

For example, if there are three descriptions as follow:

```
d1 = (minimum salary 15000)
d2 = (minimum salary 30000)
d3 = (maximum age 40)
```

then, the result of d1 conjoining with d2 will be

```
(minimum salary 30000)
```

the result of d1 conjoining with d3 will be

```
(and (minimum salary 15000), (maximum age 40))
```

3.4 Systems Developed

During the past years, many DL-based systems have been developed. This section describes two systems that are based on CLASSIC. The first system is KRS (Gaines, 1993; Gaines, 1995) developed at the Knowledge Science Institute at the University of Calgary. Another system is NeoClassic (Patel-Schneider, Abrahams, Resnick, McGuinness, and Borgida, 1996) developed in AT&T laboratories. The AT&T labs also implemented LISP and C versions of CLASSIC knowledge representation systems. They do not seem to relate directly to this thesis; thus, they will not be discussed here.

3.4.1 KRS

KRS (Knowledge Representation Server) was modeled on CLASSIC. It was implemented as a class library in Think C. The KRS was intended to be designed as an open-architecture server which could provide basic capabilities and to which functionality might be added in a principled manner. The library should have well-defined interfaces for new classes to support additional data types.

The design of the KRS made several extensions to CLASSIC. These extensions, which include inverse relation between roles, negation, etc., are concerned with the functionality of the server. These extensions to CLASSIC are necessary to knowledge acquisition applications. Though the thesis will not implement these functionalities, these extensions should be able to be supported by creating new subclasses and incorporating them into the system.

The major part of the KRS development was to design appropriate data structures (as the author called them). The main data structures of KRS include the following:

concept records hold concept definitions,

individual records hold individual definitions and specifications of role fillers,

filler records hold sets of individuals that fill roles,

dictionaries map concept, role, rule, individual names to the index number of other structures,

extension records hold subsets of individuals,

data records hold subsets of external individuals of primitive data types, e.g., integers, reals, etc.,

inclusion records hold relations between role chains,

rule records keep track of exception relationships between rules,

inverse records keep track of inverse relations between roles,

subsumption records hold computed subsumption and incoherence relations.

3.4.2 NeoClassic

NeoClassic was a new C++ version of CLASSIC implementation done at AT&T after its LISP and C versions. NeoClassic provides three low-level interfaces: a character-based interface, a graphical user interface, and an application program interface.

Domain knowledge in NeoClassic is represented as description of concepts, description of individuals in terms of concepts and descriptions. The description of individuals includes relations between individuals (roles and role fillers). In addition, rules are supported in NeoClassic to assert more information about individuals.

NeoClassic makes heavy use of C++ features. The implementation is a collection of classes. The implementation consists mainly of the following classes: *Knowledgebase*, *Description*, *Construct*, *Role*, *Concept*, *Individual*, and *Rule*. The roles of these classes are obvious from their names. Description is further divided into *thing description*, *host description*, and *Classic description*. Host descriptions refer to primitive data types such as strings, integers, and floats. Classic descriptions refer to other objects defined in the model. Thing descriptions represent the union of host description and Classic description. Concept and individual are also divided into host and Classic categories. Users of the system can write *ad hoc* test functions in C++ to extend the capabilities of the system.

3.4.3 Discussion

The systems described in the preceding two sections were both full-fledged systems which have been used in many systems (Gaines, 1995; Wright et al, 1993; McGuinness and Wright, 1998). Both implementations put much emphasis on the functionality of the system to achieve more powerful, more efficient knowledge inference systems. The idea behind many of the core functions implemented in the thesis was drawn from these systems - though the implementation of this thesis lacks many of their features.

Because these systems were either implemented early, when design patterns were not well recognized, or people implementing these systems worked mainly on knowledge representation and reasoning research, there has been no paper published about design patterns concerning these systems. However, the author believes that if the design documents and source code of the systems could be studied, many design patterns could be identified.

3.5 Summary

This chapter has given a brief description of description logics and CLASSIC. The purpose of this chapter is to give a general background about CLASSIC so that readers can more easily understand the work presented in this thesis.

CLASSIC is a small description logic that is used for knowledge representation systems. The domain knowledge in such systems is represented as descriptions. CLASSIC is mainly composed of concepts, roles, rules, and individuals. Individuals in a knowledge base are grouped into sets called concepts; roles are ordinary relations that relate individuals to each other, and rules are used to assert that an individual satisfies additional descriptions if the individual satisfies the premise.

Descriptions in CLASSIC are built up from identifiers using *description constructors*. Eight *description constructors* were described in section 3.2. These *description constructors* are all implemented in the work of the thesis. Descriptions are compositional. Complex descriptions are composed from simple descriptions.

The main reasoning of descriptions is built around subsumption relations. The structural subsumption method first normalizes descriptions, then checks if one description is more general than the other to determine the subsumption relationship.

Two implementations of CLASSIC were presented. These implementations were full-fledged knowledge representation systems. The functionality to be described in chapter 4 is derived from these systems.

Chapter 4 Requirements Analysis

This chapter describes the objectives and requirements for the knowledge inference engine. The primary objective of this research work is to evaluate whether design patterns are applicable to artificial intelligence, particularly to the knowledge inference area. Serving as a test system, the final product needs not be a full-fledged knowledge inference engine. However, the system should implement all the main features necessary to support knowledge inference. The main focus of the work should be to design the system in a disciplined way, and to evaluate whether published design patterns are applicable to the problem domain. The implemented system should be able to serve as a kernel for knowledge inference, which could be easily extended to support more features and be made more powerful.

The requirements were first derived by studying several other kinds of knowledge inference engines - NEOCLASSIC (Patel-Schneider et al, 1996), KRS (Gaines 1993; Gaines 1995), and other communication systems and frameworks such as those described in (Schmidt 1995a; 1996a).

4.1 Objectives

The primary objective of this thesis is to evaluate the applicability of design patterns in the knowledge inference domain. The primary objective of the research can be reached through the following auxiliary objectives:

1. Studying software design patterns and developing an in-depth understanding of them;

2. Designing and implementing a knowledge inference engine;

3. Applying design patterns to the design and implementation of the system as appropriate;

4. Documenting the design patterns in the context of the system.

The design and implementation of the inference engine is modeled on CLASSIC (Borgida, Brachman, McGuinness, and Resnick 1989). The requirements that will be discussed in the following sections fall in two categories: those related to the knowledge inference engine functionality, and those related to supporting functionality or system extensibility.

4.2 Knowledge Inference Engine Requirements

The requirements discussed in this subsection are organized according to the different aspects of the system. They mainly address the functionality of the knowledge inference engine. These requirements are indispensable to the function of the knowledge inference engine.

The knowledge base is composed of primitive concepts, concepts, individuals, roles and rules. The following will give a more detailed description of the requirements for these elements.

4.2.1 Knowledge Base

The knowledge base is where the collection of defined concepts, individuals, roles and rules is stored. Whenever changes occur to an element, the knowledge base will compute

the consequences of the change and update the state of the knowledge base to reflect such changes.

The knowledge base maintains the truths about the concepts and individuals in it. Any operation that will cause an incoherent concept or individual should be rejected so that the knowledge base is always in a consistent state.

Requirement 1	The system must maintain the knowledge base in a consistent state.

The knowledge base will be changed dynamically by the users' update requests. These requests may be add more components or remove previously defined components. The components include concepts, individuals, and rules. The updating operations will cause the knowledge base to compute the impact of the changes and ensure that they will not cause the knowledge base to be inconsistent.

Requirement 2	The system shall support addition of new components to the knowledge base.

Requirement 3	The system shall support removing components from the knowledge base.

At any moment, the client should be able to retrieve component information from knowledge base. There are two kinds of information about a component: defined information and inferred information. Defined information refers to those descriptions that are used to define the component (concept, individual or rule). Inferred information is the descriptions that can be inferred based on defined information of a component.

The client should be able to retrieve defined information of a component as it is. This information shows the client how the component is defined.

Requirement 4	The system shall support retrieving defined information about components in the knowledge base.

The client should also be able to retrieve inferred information about a concept or an individual. Inferred information can be obtained through different mechanisms. For a concept, inferred information is obtained through *inheritance* (see requirement 8 and requirement 12). For an individual, three mechanisms - *inheritance*, *role propagation* (see requirement 17) and *rule firing* (see requirement 19) - exist to infer information on individuals.

Requirement 5	The system shall support retrieving inferred information about components in the knowledge base.

4.2.2 Concept

Concepts are named descriptions in a knowledge base. When a concept is created, it is first checked to ensure that the descriptions used to define the concept are coherent. Incoherent concepts cannot be added to knowledge base because they would make knowledge base inconsistent.

Requirement 6	The system shall support checking the coherence of concepts.

The method used to check whether or not a concept is coherent follows the algorithm described in (Borgida and Patel-Schneider 1994; Patel-Schneider et al, 1996).

Many data types are very primitive and are frequently used in a variety of situations. Such data types include integer, floating point number, string, etc. These data types are

too primitive to require the client to define them as concepts. The system should create them as built-in data types in order to improve the usability of the system. NeoCLASSIC (Patel-Schneider et al, 1996) treats primitive data types differently from other concepts, whereas primitive data types are defined as *host concepts*. The system to be designed will not make such distinctions. Primitive data types are created as concepts when the knowledge base is first created. These concepts are called built-in concepts.

Requirement 7	The system shall support several of the most often-used data types (including integer, float pointing number, and string) as built-in concepts.

A concept will inherit information from all of its parents and parents' parents. If one concept *C* is asserted to be child of another concept *P*, then concept *C* will also satisfy the definition of concept *P*. In other words, concept *C* will inherit all the information (defined and inferred information) of concept *P*.

Requirement 8	The system shall support concepts inheriting information from parents.

One of the fundamental functions of description logic based knowledge inference systems is to classify descriptions. The system must support this functionality. The classification of concepts is based on concept subsumption relationships. If the definition of one concept, *C1*, is more generalized than another concept, *C2*, then it is said that concept *C1* subsumes concept *C2*.

Requirement 9	The system shall support classification of concepts.

In order to reduce the amount of computation and to speed up the system, it is necessary to store parent-child relationships. The stored classification information will form a

directed acyclic graph. With the classification information, clients can perform many taxonomy retrieving operations without performing intensive computation. Thus, the retrieving operations can be performed very fast.

The concept taxonomy information that clients can retrieve includes getting direct parent concepts, all ancestor concepts, direct children concepts, or all descendants, of one concept.

Requirement 10	The system shall support retrieving concept classification information.

4.2.3 Individual

The requirements discussed in this subsection are very similar to the requirements described in the previous section. This is because of the inherent similarity between concept and individual. Concepts can be seen as abstract objects, while individuals are concrete objects that exemplify the properties of concepts.

Individuals are concrete instances of concepts. When individuals are created or updated, they must be checked for coherence. The method used to check whether an individual is coherent or not is similar to that used in NeoCLASSIC (Patel-Schneider et al, 1996).

Requirement 11	The system shall support checking of coherence of individuals.

Similar to requirement 8, individuals shall inherit information from all of their parents, and parents' parent concepts. In other words, if individual *I* is known to be an instance of concept *C*, then individual *I* will also satisfy the descriptions used to define concept *C*.

Requirement 12	The system shall support individuals inheriting information from

	their parents.

The individuals will be classified when they are created, or updated. The classification of individuals is similar to that of concepts. But individuals do not have children. The classification done to individuals is to recognize them as instances of concepts based on the subsumption relationship. If the definition of individual *I* satisfies the definition of concept *C*, then individual *I* is recognized as instance of concept *C*. Technically, an individual is classified as an instance of the most specific concept that subsumes it. However, if an individual is an instance of concept *C*, the individual is also an instance of concept *C*'s parent concepts.

Requirement 13	The system shall support classification of individuals.

Instances of primitive data types are not subject to classification, as doing so has no meaning. These instances are treated as instances of built-in concepts. Their state is encapsulated in the role fillers of a certain type.

Clients shall be able to retrieve the individual classification information. The client can retrieve such taxonomy information as the direct parents, or all parents, of an individual.

Requirement 14	The system shall support retrieving individual classification information.

After individuals are created, information can be added to or removed from them. Adding information to an individual means that the individual must satisfy the newly added description. Both operations will cause the individual to be checked for coherence and then reclassified. If the update operation should cause the individual to become incoherent, the operation will be rejected and the individual will remain unchanged.

Requirement 15	The system shall support addition of new information to individuals.

Requirement 16	The system shall support retracting information from individuals.

When an individual changes, the states of those individuals with it as a role filler should also change. Those individuals should then be recomputed for coherence and be reclassified. If an individual should become incoherent, the operation should be rejected.

4.2.4 Role

Roles represent binary relations between individuals. A role can also be an attribute. When an individual *I*, that has role filler for role *R*, is normalized, the value restriction for role *R* of individual *I* is given to those role fillers. This process is called *role propagation*. If the role fillers are individuals, the value restriction of role R will be asserted to apply to those individuals.

For example, in the following CLASSIC expressions,

```
CreateConcept(Vegetarian, (and Person, (all food Vegetable)))
CreateIndividual(carrot)
CreateIndividual(Sue, (and Vegetarian, (fills food carrot)))
```

when individual *Sue* is normalized, the system recognizes that *Sue* has a *food* filler, *carrot*, and that all *food* fillers must be *Vegetable*. Therefore, *Vegetable* is propagated to *carrot*, i.e., *carrot* is asserted to be *Vegetable*. If the definition of *carrot* contradicts *Vegetable*, then an error is generated and the operation is canceled.

Requirement 17	The system shall support role propagation on individuals.

Role propagation occurs on individuals that have a value restriction and role fillers on the same role. It does not occur on concepts that have value restriction and role fillers on the same role.

4.2.5 Rule

Rule is used to assert that more information can be applied to an individual if the individual satisfies the antecedent description of the rule. Before a rule is added to the knowledge base, it must be checked to ensure that it is coherent itself, and that it will not cause the knowledge base to become inconsistent.

A rule is said to be coherent if both its antecedent and consequent descriptions are coherent. When a rule is added to knowledge base, it will be fired on all individuals that satisfy the rule's antecedent description. If rule firing would cause any individual to become incoherent, the rule will not be added into the knowledge base.

Requirement 18	The system shall support checking of coherence of rules.

Rule firing can cause the state of an individual to change. If the definition of an individual *I* satisfies the antecedent description of a rule *R*, then individual *I* is asserted to satisfy the consequence description of rule *R*. Rule *R*'s consequence description is then added to individual *I*. This process is called *rule firing*.

Requirement 19	The system shall support rule firing on individuals.

Rule firing must ensure the coherence of the knowledge base. The process can happen in two kinds of situations: at the time a rule is created, or when an individual is created or changed. In both cases, the rule is fired on individuals satisfying the rule's antecedent description.

4.3 Other Requirements

Normally, the system will run on a different process from that of the client program. The engine will listen passively for clients' connecting, updating, or retrieving requests. When receiving clients' requests, the engine will process the requests one by one and return the results to clients. Many clients may work on the same task at the same time, this requires that the engine be able to handle requests from different clients concurrently.

Requirement 20	The system shall support multiple clients connecting to the engine concurrently.

To decrease the latency of a network connection, and to respond to clients' requests quickly, the engine should have some mechanism to initiate connections and demultiplex input events efficiently.

Primitive data types will be used often in various applications. Though the initial design has defined several such data types, as discussed in requirement 7, they may not be enough to satisfy all applications' requirements. Different applications may require different or more primitive data types. The system should be designed so that it can be easily extended to support a large set of primitive data types.

Requirement 21	The system shall to extendable to support other primitive data types easily.

The intent of the initial design of the system is not to implement a powerful and full-fledged knowledge inference engine. However, this does not mean that the system should be designed so that it cannot be extended, or is hard to extend in order to provide more inference power. Rather, the design of the system should take extensibility into account, and make future extension less painful. One major possible extension to the inference

engine is the addition of more description constructors. The system now consists of eight description constructors, as discussed in section 3.1. Other constructors, such as the `not` constructor discussed by Gaines (Gaines, 1991), should be easily supported.

Requirement 22	The system shall be easily extendable to add more description constructors.

Description logic based knowledge inference systems are built around terms. Child concepts will inherit descriptions of parent concepts. When knowledge bases become larger, each concept (or individual) will inherit many descriptions. These descriptions are either concepts or compositions of description constructors. If each concept (or individual) were to create a copy of its own descriptions when it inherits information from its parents, the same description would be duplicated many times in the system. This would consume a large amount of memory, which is not an acceptable solution. In order to avoid such behavior, the system should manage memory effectively so that one description is created only once and is shared by all other concepts and individuals.

Requirement 23	The system shall have an effective mechanism to manage memory.

In knowledge representation systems, name conflict is a common issue that needs to be addressed. Different systems have different ways of addressing this issue. In KRS (Gaines 1995), Gaines allows components in different categories to have same name. In KRS, primitive concepts and concepts fall in one category; individuals, roles, and rules each fall in one category respectively. For example, an individual may have the same name as a concept. However, components in the same category cannot have same name, i.e., it is impossible to create two individuals that both have name *Sam*. NeoCLASSIC (Patel-Schneider et al, 1996) adopted a similar approach to that of KRS.

However, in knowledge representation systems, both situations - components with same name but are in different categories, or components with same name residing in same category - may happen. Therefore, both situations should be supported by the system. The system should have some mechanism to handle name conflicts among components in different categories as well as in the same category. The mechanism should be flexible enough so that client can choose to enable or disable this feature.

Requirement 24	The system shall support the existence of multiple components with the same name.

4.4 Summary

This chapter has discussed the objectives and requirements of the knowledge inference engine. The primary objective is to evaluate the applicability of design patterns in knowledge inference engines by designing and implementing a knowledge inference engine.

The complete list of requirements is summarized in table 4.1. The next chapter will explain how these requirements are satisfied in the implementation.

Table 4.1 The requirements for the Knowledge Inference Engine

Requirement 1	The system must maintain the knowledge base in a consistent state.
Requirement 2	The system shall support addition of new components to the knowledge base.
Requirement 3	The system shall support removing components from the knowledge base.

Requirement 4	The system shall support retrieving defined information about components in the knowledge base.
Requirement 5	The system shall support retrieving inferred information about components in the knowledge base.
Requirement 6	The system shall support checking of the coherence of concepts.
Requirement 7	The system shall support several of the most often-used data types (integer, floating point number, and string) as built-in concepts.
Requirement 8	The system shall support concept inheriting information from parents.
Requirement 9	The system shall support classification of concepts.
Requirement 10	The system shall support retrieving concept classification information.
Requirement 11	The system shall support checking of coherence of individuals.
Requirement 12	The system shall support individuals inheriting information from their parents.
Requirement 13	The system shall support classification of individuals.
Requirement 14	The system shall support retrieving individual classification information.
Requirement 15	The system shall support addition of new information to individuals.

Requirement 16	The system shall support retracting information from individuals.
Requirement 17	The system shall support role propagation on individuals.
Requirement 18	The system shall support checking of coherence of rules.
Requirement 19	The system shall support rule firing on individuals.
Requirement 20	The system shall support multiple clients connecting to the engine concurrently.
Requirement 21	The system shall be extendable to support other primitive data types easily.
Requirement 22	The system shall be easily extendable to add more description constructors.
Requirement 23	The system shall have an effective mechanism to manage memory.
Requirement 24	The system shall support the existence of multiple components with the same name.

Chapter 5 Design and Implementation

The previous chapter presented the objectives and requirements for the knowledge inference engine. This chapter describes the design and implementation of the system. The description also includes references back to the requirements. The system was implemented on an IBM PC under Microsoft Windows NT 4.0. It was implemented using Borland C++ 5.02. The implementation is a collection of related C++ classes, each providing certain services and joined together to provide the desired functionality and properties. Section 5.1 provides an overview of the whole class hierarchy of the system. Section 5.2 discusses the kernel of the inference engine. It is divided into two parts: the first part describes major classes used in the kernel; the second part discusses how design patterns are applied to resolve different concerns in the design. The requirements discussed in section 4.2 and some of the requirements discussed in section 4.3 are addressed in this section. Section 5.3 discusses other supporting utilities in the system. These utilities mainly address how to efficiently manage memory, and how to manage the name space flexibly. Section 5.4 illustrates the actually running of the system using two sets of data.

5.1 Overview of the System

This section describes the class hierarchy of the knowledge inference engine. Figure 5.1 is a simplified diagram of the class hierarchy that shows only the inheritance relationships. The directed arcs in the diagram represent parent-child relations, with parent at the arrow side. The class hierarchy is partitioned into several parts, which will be discussed briefly.

Classes under `KSI_KRE` form the kernel of the knowledge inference engine. They provide core functionality for description logic based knowledge representation systems. Classes under `RoleFiller` are used to represent role fillers for individuals. Class `KREParser` is used to analyze the input stream and convert it into a format that the knowledge base can understand. Class `ConstructFac` is used to mange shared descriptions created in the system. Class `NameManager` manages component names and assigns a unique id to the component. Class `GraphNode` is used to store and manage classification information.

The following sections will describe these classes in more detail. The description will cover how these classes satisfy the requirements given in chapter 4, as well as the rationale behind such a design. Many of the design decisions are described based on design patterns and how design patterns fit into the design space.

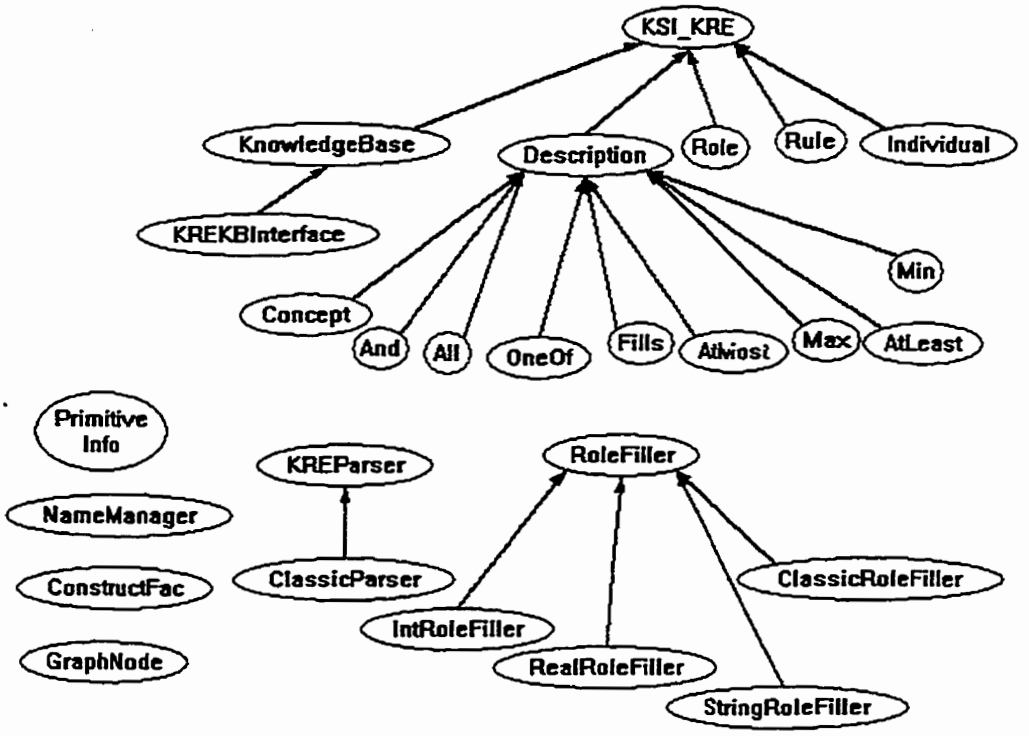


Figure 5.1 Simplified Class Hierarchy of Knowledge Inference Engine

5.2 The Kernel of the Engine

This section describes the most important part of the system: the kernel of the inference engine. The discussion is divided into two parts. The first part describes the major classes and the role they play in the kernel. The second part describes design patterns applied, why they are applied, and how they resolve different design issues. The description of this section is related to the requirements given in section 4.2 and some of the requirements given in section 4.3.

5.2.1 Participating Classes

Class `KSI_KRE` serves as the root for the kernel classes. These classes interact with each other to provide the desired functionality. The major classes in the kernel include `KnowledgeBase`, `Description`, `Role`, `Rule`, and `Individual`. Class `KnowledgeBase` is the center repository where other knowledge base objects are stored. Descriptions are the most fundamental building blocks in the system. They are modeled by class `Description`. Classes `Role` and `Rule` represent roles and rules in the system. They are relatively simple and will not be discussed in detail in this section. Class `Individual` represents individuals of the inference engine. The last class that will be discussed in this section is class `GraphNode`. Though it is not part of description logic, class `GraphNode` plays an important role in classification. The remainder of this section will discuss these classes in more detail.

5.2.1.1 KnowledgeBase Class

Class `KnowledgeBase` acts as the center repository that maintains a map (dictionary) for each type of named component. Named components in the knowledge base include *concept*, *individual*, *role*, and *rule*. Inside the knowledge base, these objects are not indexed by their name but by their id. When a named component is created, it is assigned a unique id by class `KRENameManager` (see section 5.3.2). The dictionary maps the component identifier to the actual objects. Class `KnowledgeBase` is responsible for managing these maps: whenever a component is created, the knowledge base will add an entry to the corresponding map

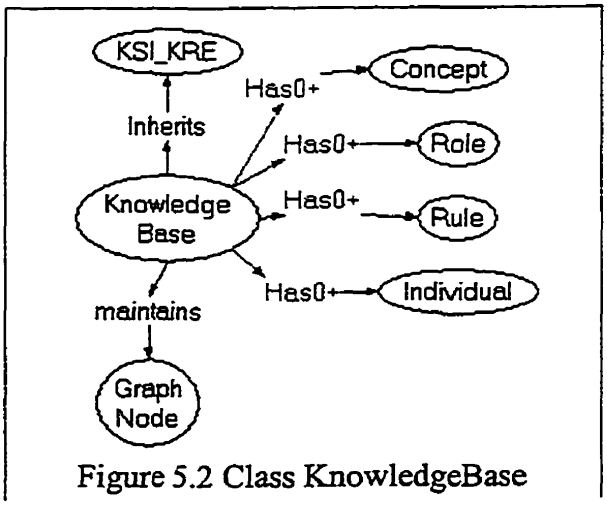


Figure 5.2 Class KnowledgeBase

representing the component; whenever a component is deleted, the knowledge base will remove the corresponding entry from the map. Class `KnowledgeBase` also performs the classification task - when a concept or individual is created, it will be classified. The `KnowledgeBase` class has an instance variable of type *GraphNode* (see section 5.2.1.4 for more detail) which will be used to store classification information. The class `GraphNode` provides certain methods for retrieving concept or individual taxonomy information, as per requirement 10 and requirement 14.

The class `KnowledgeBase` also has various methods used to create, delete, or update *concepts*, *individuals*, *roles*, or *rules*. The function of retrieving the defined and inferred information of a component is delegated to the actual component.

5.2.1.2 Description Class

Description is the most fundamental building block in description logic based knowledge representation systems. There are two types of descriptions: named descriptions and unnamed descriptions. Named descriptions include *concept* and *primitive concept*. Unnamed descriptions refer to descriptions constructed using *description constructors*. The *description constructors* supported by the system include all those described in section 3.2. Clients can only manipulate named descriptions. Unnamed descriptions can not be manipulated directly. Named descriptions may be defined in terms of named or unnamed descriptions. As shown in figure 5.1, class `Description` is the base class for both named and unnamed descriptions.

These classes provide the most important functionality needed to support knowledge inference. One of the most important functions is to decide whether one description subsumes another one. The algorithm used in the system to compute subsumption relations between descriptions is the *structural subsumption* method (see section 3.3).

The class `Description` has declared the interfaces for the key functions. The methods defined in the interface are described below.

The `normalize` method: performs the function of description normalization.

The `coherent` method: checks to see whether or not a description is coherent. It returns *true* if the description is coherent; otherwise, it returns *false*.

The `subsume` method: determines the subsumption relations. There are two versions of the `subsume` method - one to determine whether a description subsumes another description, another to determine whether a description subsumes an individual. The classification of concept is based on the first one, and the classification of individual the second.

The `conjoin` method: is used to conjoin two descriptions. If the two descriptions can be conjoined, it will return the conjoined description; otherwise it will return *NULL* description.

Subclasses of the `Description` class all implement the key methods listed above according to their specific semantics. In addition, each of the classes has defined for it some other housekeeping methods.

Descriptions in the system are identified by their ids. For named descriptions, each object is assigned a unique id by `KRENameManager` when it is created. However, for unnamed descriptions, objects of one type share one id. A unique id is assigned to the class instead of each object. This is because unnamed descriptions are shared in the system, and they cannot be manipulated directly by clients. The id is only used to identify the type of object. The use of id is based on two considerations: i) efficiency, it is more efficient to

compare two id than to compare two strings. ii) memory, an id consumes less memory than a string representation.

Besides the description id, each description also has a role id. As described in section 3.2, some descriptions constructed using description constructors are associated with roles. The role id of a description is used to identify the role associated with the description. For descriptions which have no role associated with them, including *concept*, *and*, and *fills*, a unique role identifier is assigned to each of the classes. The id and role id that are assigned to classes are managed by class `ConstNames`. `There are two methods for each class that needs both ids:`

The *xxxID* method returns the unique id for the class

The *xxxRoleID* method returns the unique role id for the class

where *xxx* is the class name. Not all description classes need both methods.

5.2.1.3 Concept Class

Class `Concept` is used to model both concept and primitive concept. It has an instance variable of type `PrimitiveInfo` that determines whether or not a concept is primitive. The design does not make primitive concept a subclass of class `Concept`. Instead, the designer uses composition to represent such relationships. Class `Concept` delegates all primitive information processing tasks to class `PrimitiveInfo`. The reasons for adopting such a design are: i) Primitive concept and concept behave in very similar ways, which are both named description. They can be used to define other named descriptions, or individuals

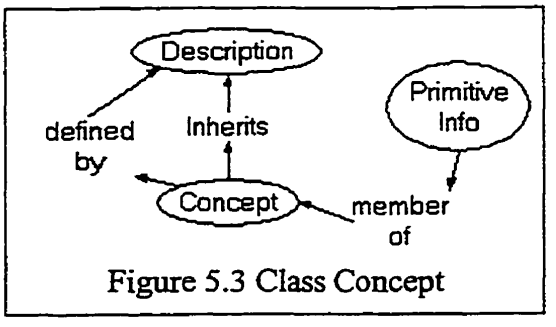


Figure 5.3 Class Concept

can be asserted to be instances of them. The processing of the two classes is almost the same, so there is no need to separate them. ii) The designer thinks that composition and delegation are more flexible than subclassing. With composition, the type of objects can be changed dynamically from concept to primitive concept, or vice versa. However, with subclassing, in order to change the type information one has to delete the original object and create a new object. Deleting a concept in the knowledge base may involve intensive computation: the use of composition reduces the need to delete a concept.

Concepts are defined in terms of descriptions. Class `Concept` maintains two definition references - one for defined information and one for normalized information. The instance variable `definedInfo` holds description information that is used to define the concept. When a concept is normalized, the normalized description is stored in the instance variable `normalizedInfo`. The attribute `definedInfo` should never be changed; any inferred information is added to `normalizedInfo`. These two references make retrieving concept definition information straightforward.

5.2.1.4 Individual Class

Class Individual models individual in CLASSIC. Individual represents a concrete instance of a concept. As discussed in requirement 7, there is no host concept, therefore there is no host individual either. Like concept, individual also has a `definedInfo` and a `normalizedInfo` instance variable, which refers to defined definition and the normalized information of the individual respectively. Unless an explicit operation requires a change to the definition of an individual, the `definedInfo`

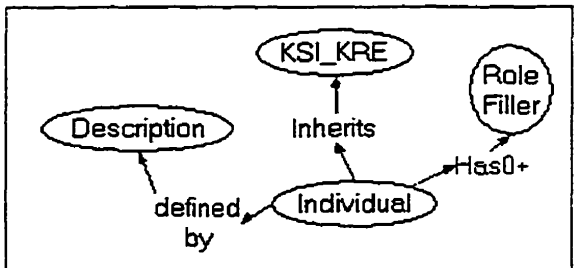


Figure 5.4 Class Individual

of an individual is never changed. Information obtained from inheritance, role propagation, and rule firing is only added to the `normalizedInfo` of the individual.

The key functions supported by the class include `normalize` and `coherent`. Method `normalize` takes the defined information of the individual and normalizes it. After the definition has been normalized, the system will perform the role propagation process (see requirement 17), if applicable. Method `coherent` checks to see if the individual is coherent or not. If the description used to define the individual is incoherent, or any role fillers of the individual are incoherent, the individual is incoherent.

The role fillers of class `Individual` are modeled using class `RoleFiller`. There are four types of role fillers in the current design: integer, floating pointer number, string, or normal individual. These types of role fillers are represented by subclasses of class `RoleFiller` (see figure 5.1). Based on role value restriction, the filler type can be determined. If a role filler is of a primitive data type, e.g., integer or string, a certain type of `RoleFiller` object, e.g., `IntRoleFiller` or `StringRoleFiller`, is created to represent the filler. The filler identifier is converted to a value of that type and is stored in the `RoleFiller` object. Otherwise, an object of class `ClassicRoleFiller` is created to store the individual filler.

In the example in section 3.2, individual ôFred Smithõ is defined as:

```
(and foreman,
     (fills salary, 50000),
     (fills division, body works),
     (fills assistant, Sam Jones))
```

The individual has three role fillers. The value restrictions for these role fillers can be obtained from the definition of concept `foreman`. The value restriction of role `salary` is

```
(and integer, US$)
```

From this, the system can determine that the filler for role `salary` is of primitive type integer. The system will then create an `IntRoleFiller` object, convert 50000 into an integer value and save the value to the role filler object.

The value restriction for role `division` is

```
(all classification, manufacturing)
```

The system checks and finds out that this is not a primitive data type. So, the system creates a `ClassicRoleFiller` object and ôbody worksõ is saved to the filler object as an individual. If ôbody worksõ has not yet been defined, the system will automatically create the individual.

The state of an individual may be updated by adding or removing information from it. When the state of an individual changes, the individual will inform the knowledge base of the change so that the system can re-classify the individual according to its new state. In addition, if the individual is a filler of some other individuals, the individual must also inform those individuals of the change so that those individuals can update their state accordingly.

5.2.1.5 GraphNode Class

Class `GraphNode` is the actual place where classification information is stored. The knowledge base will create one object of `GraphNode` for each concept. The `GraphNode` object records the classification information of the concept associated with it. The information includes direct parent concepts, child concepts, individuals subsumed by the concept, and rules whose antecedent is subsumed by the concept. The knowledge base delegates all requests to retrieve taxonomy information to this class.

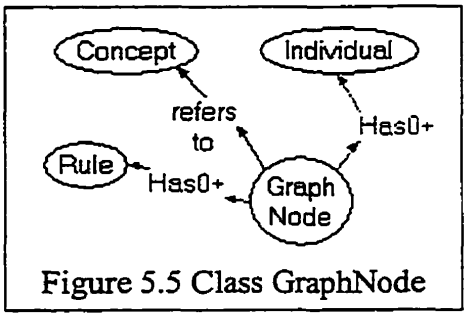


Figure 5.5 Class GraphNode

Classification information can also be stored in `Concept` and `Individual` classes. There are several advantages to managing classification information in the `GraphNode` class. First, because the function is centrally located in one class, it is easier to maintain the

system. Any changes to the classification function will affect only one class, and only this one class needs to be modified. Secondly, the classification function should not be the behavior of concepts or individuals. It should be one function of the knowledge base. If the function is put into the two classes, the interfaces of the classes will be contaminated and the classes become spaghetti classes containing many unrelated methods. Separating the function from them simplifies the classes, and makes each class play only one role in the system.

An alternative solution to this problem is to use the multiple inheritance feature of C++. A class can be defined to handle the classification information -- the concept and individual classes then both inherit from this class.

5.2.2 Design Patterns Applied in the Kernel

This section describes the design patterns that are applied in the design of the kernel of the knowledge inference engine. These design patterns are applied in order to achieve objective 2 in chapter 4. The design patterns also address many of the requirements discussed in section 4.3.

5.2.2.1 Interpreting CLASSIC

CLASSIC is a small description logic language that has well defined syntax and grammar (see Appendix A). Though the implemented system made some modifications to the grammar of the language (see section 2 of Appendix A), the language still has a well-defined grammar and syntax. Whenever there is a language to interpret, it is very natural to think of using the Interpreter pattern (Gamma, Helm, Johnson, and Vlissides, 1994, pp.243-255).

The intent of the Interpreter pattern is to represent the grammar of a language and interpret sentences in the language. The Interpreter pattern represents each grammar rule as a class. Symbols on the right-hand side of grammar rule are instance variables of these classes. The class structure of the Interpreter pattern is shown in figure 5.6. The `TerminalExpression` implements an *interpret* method associated with the terminal symbol in the grammar. The `NonterminalExpression` implements the *interpret* method for a nonterminal symbol in the grammar. Typically the *interpret* method of `NonterminalExpression` is implemented by calling the *interpret* methods of its subexpressions.

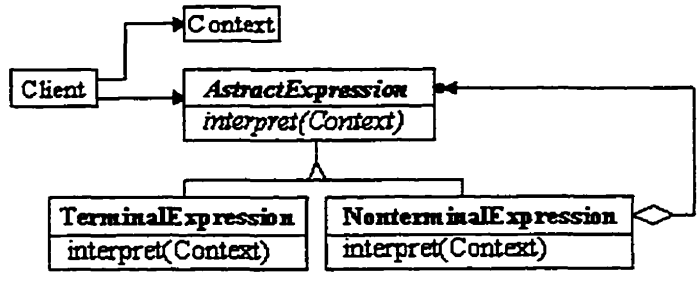


Figure 5.6 Class Structure of the Interpreter Pattern

The class structure of the Interpreter pattern begins with an `AbstractExpression`, an abstract class that is the base class for both `TerminalExpression` and `NonterminalExpression`. `TerminalExpression` and `NonterminalExpression` are concrete classes that represent the grammar rules of the language. They are child classes of `AbstractExpression`. All classes in the class structure have an *interpret* method to interpret the grammar rule appropriately. The *interpret* method takes Context as an argument. What the Context should contain depends totally on what the *interpret* method intends to do. For example, if the *interpret* method is supposed to evaluate expressions defined in the language, the Context should support looking up of the value of each variable. But if the *interpret* method needs to search a string that matches a pattern, the Context should contain the input stream and the current state of the *interpret* operation.

The client is the program that will use the pattern. The client builds (or is given) the sentence as an abstract syntax tree built with instances of `NonterminalExpression` and `TerminalExpression`. The client then initializes the context and invokes the *interpret* method. The *interpret* method uses context to store and access shared information of the Interpreter.

The Interpreter pattern is applied here to interpret the CLASSIC language. In the design, class `Description` forms the base class for the pattern. Class `Concept`, class `And`, and class `All` are `NonterminalExpressions`, which may be composed of other descriptions. Other classes in the structure are `TerminalExpressions`, which are the most basic building blocks of the language. As discussed in chapter 3, to interpret the language is to decide the subsumption relation between descriptions, i.e., method *subsume* in the design corresponds to the *interpret* method in the pattern. There is no Context in the design because the description classes have enough information to interpret themselves. The client in the design is class `KnowledgeBase`. Class `KnowledgeBase` builds the descriptions and performs the interpretation task. The interpretation done by the knowledge base is the classification of concepts or individuals based on the *subsume* method. The actual implementation class diagram is shown in figure 5.7.

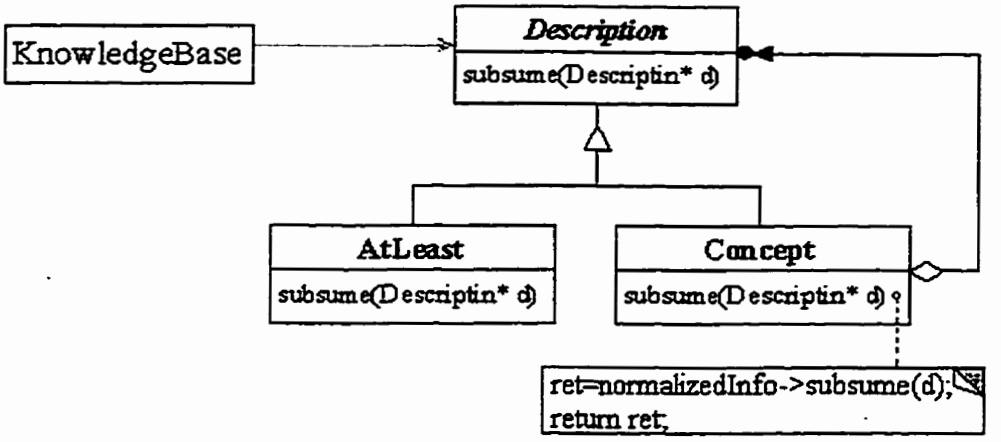


Figure 5.7 Class Structure of the Implementation of the Interpreter Pattern

To interpret (parse) a language, there are several choices. One can generate code using an automated parser generator such as *lex* or *yacc*. The code generated by such tools is usually more efficient than the one using the Interpreter pattern. However, such code is foreign to the system, and is usually more difficult to understand and thus, more difficult to maintain. The Interpreter pattern is suited for interpreting those languages that are not very complex, and where efficiency is not an important issue. This is just the case of the system discussed in this thesis - CLASSIC has a relatively simple grammar.

In addition, using the Interpreter pattern has other benefits: i) The system is simple and easy to understand, thus easy to maintain. The grammar rules of the language are represented as classes. All classes in the structure are similar, so the implementation is very simple. ii) The system is made easier to extend by using the Interpreter pattern. Section 5.2.2.2 will discuss how the system can be extended to support other description constructors.

5.2.2.2 Extending to Support More Description Constructors

By using the Interpreter pattern to represent the CLASSIC language grammar, it is easy to extend the grammar, as per requirement 22. In order to add a new description constructor to the system, the following two steps need to be performed: i) create a new subclass of class `Description`; ii) modify class `ConstNames` to assign a unique id, and role id if necessary, to the new constructor.

Class `Description` has defined the required methods as pure virtual (abstract) functions that must be implemented by all its subclasses. The methods that a new subclass must implement are shown in table 5.1.

Table 5.1 Methods required for Description Constructor

Method	**Description**
getId	This is a pure virtual method defined in class KSI_KRE. It returns the unique id of the description.
setId	This is a pure virtual method defined in class `KSI_KRE`. It sets the id of a description. For description constructors, the method does nothing because all objects of one description constructor share the same id.
getRole	Returns the role id of the description. For those descriptions related to a role, the role's id is returned. Otherwise, the unique role id of the class is returned. See section 5.2.1.2 for a more detailed description.
normalize	Normalizes the description. If the new *description constructor* is a terminal expression, the method simply returns the description itself; otherwise, it normalizes the description (as described in section 3.3) and returns the

	normalized description.
coherent	Checks if the description is coherent or not. Returns *true* if it is coherent; otherwise, it returns *false*.
conjoin	Conjoins two descriptions. Returns the conjoined description. Refer to table 3.1 for what the function actually does.
subsume	Determines if the description subsumes another description. The method returns *true* if it subsumes the argument description. Otherwise, it returns *false*.

The second step to adding a description constructor is to modify class `ConstNames` and add methods that give the class a unique id and unique role id, as appropriate (see section 5.2.1.2).

5.2.2.3 Support More Primitive Data Types

Primitive data types are supported as built-in concepts (see the discussion of requirement 7). The system maintains a table that records all currently supported primitive data types. When the knowledge base is being initialized, these concepts are created according to the table. Instances of these data types are represented as certain types of role fillers. In order to satisfy requirement 21, the Prototype pattern (Gamma et al, 1994, pp.117-126) is applied.

The Prototype pattern provides a way to create objects using prototypical instances. New objects of classes are created by copying the prototypical objects. The class structure of the Prototype pattern is shown in figure 5.8.

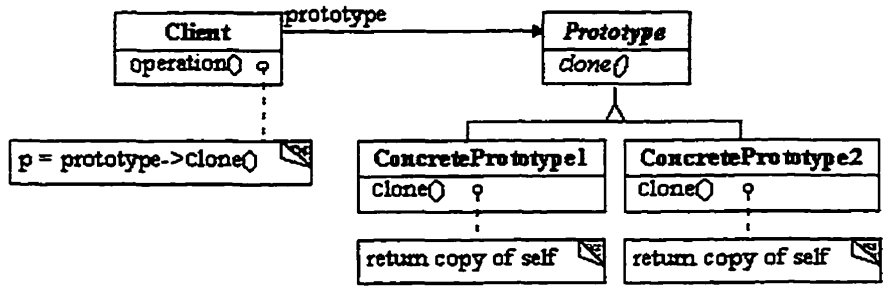


Figure 5.8 Class Structure of the Prototype Pattern

As can be seen in the above diagram, the pattern consists of class `Prototype`, which defines the interface of *clone* for cloning itself. Class `ConcretePrototype` implements the method *clone*. The client creates new objects by asking objects of concrete prototype classes to clone themselves. There are several advantages to using the Prototype pattern. First, concrete classes are hidden from client. The client is given a prototypical object without knowing what type of object it is. The ways that the client gets the prototypical object can vary greatly. By simply calling the *clone* method of the prototypical object, the client can get a new object of that type. Secondly, using the Prototype pattern enables addition or removal of products at run time. New products can be added by registering new prototype instances with the client. Existing products can be removed by unregistering them from the client. Thus, the system can be made more flexible.

The pattern is applied in the system to make adding primitive data types easier. The structure of the actual implementation of the pattern is shown in figure 5.9.

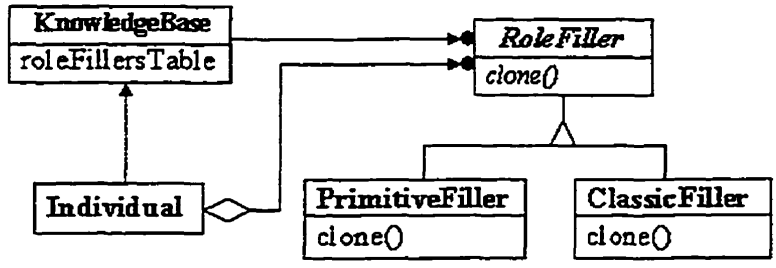


Figure 5.9 Class Structure of Actual Implementation

As shown in the above diagram, class `KnowledgeBase` maintains a table of currently registered role fillers. When the knowledge base is created, the table is initialized to hold supported role fillers for all primitive data types in the system. The primitive role fillers now supported include `IntRoleFiller`, `RealRoleFiller`, and `StringRoleFiller`. The table also maintains the built-in concepts associated with each primitive data type. When an individual needs a role filler, it asks the knowledge base to create one that satisfies certain constraints for it. The constraints here refer to the value restriction of the individual. The knowledge base looks up the table according to the constraints. The lookup process checks whether or not the constraints are subsumed by a built-in concept. If the constraints are subsumed by a built-in concept, the corresponding prototypical role filler is the one of correct type, and the knowledge base invokes the method *clone* to create a new object and returns it to the individual. Otherwise, the role filler is a CLASSIC individual, and `ClassicRoleFiller` is created.

To add a new primitive data type to the system, two changes need to be made.

1. Create a new concrete subclass of class RoleFiller. The new subclass should implement all the pure virtual methods defined in the RoleFiller class interface.

2. Modify the table that maintains currently supported primitive data types, adding the data type to be supported. The content of the table is a list of names, as character strings, for the primitive data types.

When these steps are finished, the program can be re-compiled and run; it will support the new data type.

5.2.2.4 Managing Individual Changes

The state of an individual may change as the knowledge base evolves. A change to an individual changes will affect all other individuals that have it as a role filler. These individuals in turn, will further affect other individuals. When individuals change, they need to be recomputed and classified as instances of certain concepts. In order to manage the change propagation in a systematic way, the Observer pattern (Gamma et al, 1994, pp.293-303) was applied to the design of the system.

The intent of the Observer pattern is to define dependency relations among objects so that when the state of one object changes, all objects that depend on it are notified and updated automatically. The pattern includes two classes: subject and observer. The class structure of the pattern is shown in figure 5.10.

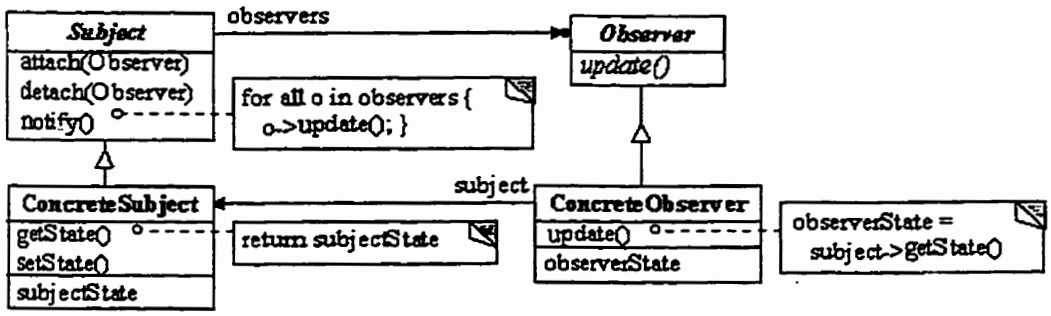


Figure 5.10 Class Structure of the Observer Pattern

One `Subject` object can be observed by many `Observer` objects. The subject maintains a list of its observers. It provides interfaces to add or remove observer objects from its observer list. When the state of a subject changes, it iterates all of its observers and invokes their *update* method. The class `ConcreteSubject` also provides an interface for an `Observer` object to get its state so that the observer can determine what has been changed and how it should act on the change. The `Observer` class provides an *update* method through which a `Subject` object can notify it of its changes. When an `Observer` object is created, it registers itself to the object of `Subject` that it wants to observe. After the registration, all changes are handled automatically.

In the inference engine, the subject and the observer are of the same class - class `Individual`. The actual class structure is shown in figure 5.11. If individual *I1* is a role filler of individual *I2*, then *I2* acts as an observer of *I1*. Whenever *I1* changes, it invokes the *update* method of *I2* so that *I2* can update itself automatically.

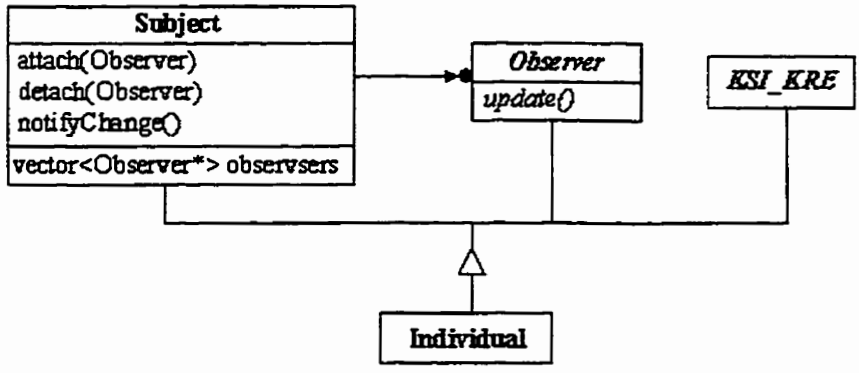


Figure 5.11 Class Structure of Actual Implementation

When processing an individual, the system analyzes the role fillers of the individual one by one. If a role filler is recognized as a CLASSIC role filler, i.e., the filler is an individual, then the individual registers itself as an observer of the filler individual.

Through the registration process, the observer/subject relation is established between individuals.

Two kinds of operations can result in the removal of the observer/subject relation between two individuals, *O* and *S* where *O* is observer and *S* is subject. The first kind of operation is to change the definition of individual *O* and remove the role that relates *O* to *S*. The second kind is to delete individual *O*. However, the deletion operation is not guaranteed to succeed. If individual *O* is at the same time the subject of other individuals, it can not be deleted. The same will happen if someone tries to delete *S*: because the system checks and finds that *S* still has *O* as its observer, the system will not delete individual *S*.

Through the use of the Observer pattern, the implementation of individual is made simpler. The design and implementation can be focused more on static aspects. Most of the design deals with only one individual object, and the interaction among individuals is handled by the pattern.

5.2.2.5 Simplifying the Knowledge Base Interface

The knowledge base is composed of many building blocks - *concepts*, *individuals*, *roles*, and *rules*. To use the system, one needs to be familiar with the interfaces of those building blocks. For some users, learning all the interfaces is overly burdensome. In order to make the system easier to use, the Facade pattern (Gamma et al, 1994, pp.185-193) was applied to make the interface simpler.

The Facade pattern is also known as the wrapper pattern. The main purpose of the pattern is to define a unified interface to a set of interfaces in a subsystem. The high-level interface defined by the pattern hides unnecessary complexity of the subsystem from

users, making the subsystem easier to use. The class structure of the Facade pattern is shown in figure 5.12.

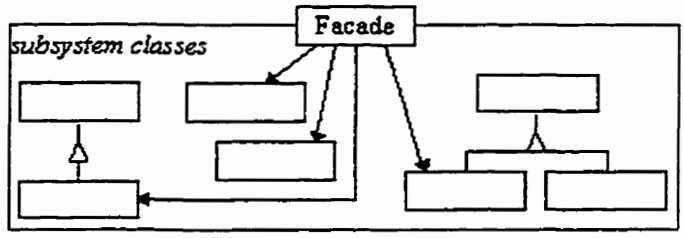


Figure 5.12 Class Structure of the Facade Pattern

In this pattern, there are two participants: the Facade, and classes of the subsystem. The Facade object knows the functionality of classes in the subsystems, and delegates client requests to appropriate objects of the subsystem classes. The subsystem classes implement the actual functionality. They handle the requests passed from the Facade object. However, they have no knowledge of the facade.

The knowledge inference engine contains many classes. These classes interact and communicate with each other. The interaction of these classes is complex. In order to simplify the use of the system, the Facade pattern was applied to hide the complexity and provide a high-level interface that is easier to use.

The facade class of the knowledge inference engine, class `KBInterface`, defines a set of interfaces that clients can use to interact with knowledge base without knowing much about the internal structure. The methods defined in the interface are shown in table 5.2.

Table 5.2 Simplified Interface of Knowledge Base

Method	**Description**
`create(int objType,` `string& initData,`	Used to create a component in the knowledge base based on the arguments. The type of component is

`int enforce=0)`	decided by argument *objType*: 0 û concept, 1 û primitive concept, 2 û role, 3 û rule, 4 û individual. The argument *initData* is the data used to define the component. The argument *enforce* decides what the system should do in the case of a name conflict. If enforce is set to no-zero, the system will create the component even there is name conflict. Otherwise, the system will return an error informing the client there is name conflict. The method returns 0 if the operation succeeds, otherwise a non-zero value representing the error that occurred is returned.
`update(IDType id, string& data)`	Used to update a component. The argument *id* identifies the component to be updated. The second argument, *data*, is the new definition for the component. Because the update of concepts is very complex and requires intensive computation to re-classify the knowledge base[1], the system supports the update of individuals only. The method returns 0 if the operation succeeds; otherwise it returns a non-zero value representing the type of error that occurred.

[1] To update a concept in the knowledge base, the system needs to delete the concept, and then replace it with a new concept. Both step will involve large amount of computation to classify other concepts, individuals, and rules.

`find(string& name, int type)`	Used to search the knowledge base and return all components found. The argument *name* is a character string representing the component name. The argument *type* has the same meaning as the argument *objType* in method `create` and represents the type of component to be searched. The method returns the set of components found in the knowledge base. If no component is found, it returns a null set.
`findFirst(string& name, int type)`	Used to search the knowledge base and return the first component found. The arguments of this method are the same as that of method *find*. The method returns the id of the component. If no component is found, 0 is returned.
`exist(IDType id)`	Used to check whether or not a certain component exists in the knowledge base. *IDType* is the type for ids. The method takes one argument, which is the id of the component, and checks to see if the component is defined in the knowledge base. It returns *true* if the component is defined; otherwise it returns *false*.

As one can see, the interface gives the client the power to use the knowledge base, but hides all the unnecessary details from client.

5.2.2.6 Accepting Different Input Formats

The engine uses CLASSIC as its internal data presentation. It can only understand the CLASSIC data format. In order to allow as many kinds of clients as possible to connect to it, the engine should not restrict the data format that a client uses. This requires the engine to provide a family of parsers which can translate the client data format into its internal data format. The parsers are all related and differ only in the external data format they can translate. In order to avoid using many if-then-else checks in the program (which is hard to extend and maintain), the Strategy pattern (Gamma et al, 1994, pp.315-323), as shown in figure 13, is applied in the design.

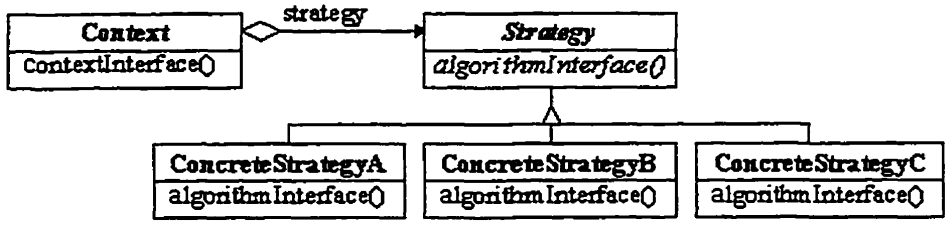


Figure 5.13 Class Structure of the Strategy Pattern

The Strategy pattern, as discussed in (Gamma, et al, 1994), is used mainly to define a family of algorithms. Each algorithm is encapsulated as a class, and all such classes are interchangeable. By applying the pattern, the algorithms may vary independently from the client that uses them because they provide a common interface to the client. Use of the pattern also eliminates the conditional statements from the implementation.

Figure 5.14 shows how the pattern was actually applied in the design.

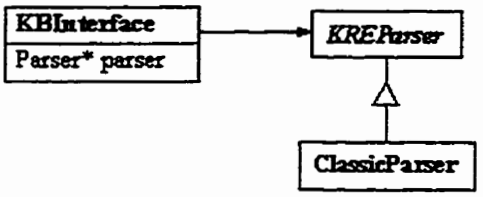


Figure 5.14 Class Structure of the Strategy Implementation

The class `KBInterface`, which is a facade class discussed in section 5.2.2.5, has a reference to the base class of the strategy pattern. The object of `KBInterface` takes an object of a concrete `KREParser` subclass as an argument when it is initialized. Later, the object uses the parser object to translate input data from the client to internal data format. The actual parser that will be used can be configured at run time. When a client program connects to the knowledge base, it tells the knowledge base what data format it uses, and the knowledge base can then set the parser to an appropriate one to communicate with the client. (Because of time constraints in the writing of this thesis, only one concrete `KREParser` class has so far been created.) As shown in figure 5.14, the class `ClassicParser` reads in data in CLASSIC format and translates the input into the format `KBInterface` requires.

To extend the engine to support other input data formats, the only thing that needs to be done is create a new concrete subclass of class `KREParser`. The KREParser defines all the interfaces that a subclass should implement. Thus, the design and implementation of the new class should be straightforward.

5.3 Supporting Functionality

This section describes two main supporting functions of the knowledge inference engine. The first one is managing memory - making all descriptions sharable - thus reducing memory consumption. The second supporting function that will be discussed is the name

manager. The name manager manages the name space of the knowledge base. All names used in the system are centrally managed by this class.

5.3.1 Description Constructor Factory

As discussed in requirement 23, memory management is a critical issue in the design of the engine. The engine should have an effective mechanism for managing memory, otherwise the engine will consume large amounts of memory. In order to reduce the amount of memory required, components in the knowledge base are shared; each component is created just one time and is shared by all others. If one description needs to use another description, it will get a reference to the other one instead of creating a new copy of that description.

To achieve the above goal, two patterns were used in the design of the name manager (class `ConstructFac`). The Singleton pattern (Gamma et al, 1994, pp.127-134) was used to ensure that only one instance of class `ConstructFac` is created. A variant of the Flyweight pattern (Gamma et al, 1994, pp.195-206) was used to achieve the sharing of description objects. The rest of this section will briefly discuss the two patterns, and then discuss how they were applied in the engine.

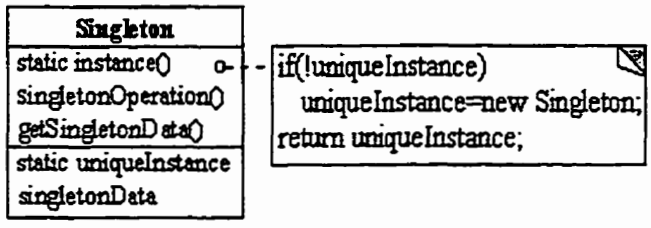


Figure 5.15 Class Structure of the Singleton Pattern

The Singleton pattern provides a way to ensure that there is no more than one instance of a certain class in the system at any time. The `Singleton` class is the only point where the instance of the class can be accessed. The pattern is shown in figure 5.15.

The `Singleton` class has a static method *instance* that enables a client to access the instance. By making the class itself responsible of managing its sole instance, it is easier to keep the system consistent.

The intent of the Flyweight pattern is to efficiently support a large number of fine-grained objects by sharing them. If a system has many objects, it will be very expensive because these objects will consume a lot of memory and may incur heavy run-time overhead. The Flyweight pattern introduces a way to share objects, so that one object can be used in multiple contexts. The pattern is shown in figure 5.16.

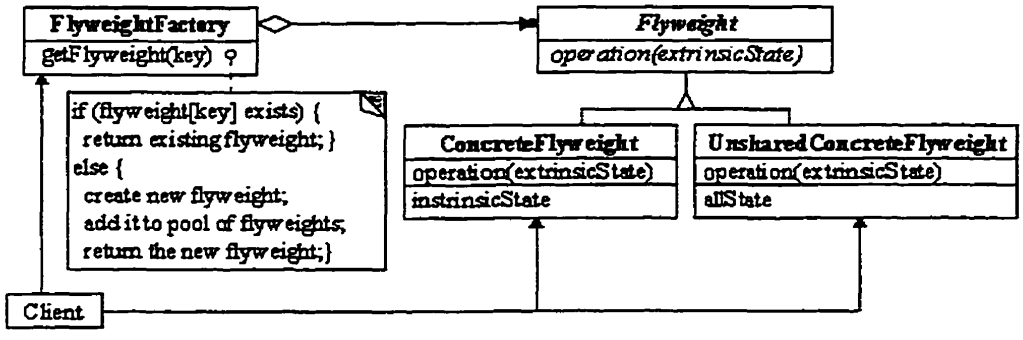


Figure 5.16 Class Structure of the Flyweight Pattern

The key concept of the Flyweight pattern is dividing the state of an object into two parts: *intrinsic* and *extrinsic* state. The intrinsic state is that information which does not depend on context. The extrinsic state, on the other hand, depends on the context. What is shared is the intrinsic information. Extrinsic state can not be shared. Class `FlyweightFactory` is a repository where sharable flyweight objects are stored. It provides an interface where the client can get a flyweight object by some kind of key.

The class `ConstructFac` in the engine is both a Singleton and a Flyweight Factory. The class structure is shown in figure 5.17. The `ConstructFac` class manages all the description constructors that are used in the inference engine. For each process of the engine there can be only one instance of the class. Thus, the `ConstructFac` is made a

Singleton class, which manages its sole instance. The client accesses the sole instance *_instance* through its member function *Instance*. The constructor of class `ConstructFac` is made private so the client has no way to create instances of the class except by using the provided interface.

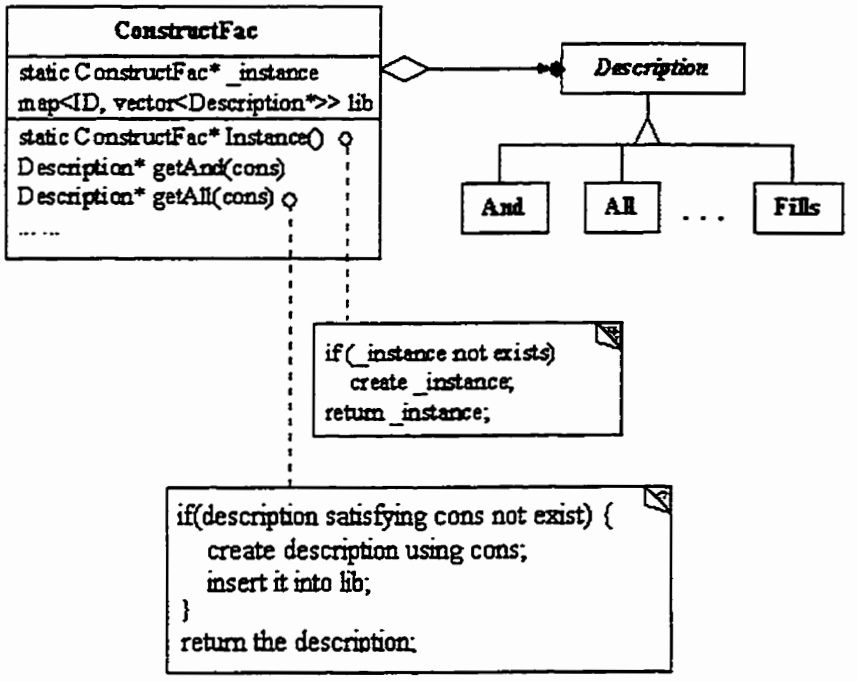


Figure 5.17 Class Structure of ConstructFac

The class has another instance variable *lib*, in which all objects of shared descriptions are stored. The *lib* is a dictionary whose key is the role identifier of descriptions. This means that descriptions related to the same role are stored under the same key. Descriptions under one key are stored in a list without ordering. There is an interface from which a client can get a reference of the desired description. If a client needs to get a description, he needs to invoke an appropriate method. For example, if the client needs to get description ô`AtLeast 1 position`õ, he invokes method *getAtLeast* with two arguments: id of role `position` and 1. If there exists a description that matches the arguments, a reference to the description is returned to the client. Otherwise, a new description is

created based on the parameters. The newly created description is put into the *lib* and a reference to it is returned to the client. All of the descriptions in the system are created in this way. Thus, at any time, there is only one copy of each kind of description, and all of them are shared by others.

The design is similar to the Library pattern discussed in (Kremer, 1997, p.137). The major difference between these two is that the Library pattern creates and returns a new copy when a client asks for an object, whereas the design in the inference engine returns only a reference to the object.

5.3.2 Name Space Management

As per requirement 24, the engine should have an effective way to manage the name space of the knowledge base. Similar to the `ConstructFac` discussed in the previous section, the name space manager should also be a Singleton. There can only be one instance of the manager in the system.

The design of the name manager is very similar to that of class `ConstructFac`. The class that manages name space in the system is called `KRENameManager`. The class is a singleton. It manages its sole instance and provides an interface for clients to access the instance. The class acts as a library for all the names used in the knowledge base. The client may query a component's name by its id, or query a component's id by its name. In order to provide efficient services to these queries, the name manager maintains two maps - one from component id to name, another from component name to id.

Before a component is created, the knowledge base inserts the name of the component into the name manager and gets a unique id for the component. If the name has been used by another component of the same type, the client can choose to change the component

name or create a component with that name. If the client chooses to change the component name, the change is easily handled by the system. Otherwise, the name manager will insert the name with a unique tag into its maps, and assign a unique id to the component. Because the system uses component id to uniquely identify a component, requirement 23 is relatively easy to satisfy.

5.4 Testing

After the system was developed, two sets of data were used to testing the correctness of the system. The first set of data was listed in section 3.2. The second set of data was obtained from AT&T NeoClassic tutorial Web site (Abrahams et al, 1996). Then, this section discusses the performance of the system in terms of time and space using recursive example.

5.4.1 The Testing Program

The testing program is a small program which uses interface provided by class `KBInterface`. The program is very simple and provides necessary functionality to perform the testing task. This section briefly discusses the function and the output of the program.

The client can use two commands to interact with the program. One command is used to read data into the knowledge base from a file; another one is used to get information about a component from knowledge base.

read command

This command is used to read data into the knowledge base from a file. The client can type in *read* command at the command line. The program will prompt the client to enter the file name to be read in and then proceeds reading the file.

get command

This command is used to query component information from the knowledge base. The client can type in *get* command at the command line. The program will prompt the client to enter the name and type of the component to be queried.

The program will print the query result to the standard output device. The information output by the program includes name of the component, and one or more of the following items.

toldDef - the description that is used to define the component.

normDef - the normalized description of toldDef of the component. The information in this part include inherited and inferred information.

direct parents - the concepts that are direct parents of the component. It reflects to direct parents of a concept or individual.

direct children - the concepts that are direct children of the concept. It reflects to direct children of a concept.

direct instance - the individuals that are directly subsumed by the concept. It reflects to direct instance of a concept.

direct rule - the rules associated with the concept.

fillers - the fillers of the individual.

5.4.2 Testing Data 1

The explanation of this set of data was given in section 3.2. This set of data creates an employee hierarchy for a company. Figure 5.18 is part of the input file.

```
createPrimitive (US$, ())
createPrimitive (US$000, ())
createPrimitive (senior employee, ())
createPrimitive (senior employee's assistant, ())
createRole (classification)
createRole (salary)
createRole (division)
createPrimitive(division, (and,(all, classification,
          (oneOf, sales, marketing, accounting, manufacturing)),
     (all, revenues, (and, US$000, Integer))))
. . .
```

Figure 5.18 Snippet of Test Data 1

Figure 5.19 shows how concepts are classified in the knowledge base. As one can see in the figure, concept `employee` has concept `Thing` as its parent and concept `foreman` as its child, whereas concept `foreman` has concept `employee` as its parent and concept `senior foreman` as its child. The text in bold are what a client typed in, and the text in normal font are system output.

Figure 5.20 shows the result of querying data about individual `body works`. Figure 5.21 shows the result of querying data about individual `fred smith`.

```
command: read
file name: test1.txt
Reading file ...
Command: get
Component name: employee
Type (0-concept, 1-primitive, 2-role 3-rule 4-individual): 1
employee: employee
NormDef ::= (AND (All salary (AND Integer US$ ) )
                 (minimum salary 15000) (maximum salary 160000)
                 (All division division)
Direct parents: Thing
Direct children: foreman
Direct instances:
Direct rules:

Command: get
Component name: foreman
Type (0-concept, 1-primitive, 2-role 3-rule 4-individual): 1
foreman: foreman
NormDef ::= (AND employee
                 (All salary (AND Integer US$ ) )
                 (minimum salary 35000) (maximum salary 70000)
                 (All division (AND division (All classification
                   (oneOf sales, marketing, accounting, manufacturing))
                   (fills classification manufacturing)
                   (All revenues (AND US$000 Integer ) ) ) )
                 (atLeast 1 division) (atMost 1 division))
Direct parents: employee
Direct children: senior foreman
Direct instances:
Direct rules:
```

Figure 5.19 Querying Result of Concepts Employee and Foreman

```
Command: get
Component name: body works
Type (0-concept, 1-primitive, 2-role 3-rule 4-individual): 4
body works: body works
ToldDef ::= (fills revenues 3000)
NormDef ::= (AND division (All classification
(one of sales, marketing, accounting, manufacturing))
(fills classification manufacturing)
(fills revenues 3000)
(All revenues (AND US$000 Integer ) ) )
Direct parent: division
Fillers:
Classification: manufacturing(classic individuals)
Revenues: 3000 (integers)
```

Figure 5.20 Querying Data of Individual "body works"

```
Command: get
Component name: fred smith
Type (0-concept, 1-primitive, 2-role 3-rule 4-individual): 4
fred smith: fred smith
ToldDef ::= (AND foreman (fills salary 50000)
(fills division body works)
(fills assistant sam jones))
NormDef ::= (AND foreman
Senior employee
(All salary (AND Integer US$ ) )
(minimum salary 35000) (maximum salary 70000)
(fills salary 50000)
(All division (AND division (All classification
(oneOf sales, marketing, accounting, manufacturing))
(fills classification manufacturing)
(All revenues (AND US$000 Integer ) ) ) )
(atLeast 1 division) (atMost 1 division)
(fills division body works) (fills assistant sam jones)
(All assistant senior employee's assistant)
(atLeast 1 assistant) )
Direct parent: senior employee, senior foreman
Fillers:
Salary: 50000 (integers)
Division: body works(classic individuals)
Assistant: sam jones(classic individuals)
```

Figure 5.21 Querying Data of Individual "fred smith"

5.4.3 Testing Data 2

The second set of data was obtained from AT&T NeoClassic Web site. The set of data defines a knowledge base about wine and food. The data can be obtained from (Abrahams, et al, 1996). Figure 5.22 shows part of the concept hierarchy of the knowledge base.

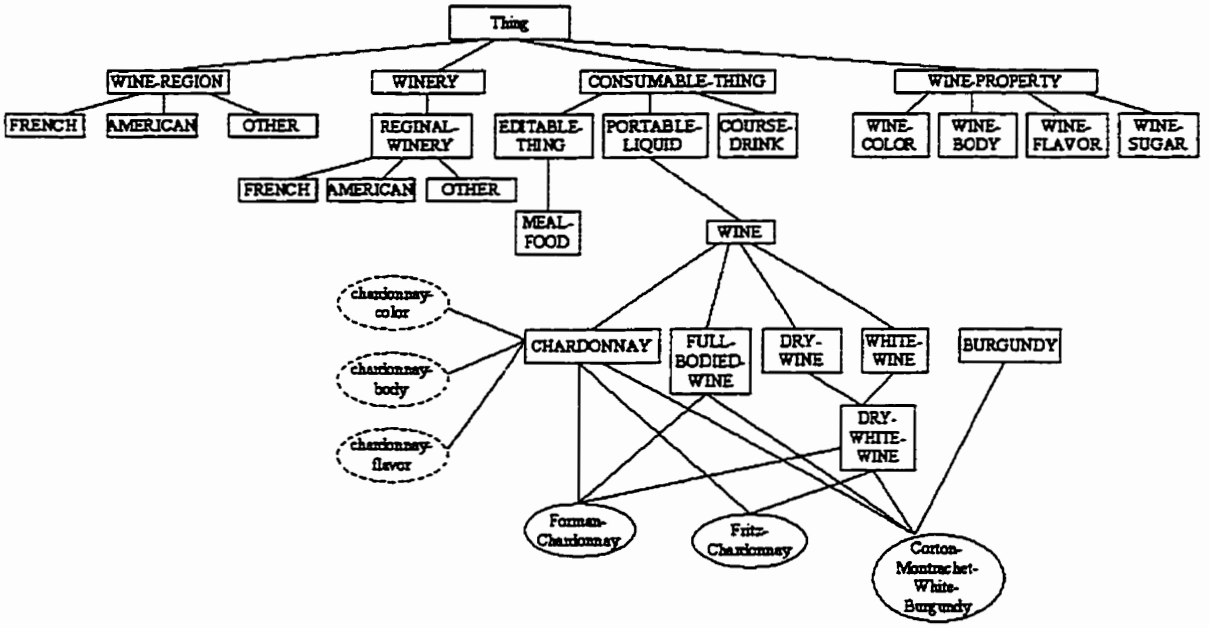


Figure 5.22 Concept Hierarchy of Wine and Food

In the above figure, a concept is represented as a box, an individual is represented as a ellipse, and a rule is represented as dash-lined ellipse. The component name is given by the text within a graph. The subsumption relationship is represented by connecting two concepts with a line, where the concept lying above subsumes the concept under it.

After the data was loaded to the system, several tests were performed on it. Figure 5.23 shows a query about concept CHARDONNAY. The definition for the concept is:

```
(and WINE (fills grape Chardonnay))
```

```
Command: read
File name: test2.txt
Reading file ...
Command: get
Component name: CHARDONNAY
Type (0-concept, 1-primitive, 2-role 3-rule 4-individual): 0
CHARDONNAY: CHARDONNAY
NormDef: (AND POTABLE-LIQUID
              (All color (AND WINE-PROPERTY (oneOf White, Rose, Red)))
              (atLeast 1 color)
              (All body (AND WINE-PROPERTY
                  (one of Light, Medium, Full)))
              (atLeast 1 body)
              (All flavor (AND WINE-PROPERTY
                  (oneOf Delicate, Moderate, Strong)))
              (atLeast 1 flavor)
              (All sugar (AND WINE-PROPERTY
                  (oneOf Sweet, Off-Dry, Dry)))
              (atLeast 1 sugar) (All grape EDIBLE-THING)
              (atLeast 1 grape) (fills grape Chardonnay))
Direct parents: WINE
Direct children:
Direct instances: Forman-Chardonnay, Fritz-Chardonnay,
                  Corton-Montrachet-White-Burgundy,
                  Puligny-Montrachet-White-Burgundy
Direct rules: chardonnay-color, chardonnay-body, chardonnay-flavor
```

Figure 5.23 Querying Data of Concept CHARDONNAY

As shown in figure 5.22, concept CHARDONNAY is classified to be child of concept WINE, and inherits all information from WINE. The system also recognizes individuals Forman-Chardonnay, Fritz-Chardonnay, etc. to be its direct instances.

Then, a query of individual Forman-Chardonnay was performed. The result of the query is shown in figure 5.24.

The definition for individual Forman-Chardonnay is as follows:

```
(and CHARDONNAY (fills body Full) (fills flavor Moderate)
                (fills sugar Dry) (fills maker Forman))
```

```
Command: get
Component name: Forman-Chardonnay
Type (0-concept, 1-primitive, 2-role 3-rule 4-individual): 4
Forman-Chardonnay: Forman-Chardonnay
toldDef ::= (AND CHARDONNAY (fills body Full) (fills flavor Moderate)
                            (fills sugar Dry)  (fills maker Forman) )
normDef ::= (AND POTABLE-LIQUID
                 (All color (AND WINE-PROPERTY
                     (one of White, Rose, Red) ) )
                 (atLeast 1 color)  (fills color White)
                 (All body (AND WINE-PROPERTY (oneOf Medium, Full) ) )
                 (atLeast 1 body)  (fills body Full)
                 (All flavor (AND WINE-PROPERTY
                     (oneOf Moderate, Strong)  ) )
                 (atLeast 1 flavor)  (fills flavor Moderate)
                 (All sugar (AND WINE-PROPERTY
                     (oneOf Sweet, Off-Dry, Dry)  ) )
                 (atLeast 1 sugar)  (fills sugar Dry)
                 (All grape(AND GRAPE EDIBLE-THING ) )
                 (atLeast 1 grape) (fills grape Chardonnay)
                 (All appropriate-drink
                    (AND (fills color White, Red)
                         (fills body Light, Medium, Full)
                         (fills flavor Delicate, Moderate, Strong)
                         (All flavor (oneOf Moderate, Strong) )
                         (fills sugar Sweet, Off-Dry, Dry)  ) )
                 (atLeast 1 region)  (All region WINE-REGION )
                 (fills maker Forman)
                 (atLeast 1 maker)  (All maker WINERY ) )
Direct parents: FULL-BODIED-WINE, DRY-WHITE-WINE, CHARDONNAY
Fillers:
       color: White(classic individuals)
       body: Full(classic individuals)
       flavor: Moderate(classic individuals)
       sugar: Dry(classic individuals)
       grape: Chardonnay(classic individuals)
       maker: Forman(classic individuals)
```

Figure 5.24 Querying Result of Individual Forman-Chardonnay

The individual is first classified as an instance of concept CHARDONNAY. Concept CHARDONNAY has three associated rules: chardonnay-color, chardonnay-body, and chardonnay-flavor. The definitions of these rules are as follows:

```
createRule (chardonnay-color, CHARDONNAY, (fills, color, White))
createRule (chardonnay-body,
        CHARDONNAY, (all, body, (oneOf, Full, Medium)))
createRule (chardonnay-flavor,
        CHARDONNAY, (all, flavor, (oneOf, Strong, Moderate)))
```

Because individual Forman-Chardonnay is recognized as an instance of the concept, the system fires these rules on the individual. The additional information is then added to the individual, and the individual informs the system to reclassify it. Because the filler for role body is Full, which satisfies the definition of concept FULL-BODIED-WINE, now the individual is classified to be also an instance of concept FULL-BODIED-WINE. The filler for role color is White and filler for role sugar is Dry, so the individual is also classified to be an instance of concept DRY-WHITE-WINE, as shown in figure 5.24.

The definition of concepts FULL-BODIED-WINE and DRY-WHITE-WINE are as follows:

```
createConcept (FULL-BODIED-WINE, (and, WINE, (fills, body, Full)))
createConcept (WHITE-WINE, (and, WINE, (fills, color, White)))
createConcept (DRY-WINE, (and, WINE, (fills, sugar, Dry)))
createConcept (DRY-WHITE-WINE, (and, DRY-WINE, WHITE-WINE))
```

5.4.4 Performance Test

This section tests the performance of the system in terms of time and space. Nebel (1990) has given a constructive example of a simple set of concept definitions where computation of subsumption is intractable because of the growth of unnamed descriptions. The recursive example offers a worst-case example on term subsumption logics. Figure 5.25 shows an example set of concept definitions for the testing. The set begins with the definition of a primitive C10 and goes through a sequence of concepts down to C0 defined in terms of preceding concepts.

```
createPrimitive (C10, ())
createConcept (C9,  (and, C10, (all, r1, C10), (all, r2, C10)))
createConcept (C8,  (and, C10, (all, r1, C9), (all, r2, C9)))
createConcept (C7,  (and, C10, (all, r1, C8), (all, r2, C8)))
createConcept (C6,  (and, C10, (all, r1, C7), (all, r2, C7)))
createConcept (C5,  (and, C10, (all, r1, C6), (all, r2, C6)))
createConcept (C4,  (and, C10, (all, r1, C5), (all, r2, (and, C5, C10))))
createConcept (C3,  (and, C10, (all, r1, C4), (all, r2, (and, C4, C8))))
createConcept (C2,  (and, C10, (all, r1, C3), (all, r2, (and, C3, C6))))
createConcept (C1,  (and, C10, (all, r1, C2), (all, r2, (and, C2, C4))))
createConcept (C0,  (and, C10, (all, r1, C1), (all, r2, (and, C1, C2))))
```

Figure 5.25 Nebel's Constructive Example with 10 Concepts

Table 5.3 System Performance

Number	Time (seconds)	Space (bytes)
0	0.00	24
1	0.01	90
2	0.01	303
3	0.01	573
4	0.01	966
5	0.01	1686
6	0.01	2349
7	0.02	4149
8	0.04	5262
9	0.06	9582
10	0.08	11505
11	0.16	21585
12	0.32	25038
13	0.65	48078
14	1.19	54501
15	2.99	106341
16	5.12	118614
17	10.42	233814
18	18.88	257697
19	44.69	511137
20	79.87	558150
21	210.60	1111110
22	352.63	1204293

Table 5.26 shows the time in seconds and the memory consumption in bytes for Nobel's examples 0 through 22 with the system running on a Pentium 266 PC under Microsoft Windows NT 4.0. The time and space metrics are useful vaules in estimating how the system can cope with realistic problems.

5.5 Summary

This chapter has provided an overview of the design and implementation of the knowledge inference engine. The most important part of the knowledge inference engine is the kernel of the engine, which provides all of the core functionality. Section 5.2 describes the major classes that the kernel is composed of. In the design and the implementation of the kernel, several design patterns are applied to achieve different goals:

- the Interpreter pattern interprets the CLASSIC language, and makes the design and implementation simpler. In addition, use of the pattern makes it easier to extend or modify the grammar of the CLASSIC language, such as extending CLASSIC by adding more description constructors.

- the Prototype pattern creates new objects by using prototypical objects. By applying this pattern, the engine can be extended to support other primitive data types easily.

- the Observer pattern allows observers be informed and updated automatically when a subject changes. This pattern is applied to manage the change process of individuals.

- the Facade pattern makes it easier to use the knowledge inference engine by providing a simple interface that hides much of the complexity from the client. The client interacts solely with the interface without knowing any parts of its subsystems.

- the Strategy pattern encapsulates the parsing algorithms into classes. Each of these classes is exchangeable. Thus, the knowledge base can be configured to support different formats of client input data dynamically.

Section 5.3 describes other supporting functionality in the knowledge inference engine. It covers mainly memory, and the name space management facility. Two patterns are applied in the design of these functionalities:

- the Singleton pattern ensures that no more than one instance of a class is created in the system. There can be only one instance of the description constructor factory and name manager in the system. These two classes are made singleton.

- the Flyweight pattern enables objects within the system to be shared. There may be many descriptions in the system. If these descriptions can be shared, memory is used more efficiently. The Flyweight pattern was applied to manage descriptions in the system so that each description is created one time and is then shared by others.

Section 5.4 illustrated the actually running of the system using two set of data and gave some performance data of the system using Nebel's constructive example.

This chapter serves as the design documentation for the knowledge inference engine. Though it has not covered all aspects of the design, it does cover the most important parts

of the system. The requirements given in chapter 4 are also addressed in the description of the design.

Chapter 6 Conclusion

This chapter evaluates this research work and describes two possible directions for future work related to the research. The evaluation addresses the objectives described in chapter 1 and chapter 4, and how these objectives are satisfied by the research work. Based on the discussion of objectives, the chapter also draws the conclusion that design patterns are applicable to the knowledge inference domain.

In addition, this chapter gives two possible directions for future work related to the research work. First, existing knowledge inference systems should be studied and checked for the existence of design patterns in those systems. Secondly, since the research was undertaken with the intention of developing a completed knowledge inference system, much work can be done to extend the system. The extension work can in turn, evaluate the quality of the current design.

6.1 Addressing the Objectives

The primary objective of this research work was described in chapter 1:

"to evaluate the applicability of design patterns in the knowledge inference domain."

In order to achieve the primary objective, four auxiliary objectives must be achieved. These four auxiliary objectives are:

1. Studying software design patterns and getting an in depth understanding of them;

2. Designing and implementing a knowledge inference engine;

3. Applying design patterns to the design and implementation of the system as appropriate;

4. Documenting the design patterns in the context of the system.

The auxiliary objectives are concerned with i) studying and the application of design patterns; ii) designing and implementation of a knowledge inference engine. If the four auxiliary objectives are accomplished, the conclusion that design patterns are applicable to the domain can be drawn. Otherwise, the conclusion will be that design patterns are not applicable to the knowledge inference domain.

Chapter 4 discussed the requirements for the system. Chapter 5 discussed the design and implementation of the system based on the requirements analysis. The descriptions in these two chapters covered the major functionality of the implemented knowledge inference system. The major functionality of the system includes:

- creating components to represent the domain knowledge. Components that can be created in the knowledge base include primitive concepts, concepts, individuals, roles, and rules.

- updating the definition of components. Usually the updating operation works on individuals, i.e., changing the state of individuals.

- removing information from the knowledge base.

- checking the consistence of the knowledge base. If all components in the knowledge base are coherent, then the knowledge base is consistent. The knowledge base is always kept in a consistent state, which means that incoherent components are not added to the knowledge base.

- classifying components in the knowledge base. The classification process is based on the subsumption relationships between components. This is the essential function that a description logic based knowledge representation system should support.

The above description clearly shows that the second auxiliary objective - to design and implement a knowledge inference engine - is accomplished. The implemented system possesses the basic functionality to support knowledge inference, though it is not as powerful as some systems that are already implemented, such as the ones discussed in section 3.3.

Section 5.2 and section 5.3 discussed the design and implementation. These sections discussed the major classes that provide the functionality of knowledge inference; they also discussed extensibility and flexibility issues. In addition, if design patterns were applied to the design, they were discussed under the context of the system. As one can see, several design patterns were applied in the system. These design patterns either simplify the design and implementation of the system, or make the system more easily extensible. For example, the Interpreter pattern simplifies interpreting the CLASSIC language, and the Observer pattern simplifies the change management of individuals. The design patterns identified in the system include:

- the Interpreter pattern, which is used to represent the CLASSIC language and interpret components of the language. The pattern makes the extension of language easier. New *description constructors* can be added without affecting other parts of the system.

- the Prototype pattern, which is used to represent role fillers. The application of the pattern makes it easier to plug in and support new data types.

- the Observer pattern, which is used to manage changes to individuals. Changes to the state of one individual may affect other individuals in the system. The pattern establishes dependency relationships between individuals, so that when the state of an observed individual changes, other individuals will be informed of the change and will be updated automatically.

- the Facade pattern, which is used to simplify the interface of the knowledge base. The knowledge base is composed of many components and they interact with each other. The interaction may be complex and hard to understand for those who do not understand those components. The pattern encapsulates the complexity of the interactions and provides a very simple interface for others to use.

- the Strategy pattern, which is used to encapsulate the algorithms of how input streams are interpreted. In order to allow as many kinds of clients as possible to connect to it, the system should not restrict the data format that a client uses to communicate with the engine. A parser is needed to translate client data formats to the format that the engine can understand. The pattern encapsulates each kind of parser as a class and makes them interchangeable. New client data formats can be supported by plugging-in a new parser.

- the Singleton pattern, which is used to ensure that no more that one instance of a class is created. The singleton class in responsible for managing its instance, thus avoiding the drawback of using global variables. In the implemented system, descriptions are shared across the system, i.e., there is only one copy of each description and it is shared by other components in the system. The descriptions are stored in a class that manages all of the instances created. There can be only one such manager, and it is made a singleton. In addition, the name space

manager, which manages the string names of components in the system, is also a singleton.

- the Flyweight pattern, which is used to share descriptions within the system. When a system uses a large number of fine-grained objects, it will be very costly if each use of the object requires a new copy of the object. Making these objects sharable in different contexts greatly reduces resource consumption. The pattern is used to make descriptions sharable. The use of this pattern also makes memory management more efficient and less error prone.

The preceding discussion shows that the first auxiliary objective (studying design patterns) is fulfilled. The third (applying design patterns to the system) and fourth (documenting the patterns) auxiliary objectives are fulfilled by the discussions in section 5.2 and section 5.3. Because the four auxiliary objectives are achieved, the primary objective is also achieved. Seven patterns have been applied to the design and implementation of the system, so the conclusion that design patterns are applicable to knowledge inference systems can also be drawn.

6.2 Future Work

This section will discuss several directions in which the future work could proceed. The future directions concern two different aspects of this research work. The first direction is the study of other knowledge inference systems and trying to ômineö patterns in those systems. The other direction concerns with the implemented system û the extension of the system and studying the actual consequences of the application of design patterns.

6.2.1 Design Patterns in Other Systems

The approach taken in this research work was to first study design patterns, and then try to apply design patterns to the design and implementation of a knowledge inference engine. Whether design patterns are applicable to the domain depends on whether or not any design patterns could be applied to the system. Another approach to the evaluation is to do reverse engineering, or "pattern mining". The idea of "pattern mining" is to study existing systems in-depth, then identify and document design patterns in the systems.

"Pattern mining" is an important activity conducted in the pattern community. Many of the design patterns currently available were found this way. By studying existing systems, it is possible to find many domain specific patterns. These patterns are valuable assets to the domain, which may be buried in experts' mind or otherwise lost in source code.

There are many existing systems that may be considered for "pattern mining". Such systems may include KRS, NeoClassic (discussed in chapter 3), Loom (MacGregor and Brill, 1992), etc. These systems were all implemented in Objected Oriented languages, which makes them valuable subjects for study as the purpose of this research is to study design patterns, especially Objected Oriented software design patterns in knowledge inference systems. However, this is not to say that other systems may not be studied.

6.2.2 Extending the System

As discussed before, the main focus of this research was not to develop a powerful knowledge inference system. Thus, the system developed to date supports only basic knowledge inference functionality. This leaves much space for future extension, which is another future direction of work - to extend the system to include other functions.

There are two types of extension: one extension is to improve the power of the system, and another is to extend it to interact with other programs.

6.2.2.1 Improving the Representation Power of the System

Complex descriptions in the system are built from simple descriptions using description constructors. One way to improve the power of the system is to add more description constructors to the system. With more description constructors, the system will have richer representation power. The possible description constructors may include the *not* constructor that negates a description, and the *some* constructor that specifies partial constraints. Another way to enhance the system is to add more data types to the system. Since the system treats primitive data types as built-in concepts, more data types will give users more representation power.

The system had implemented several data types to represent integer, floating point number, and character string. A user may need to use other data types to represent domain knowledge. The system may be enhanced by adding other data types, such as date, or other types that may be required by specific applications. The more types of data in the system, the more representation power the system should have.

However, the more description constructors and data types the system supports, the more complex the system will become. This is especially true for description constructors. Adding more description constructors will increase the complexity of the system, and more intensive computation will be required. Thus, when extending the system, one should balance representation power and system complexity so that the representation of the system is rich enough, while at the same time not requiring too much computation.

6.2.2.2 Interacting with Other Programs

The system is designed as a server application which accepts a client request, does certain work on behalf of the client and returns results to the client. Obviously, another direction for extending the server is to extend the interface of the system, enabling more kinds of clients to interact with it.

To the time this thesis was written, the system had only a text-based interface; client programs communicate with the server through text streams, and results are returned as text streams. The text-based interface is simple. The syntax of the text stream is the same as the grammar of CLASSIC described in Appendix. For example, to create a concept *adult* which is defined as a *person* and whose age is greater than 18, one can send the following text stream to the system:

```
create_concept(adult, (and, person, (minimum age, 18)))
```

where *person* is a previously defined concept in the system.

Though the text-based interface is very easy, some argue that visual presentation of knowledge is more readily understood (Nosek and Roth, 1990). Extending the system to add a visual front-end should greatly improve the usability of the system. Right now, the author is working on a task to extend the system so that it can interact with Constraint Graphs (Kremer, 1997), which is a framework for concept mapping languages. The work will be reported elsewhere when the work is finished.

The future work outlined in this section has two implications for the research work. First, the implemented system can be seen as a test that evaluates whether design patterns actually bring benefits to system design. Current work has shown that the application of design patterns in the system does simplify extension work. The extension of the system

to interact with Constraint Graph needs only another Parser class which understands the data format of the Constraint Graph. Secondly, through the extension, the system may be used in other applications.

6.3 Summary

This thesis has described research work that evaluates whether or not design patterns are applicable to knowledge inference systems. Although design patterns have been recognized by the software industry for several years, there are no reports regarding design patterns or their application in knowledge inference systems. What is the reason for this phenomenon? Is it because design patterns are not applicable to the domain?

The author thinks that design patterns are applicable to the domain. There are several approaches to verify the proposition û one can study existing systems and interview domain experts, or one can try to apply design patterns to the design and implementation of such a system. The author chose the latter approach û design and implement one such system and check whether or not design patterns are applicable.

The thesis has described design patterns, their components, their value, and their applications. A pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. Design patterns provide a structural and easily understood form for documenting and sharing successful experience among developers. They help improve communication among developers by providing a common vocabulary that has a higher level abstraction. Design patterns have been applied in a wide variety of systems, such as telecommunication, concurrent programs, telephony, MIS, GUI, etc.

The requirements analysis for the implemented knowledge inference system was given in chapter 4. In addition to the system design and implementation, chapter 5 described the design patterns applied in the system. These design patterns make the system more flexible and extensible.

The final chapter has described how the research has met its original objectives of evaluating the applicability of design patterns in knowledge inference systems. The research shows that design patterns are applicable to this particular domain. The chapter also gives two possible directions for future work:

- "pattern mining" – studying existing knowledge inference systems, identifying and documenting design patterns in those systems,
- extending the system so as to give it more representation power, or the ability to interact with other programs.

Appendix: The CLASSIC Grammar

A.1 Original Grammar of CLASSIC

This section is the original grammar defined in CLASSIC. The grammar is excerpted from (Resnick, Patel-Schneider, McGuinness, Weixelbaum, Abrahams, Borgida, and Brachman 1996).

```
Description          ::= ThingDescription |
                         ClassicDescription |
                         HostDescription |
                         IncoherentDescription
ThingDescription     ::= Thing |
                         ( and )
ClassicDescription   ::= ClassicThing |
                         ClassicConcept |
                         ( and ClassicDescription+) |
                         ( oneOf ClassicIndividual+) |
                         ( atLeast PositiveInteger  Role) |
                         ( atMost NonNegativeInteger  Role) |
                         ( fills Role ClassicIndividual+)
                         ( fills Role HostIndividual+) |
                         ( all Role Description) |
                         ( testC ClassicTestGenerate Parameter*)
HostDescription      ::= HostThing |
                         Number |
                         Integer |
                         Float |
                         String |
                         HostConcept |
                         ( and HostDescription+) |
                         ( oneOf HostIndividual+) |
                         ( minimum Number) |
                         ( maximum Number) |
                         ( testH HostTestGenerate Parameter*)
```

```
IncoherentDescription ::= (one-of)

Role                  ::= Symbol
ClassicConcept        ::= Symbol
HostConcept           ::= Symbol
Rule                  ::= Symbol
ClassicIndividual     ::= Symbol
HostIndividual        ::= " string" |
                          int |
                          float
ClassicTestDetail     ::= Symbol
HostTestDetail        ::= Symbol
Number                ::= int |
                          real
Parameter             ::= NeoObject
```

A.2 Modified Grammar Used in Project

```
Description            ::= Concept |
                           DescriptionConstructor
Concept                ::= Symbol
DescriptionConstructor ::= ( and Description+) |
                           ( oneOf Individual+) |
                           ( atLeast PositiveInteger Role) |
                           ( atMost NonNegativeInteger Role) |
                           ( minimum Role Number) |
                           ( maximum Role Number) |
                           ( fills Role Individual+) |
                           ( all Role Description)
Role                   ::= Symbol
Rule                   ::= Symbol
Individual                 : Symbol |
                           : " string" |
                           = Integer |
                             Float
Symbol                 ::= String
string                 ::= { char }*
```

```
Number                  ::= Int |
                            float
PositiveInteger         ::= [ 1..9 ]{ 0..9 }*
NonNegativeInteger      ::= { 0..9 }*
```

References

Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. New York, Springer, 1996.

Abrahams, M.K., McGuinness, D.L., Patel-Schneider, P.F., Resnick, L.A., Thomason, R.H., and Conati, C. (1996). NeoClassic Knowledge Representation System Tutorial, AT&T, http://www.research.att.com/sw/tools/classic/papers/NeoTut/NeoTut.html, 1996.

Alexander, C. (1977). *A pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

Alexander, C. (1979). *The Timeless Way of Building*, Oxford University Press, 1979.

Apple Computer Inc. (1989). *Macintosh Programmers Workshop Pascal 3.0 Reference*, Cupertino, California, 1989.

Appleton, B. (1997). *Patterns and Software: Essential Concepts and Terminology*, http://www.enteract.com/~bradapp/docs/patterns-intro.html, April 1997.

Beck, K. Coplien, J. O., Crocker, R., Dominick, L., Meszaros, G., Paulisch, F., and Vlissides, J. (1996). Industrial Experience with Design Patterns, *Proceedings of ICSE '96*, Berlin, pages 103-114, March 1996.

Borgida, A. (1992). Towards the Systematic Development of Description Logic Reasoners: CLASP Reconstructed, *KR-92*, 1992.

Borgida, A. and Brachman, R. (1992). Customizable Classification Inference in the ProtoDL Description Management System, *Conference on Information and Knowledge Management*, Baltimore, November 1992.

Borgida, A. (1995). Description Logics in Data Management, *IEEE TKDE*, October 1995.

Borgida, A., Brachman, R. J., McGuinness, D. L., and Resnick, L. A. (1989). CLASSIC: A Structural Data Model for Objects, *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1989.

Borgida, A.and Brachman, R. J. (1993). Loading Data into Description Reasoners, *Proceedings 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC., May 1993

Borgida, A. and Patel-Schneider, P. F. (1994). A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic, Journal of Artificial Intelligence Research, Vol. 1, 1994.

Borgida, A. and Kudenko, D. (1994). Modular Implementation of Individual Reasoning in PROTODL - the Extensible Describption Logic Management System, *Technical Report, lcsr-tr-237*, Department of Computer Science, Rutgers University, New Brunswick, December 1994.

Borgida, A. and Mcguinness, D. L. (1996). Asking Queries about Frames, *KR-96*, Boston, Mass., 1996.

Brachman, R.J., Selfridge, P.G., Terveen, L.G., Altman, B., Borgida, A., Halper, F., Kirk, T., Lazar, A., McGuinness, D.L., and Resnick, L.A. (1993). Integrated Support for Data Archaeology, *International Journal of Intelligent and Cooperative Information Systems* , 2:159-185, 1993.

Brown, K. (1996). Using Patterns in Order Management Systems: A Design Patterns Experience Report, *Object Magazine*, January 1996.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons*, New York, New York, 1996.

Coplien, J. O. (1994). Software Design Patterns: Common Questions and Answers, *Proceedings of Object Expo New York*, New York, SIGS Publications, June 1994.

Coplien, J. O. (1996a). The Human Side of Patterns, *C++ Report*, 8(1), January 1996.

Coplien, J. O. (1996b). Software Patterns, *SIGS Books*, New York, New York, 1996.

Coplien, J. O. (1997a). Idioms and Patterns as Architectural Literature, *IEEE Software Special Issue on Objects, Patterns, and Architectures*, 14(1), January 1997.

Coplien, J. O. (1997b). *A Pattern Definition*, http://st-www.cs.uiuc.edu/users/patterns/definition.html, March 1997.

Coplien, J. O., and Schmidt, D.C., Eds (1995). Pattern Languages of Program Design, *Reading*, Massachusetts, Addison-Wesley, 1995.

Duell, M. (1996). Experience in Applying Design Patterns to Decouple Object Interactions on the Intelligent Peripheral Prototype, *OOPSLA*, Addendum, 1996.

Gabriel, R. P. (1997). Developing Patterns Studies in Architecture Point the Way to Understanding and Improving Software Development, *Info World*, Vol. 19, Issue 5, February 1997.

Gaines, B. R. (1991). An Interactive Visual Language for Term Subsumption Language, *IJCAI'91: Proceedings of 12th International Joint Conference on Artificial Intelligence*, San Mateo, California, August 1991.

Gaines, B. R. (1993). A Class Library Implementation of a Principled Open Architecture Knowledge Representation Server With Plug-in Data Types, *IJCAI'93: Proceedings of 13th International Joint Conference on Artificial Intelligence*, San Mateo, California, 1993.

Gaines, B. R. (1995). Class Library Implementation of an Open Architecture Knowledge Support System, *International Journal of Human-Computer Studies*, 41(1-2), 1995.

Gamma, E. (1991). *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools*, PhD Dissertation, Institute of Information, University of Zurich, 1991.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1994.

Grady, B. (1994). Object-Oriented Design with Applications, 2nd Edition, CA: *Benjamin/Cumming*, 1994.

Jain, P., and Schmidt, D. C. (1997). Service Configurator: A Pattern for Dynamic Configuration of Service, *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

Krasner, G. E. and Pope, S. T. (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, SIGS Publications, New York, New York, 1(3), August/September 1988.

Kremer, R (1997). Constraint Graphs: A Concept Map Meta-Language, *PhD Dissertation*, Department of Computer Science, University of Calgary, June, 1997.

Lavender, R. G., and Schmidt, D. C. (1995). Active Object – An Object Behavioral Pattern for Concurrent Programming, *Proceedings of the Second Pattern Languages of Programs conference*, Monticello, Illinois, September, 1995.

Lea, D. (1997a). Christopher Alexander: An Introduction for Object-Oriented Designers, *http://gee.cs.oswego.edu/dl/ca/ca/ca.html*, Computer Science Department, State University of New York at Oswego, March 1997.

Lea, D. (1997b). Patterns-Discussion FAQ, *http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html*, Computer Science Department, State University of New York at Oswego, April 1997.

MacGregor, R.M. and Brill, D. (1992). Recognition Algorithms for the Loom Classifier, *Proceedings of the Tenth National Conference on Artificial Intelligence*, (AAAI 92), pp. 774-779, 1992. http://www.isi.edu/isd/LOOM/LOOM-HOME.html

McGuinness, D. L. and Borgida, A. (1995). Explaining Subsumption in Description Logics, *IJCAI'95: Proceedings of 14th International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

McGuinness, D. L. and Isbell, C. (1995). Description Logic in Practice: A CLASSIC Application, *IJCAI'95: Preceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

McGuinness, D.L. and Wright, J.R. (1998). An Industrial Strength Description Logic-based Configurator Platform, to appear in B. Faltings and G. Freuder, editors, *IEEE Expert Special Issue on Configuration*, 1998.

McClure, C. (1997). Reuse Engineering: Adding Reuse to the Software Development Process, *Prentice-Hall*, 1997.

Nebel, B. (1990). Reasoning and Revision in Hybrid Representation Systems. Berlin: *Springer-Verlag*, 1990.

Nosek, J.T., and Roth, I. (1990). A Comparison of Formal Knowledge Representation Schemes as Communication Tools, Predicate Logic vs. Semantic Network, *International Journal of Man-Machine Studies*, 33: 227-239, 1990.

Patel-Schneider, P. F., Abrahams, M., Resnick, L. A., McGuinness, D. L., and Borgida, A. (1996). *NeoClassic Reference Manual: Version 1.0*, Artificial Intelligence Principles Research Department, AT&T Labs Research, 1996.

Prechelt, L. (1997). An experiment on the usefulness of design patterns: Detailed description and evaluation, *Technical Report* 9/1997, Fakultat fur Informatik, Universitat Karlsruhe, Germany, June 1997. ftp.ira.uka.de.

Prechelt, L., Unger, B., Philippsen, M., and Tichy, W. (1997). Two Controlled Experiments Assessing the Usefulness of Design Pattern Information During Program Maintenance, submission to Empirical Software Engineering, December 1997.

Pree, W. (1995). *Design Patterns for Object-Oriented Software Development*, Reading, Addision-Wesley, Reading, Mass., 1995.

Resnick, L. A., Patel-Schneider, P. F., McGuinness, D. L., Weixelbaum, E., Abrahams, M. K., Borgida, A., and Brachman, R. J. (1996). *NeoClassic User's Guide: Version 0.7*, Artificial Intelligence Principles Research Department, AT&T Labs Research, 1996

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

Salingaros, N. A. (1997). *Some Notes On Christopher Alexander*, http://www.math.utsa.edu/sphere/salingar/Chris.text.html, April 1997.

Schmidt, D. C. (1994). Reactor - An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching, *Proceedings of the First Pattern Languages of Programs Conference*, Monticello, Illinois, August, 1994.

Schmidt, D. C. and Stephenson, P. (1995). Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms, *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

Schmidt, D.C. (1995a). Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software, *Communications of the ACM*, Special Issue on Object-Oriented Experiences, Vol.38, October 1995.

Schmidt, D.C. (1995b). An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit, *Technical Report*, WUCS-95-31, Department of Computer Science, Washington University, St. Louis, MO, 1995.

Schmidt, D.C. (1995c). Object-Oriented Components for High-speed Network Programming, *the Proceedings of the 1st Conference on Object-Oriented Technologies and Systems*, USENIX, Monterey, June 1995.

Schmidt, D. C., Johnson, R. E., and Fayad, M. (1996). Software Patterns, *Communications of the ACM*, Special Issue on Patterns and Pattern Languages, Vol. 39, No. 10, October 1996.

Schmidt, D. C. (1996a). A Family of Reusable Design Patterns for Application-level Gateways, *Theory and Practice of Object Systems*, special issue on Patterns and Pattern Languages, Wiley and Sons, Vol 2, December 1996

Schmidt, D. C. (1996b). A Family of Design Patterns For Flexibly Configuring Network Services in Distributed Systems, *Proceedings of the International Conference on Configurable Distributed Systems*, Annapolis, Maryland, May 1996.

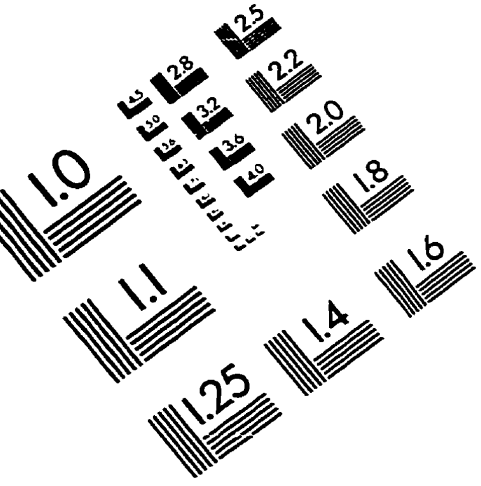
Schmidt, D. C. (1996c). Acceptor and Connector: Design Patterns for Initializing Communication Services, *European Pattern Language of Programs conference*, Kloster Irsee, Germany, July *1996.*

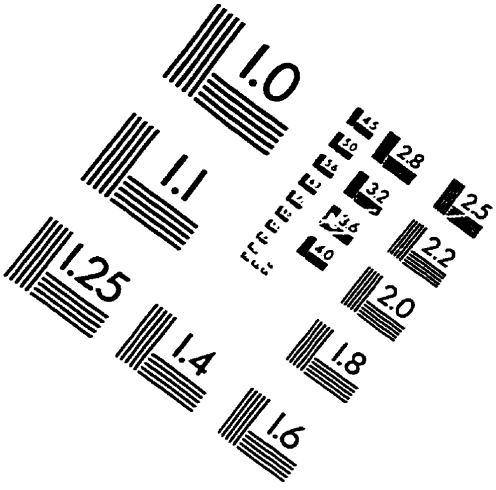
Schmidt, D. C., and Cleeland, C. (1997). Applying Patterns to Develop Extensible and Maintainable ORB Middleware, *Communications of the ACM*, Special Issue on Software Maintenance, Vol. 40, No. 12, December 1997.

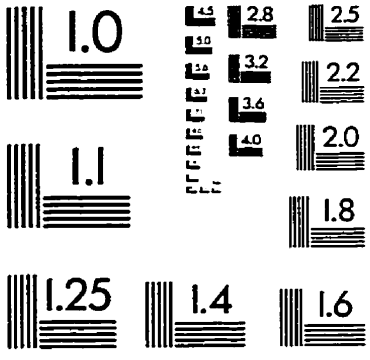
Vlissides, J. (1997). Patterns: The Top Ten Misconceptions, *Object Magazine*, March 1997.

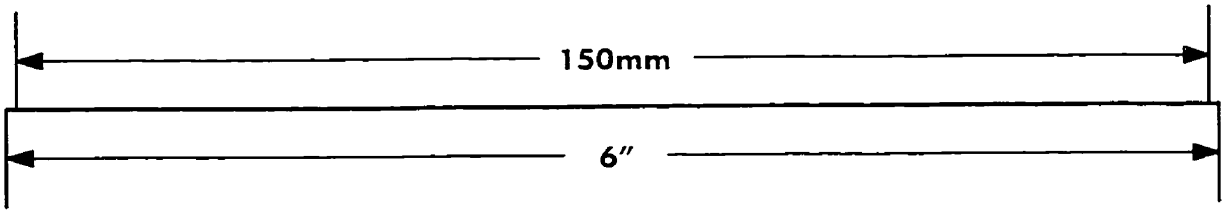
Wright, J.R., Weixelbaum, E.S., Brown, K., Vesonder, G.T., Palmer, S.R., Berman, J.I., and Moore, H.H. (1993). A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T network systems, *Proceedings of the Innovative Applications of Artificial Intelligence Conference*, pp.183–193, 1993.

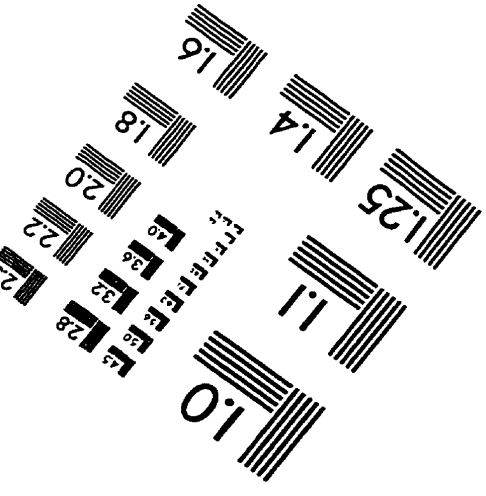
IMAGE EVALUATION
TEST TARGET (QA-3)

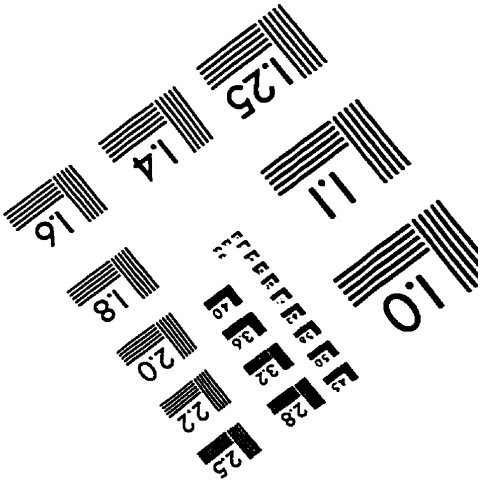












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989