ADAP: A component-based model using design patterns with applications in E-Commerce

Katrina Yun Ji

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd-project

Part of the Software Engineering Commons

Recommended Citation
https://scholarworks.lib.csusb.edu/etd-project/1694

This Thesis is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.
ADAP: A COMPONENT-BASED MODEL USING DESIGN PATTERNS WITH APPLICATIONS IN E-COMMERCE

A Thesis Presented to the Faculty of California State University, San Bernardino

In Partial Fulfillment of the Requirements for the Degree Master of Science in Computer Science

by

Katrina Yun Ji

June 2000
A COMPONENT-BASED MODEL
USING DESIGN PATTERNS WITH APPLICATIONS IN E-COMMERCE

A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

by
Katrina Yun Ji
June 2000

Approved by:
Dr. Arturo I. Concepcion, Chair, Computer Science
Dr. Richard J. Botting
Dr. Tonglai Yu

23 May 2000
Date
ABSTRACT

Design patterns are core solutions to recurring problems. They provide reusability in software designs and rationale behind the design. They are also a common way of communication among software engineers, and can avoid the software development from scratch. To take advantage of design patterns, there have been efforts on automating the process of using design patterns. However, there is still no clearly defined model of using design patterns. This thesis proposes ADAP (Applying Designs and Patterns), a component-based model of applying design patterns, with a well defined process of extracting the software design from generic design patterns up to implementation of the software design.

ADAP model defines five states when applying design patterns: generic and domain-specific, concrete, specific, integrated, and implemented design patterns. The generic and domain-specific design patterns are original descriptions of solutions to recurring problems in natural languages. The concrete design patterns eliminate the ambiguities in the solution for a software development. The specific design patterns apply the concrete pattern in the specific environment. The integrated design pattern consists of all specific design patterns in the design. Finally, the system is implemented based on the integrated
design pattern. This model is not only guidance for software engineers, but also a basis for automating the process of using design patterns. In addition, the ADAP Model suggests representations of different levels of design patterns that provide documentation of the software design. The representation provides a good way to understand the design and makes it easier to maintain the software in the future. The ADAP Model is suitable for E-Commerce applications because of the component-based technology that is used in building them. To show the applicability of the model, there is an illustration of how to design and implement mobile agents for an E-Commerce application.
ACKNOWLEDGMENTS

21 months have passed. Time certainly flies. It looks like my study at California State University, San Bernardino (CSUSB) is quickly approaching an end. It is hard to believe that I have learned so much from outstanding scholars, teachers and friends here at CSUSB. Looking back, before this learning experience, I had only limited knowledge in computer science. Now I feel that I have advanced in many ways. In the area of software engineering, for example, I gained much insight into software reusability with design patterns. It turns out to be a very interesting research area, which I have incorporated into my thesis work.

As I am writing this acknowledgement, I can not help but wondering how I can ever get this far. In the beginning of the thesis work, I encountered great difficulties and often wonder how I would proceed to a full closure. I am grateful that, during my down times, my thesis advisor, Dr. Concepcion, has been very instrumental in pushing my thesis work through. He has always been encouraging throughout the whole process. Without his guidance, I can hardly imagine where I will stand now.
I would also like to use this opportunity to thank my other committee members, Drs. Botting and Yu, whose feedback and suggestions have strengthened this research in major ways. Dr. Botting's research collection on software engineering has eased my literature review on the thesis topics. Dr. Yu's comments on the thesis draft help me to improve on the overall quality of this research.

Finally, I want to thank my parents and my brother, whose timely encouragement and caring have kept me going. I would also like to thank my church friends, whose constant prayers have been very much appreciated.
TABLE OF CONTENTS

ABSTRACT ......................................................... iii
ACKNOWLEDGMENTS .............................................. v
TABLE OF CONTENTS .............................................. vii
LIST OF TABLES ................................................ x
LIST OF ILLUSTRATIONS ......................................... xi

CHAPTER ONE Introduction

1.1 SOFTWARE REUSABILITY .............................. 1
1.2 Design Patterns as a Tool for Design Reuse ........ 3
1.3 Motivation for the Research ......................... 5
1.4 Research Goal: A Model of Applying Design
   Patterns - ADAP ................................................ 7
1.5 Organization of this thesis .......................... 8

CHAPTER TWO Design Patterns

2.1 Background of Design Patterns ....................... 10
2.2 Design patterns
   2.2.1 Description of design patterns .................. 16
   2.2.2 An example of a design pattern ............... 19
   2.2.3 Organization of design patterns .............. 22
   2.2.4 Different kinds of design patterns .......... 23
2.3 Current applications of design patterns .............. 26
   2.3.1 SPOOL project ....................................... 27
   2.3.2 Tool support for design patterns ............. 28
   2.3.3 Automatic code generation ..................... 31
   2.3.4 Research on design patterns .................. 32
CHAPTER THREE

A component-based model using design patterns (ADAP).

3.1 Motivation .................................................. 34
3.2 The process as suggested by GOF .......................... 38
3.3 The characteristic of design patterns ...................... 39
3.4 Goals of ADAP ............................................. 41
3.5 ADAP - The process of applying design patterns ....... 41

3.5.1 Generic design patterns ................................. 42
3.5.2 Domain Specific Design Patterns ...................... 43
3.5.3 Concrete design patterns ............................... 45
3.5.4 Specific design patterns ............................... 48
3.5.5 Integrated design patterns ............................. 50
3.5.6 Implemented design patterns ........................... 51
3.5.7 Relationships between the different states ............ 51

3.6 Representation of design patterns

3.6.1 The reason to formalize design patterns .............. 52
3.6.2 Adequacy of DisCo Language ......................... 55
3.6.3 DisCo Language ........................................ 57
3.6.4 Unified Model Language(UML) ....................... 61

3.7 Concreteness - From generic design pattern to concrete
design pattern ................................................. 62

3.8 Specification - From concrete design patterns to
specific design pattern ...................................... 67

3.9 Integration - From specific design pattern to
integrated design pattern .................................... 70
# LIST OF TABLES

Table 2.1 Parts in a Design Pattern .............................. 18
Table 2.2 Design pattern space ................................. 23
Table 2.3 Architectural Patterns ............................... 26
Table 2.4 Composition of SPOOL environment .................. 28
Table 3.1 Expressions in a Concrete design pattern .......... 64
Table 3.2 Transformation from DisCo language to UML .......... 69
Table 4.1 Transition table of Master-Slave pattern .......... 87
Table 4.2 Transition table of Strategy pattern ............... 91
Table 4.3 Functions for each methods in the design .......... 93
LIST OF ILLUSTRATIONS

Figure 1.1 Four levels of software reuse . . . . . . . 3
Figure 1.2 Current methodology for using design patterns 6
Figure 2.1 Structure of the Factory Method pattern . . 20
Figure 2.2 Key Components in the environment of tool 
support for design patterns . . . . . . . 30
Figure 2.3 System layers of the Tool support for object-
oriented (design) patterns . . . . . . . 31
Figure 3.1 Structure of singleton Design Pattern . . . 35
Figure 3.2 Structure of master-slave pattern . . . . . 44
Figure 3.3 The structure of the Observer pattern . . . 46
Figure 3.4 UML diagram for a specific Observer pattern 49
Figure 3.5 The ADAP Model . . . . . . . . . . . . . . 52
Figure 3.6 An illustration of the Observer pattern . . 60
Figure 3.7 Integration of Template Method & Builder 
Pattern . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
Figure 4.1 Classes and interfaces in the Aglet API . . 77
Figure 4.2 A life cycle of an aglet . . . . . . . . . . 78
Figure 4.3 Architecture of the Mobile Agent 
Demonstration . . . . . . . . . . . . . . . . . . . . . . . 80
Figure 4.4 Structure and Participants of the Master-Slave 
Pattern . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
Figure 4.5 Structure and Participants of Strategy 
Pattern . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
Figure 4.6 Specific Master-Slave pattern . . . . . . 88
Figure 4.7 Specific Strategy pattern for the Demonstration ................. 90
Figure 4.8 Integrated pattern for the demonstration .. 94
Figure 5.1 The ADAP model ........................................... 97
Figure B.1 Structure of Builder Pattern ................. 102
Figure B.2 Structure of Template Pattern ................. 102
Chapter One

Introduction

1.1 Software Reusability

In the area of software development, software engineers and programmers always make great efforts to reduce costs as well as to increase software productivity and its reliability. Among those efforts, software reusability, a way to avoid software development from scratch, has been extensively studied and implemented. In view of the current technological trend and the existing reuse techniques, this thesis is focused on software design reuse technology, with a special emphasis on design patterns that would provide core solutions to some of the recurring problems commonly occur in the software design processes. Design patterns are a relatively new research area that will facilitate achieving a higher level of reusability in software development.

In general, there are the following four different levels of reusability in software development: source code reuse, object reuse, component reuse, and design reuse. The following is a brief explanation of each:

(1) Source code reuse: This type of reuse uses previously-written pieces of codes. It exists in
most programming languages. By calling available functions and subroutines from the library, a programmer thus saves the trouble of writing lines of codes from scratch.

(2) Object reuse: An object is an instance of a class. A class defines a collection of objects with common attributes and behaviors. Object reuse encapsulates the states and methods of an object inside a particular class. This level of reuse originated from the object-oriented paradigm. It is easier for later reuse of the class in a different context. Inheritance is also a way to reuse properties of a class. Features of an object in a parent class are passed down (i.e. inherited) to child classes, thus preserving the attributes and methods already included in that parent class.

(3) Component reuse: According to Brown [Brown 1998], component reuse is a "... process of building systems by way of combination and integration of pre-engineered and pre-tested software objects." It is an evolution of the object reuse technology that uses a higher level of abstraction than that
Design reuse: It is to make use of good solutions in prior software designs. Comparing with the aforementioned three levels of reuse, design reuse contains even higher level of abstraction. It reuses ideas in stead of codes.

As I mentioned earlier, the major focus of this thesis is on design reuse, particularly on design patterns and their applications in design reuse. I will provide further details on this subject in the remaining parts of the thesis.

Figure 1.1 is an illustration of the four levels of software reuse.

<table>
<thead>
<tr>
<th>Source Code Reuse</th>
<th>Object reuse</th>
<th>Component reuse</th>
<th>Design reuse</th>
</tr>
</thead>
</table>

Figure 1.1 Four levels of software reuse.

1.2 Design Patterns as a Tool for Design Reuse

Design reuse has become a popular topic in software engineering for several years. The concept of design
patterns in software engineering originated from the works of Alexander [1977,1979], an architect, who wrote a couple of books on the topic of pattern language in architecture. In the 1990s, when object-oriented concept was becoming popular, people from the software development community discovered that Alexander's concept of pattern language could also be applied to software design. The concept of pattern language in architecture could be applied to software design. After the book *Design Pattern: Elements of Reusable Object-Oriented Software* by Gamma et al. [1995], the concept of using design patterns in software development has been widely accepted as viable. There have been more and more reports on successful applications of using design patterns in software engineering. There are also many on-going design patterns' discussion groups being active around the world [Hillside 2000].

Design patterns are available core solutions to recurring problems. They are descriptions of key elements in a solution that were written by experienced software engineers. According to Gamma et al. [1995], each solution could be applicable to different contexts in different ways. Therefore, the solution is flexible and extensible, thus allowing a higher level of reusability. By applying design patterns to software development, it is feasible for an inexperienced developer to reuse a
previously-proven solution without undergoing a prolonged training or extensive work experience.

Design patterns reveal the rationale behind the design processes. Each pattern is a solution to a problem. In the pattern, all classes that have contributed to solving prior problems remain in the system. By applying those proven design patterns, software design becomes a process of problem solving. In addition, it forgoes the concern of the behaviour of individual classes that make up the design patterns.

Design patterns provide a better way to understand a complex system. Furthermore, they are good communication tools with which software engineers get to know design concepts at an abstract level.

1.3 Motivation for the Research

Using design patterns in software development is not as easy as reusing pieces of codes, an object, or a component. Design patterns are in the abstract level. In the description of design patterns, there are only key elements such as classes that may contain necessary attributes, methods, and relationships among classes that contribute to the solution in the pattern. In a new application, different attributes and methods need to be added to the classes.
Figure 1.2 below shows that, in the implementation of software systems, it is hard to distinguish these elements by just reading the programming codes, due to the transition from design patterns to programming codes in the process of current software development. In software engineering, it is essential to follow a model and to document the whole process of applying design patterns.

In this research, I will define such model and document such process from the abstract level to the implementation level. The proposed model will direct the software developers from the description of design patterns in the abstract level to software designs that will be implemented in a specific programming language.

![Diagram]

Figure 1.2 Current methodology for using design patterns.

The SPOOL project (Spreading Desirable Properties into the design of Object Oriented, Large Scale Software Systems) by Keller [1999] was able to extract design patterns through the reverse engineering of programming source codes. Another project by Florign [1999] has
focused on code generation in design patterns. However, neither researcher has come up with a clearly-defined model with which to apply design patterns.

1.4 Research Goal: A Model of Applying Design Patterns - ADAP

As was mentioned in the above section, this thesis will propose a model of applying design patterns - abbreviated as ADAP (Applying Design And Patterns) in this research. The model incorporates two more levels in order to ease the transition from the abstract level to the implemented level as well as to provide additional guidance for software developers.

The ADAP is not only a model, it is also a basis of automating design pattern applications. It also builds a basis for developing tools for applying design patterns. Without automating the whole process, applying design patterns will be laborious and error-prone, particularly for complex systems. Design patterns can be effectively used in software designs when a supporting tool is available. The ADAP model will facilitate the development of such new tools in software design.

By applying design patterns, software engineers could keep up with new technologies in a short time. This advantage makes design patterns very useful in building
applications relating to E-commerce or Internet in areas such as network security, transaction processing, mobile agents, instant messaging, online advertisements, micro-browser for mobile users, and so on. In this research, I will illustrate how the ADAP can be applied in the design of a system of mobile agents. More details on the conceptual background and motivation for this research can be seen in Chapter Three.

1.5 Organization of the Thesis

The remaining sections of the thesis will be organized in the following fashion: Chapter 2 introduces design patterns including the background, the description, and examples of design patterns. In addition, several ongoing projects relating to the automation of using design patterns will also be briefly reported in the chapter.

Chapter 3 is a detailed description of the proposed model for applying design patterns (ADAP), which includes definitions of different states of design patterns and processes.

Chapter 4 provides an example of ADAP application to the software design in mobile agents. This example illustrates, step-by-step, how ADAP directs the software development in mobile agents to take place. This illustrative system is implemented in Java language.
Finally, Chapter 5 summarizes the contribution of ADAP to software reuse, lists areas for future research, and reports on limitations of the research.
Chapter Two

Design Patterns

2.1 Conceptual Background of Design Patterns

The concept of design patterns in software engineering originated from the works of Alexander [1977, 1979]. Two of his books that presented an entirely new attitude towards architectural design are *The Timeless Way of Building* [Alexander 1979] and *A Pattern Language* [Alexander 1977].

From Alexander's observation, the most beautiful structures in the world are made by people who live in the place where the structure is located, but are not made by architects. Since architectural design needs professional techniques, Alexander began to recognize patterns in the architecture around the 1960s. With those recognized patterns, he was able to provide solutions to different architectural design problems in the form of fixed patterns. These patterns could be used even for ordinary, inexperienced people, who desired to design buildings by themselves. Alexander pointed out that, unless buildings are made by all the people in the society where those buildings located, and that those people share a common pattern language, the towns and buildings would not become alive.
Alexander wrote descriptions of patterns by recognizing similarities between several well-regarded architectures, in the form of the design problems and their solutions. Each solution provides relationships needed to solve a problem in a general and abstract fashion. By adapting a particular design that had been used to solve previous problems, lay-architects were able to come up with novel designs that fulfil their preferences and meet local conditions in the place where the new structure is located.

In his other book, A Pattern Language [1977], Alexander described 253 patterns that can be used to create infinite combinations in architectural designs. Alexander reported the following three groups in the pattern language: the global group, the building-level group, and the detail-level group.

- Patterns 1 through 94 are global patterns that define a town or a community. For example, Patterns 16 through 20 define how to connect communities with each other:
  
  Pattern 16: Web of public transportation.
  Pattern 17: Ring Roads
  Pattern 18: Network of learning
  Pattern 19: Web of shopping
Pattern 20: Mini-buses.

There are 15 sub-groups in the global patterns, such as Patterns 28 through 34 that are used for the formation of local centers, while Patterns 87 through 94 are for the local shops and gathering places.

- Patterns 95 through 204 are building-level patterns that define individual buildings and space between buildings. For example, Patterns 169 through 178 define how to decide the arrangement of gardens and places in the gardens. There are 13 sub-groups in this category, such as Patterns 119 through 126 are used for the paths and squares between the buildings; and Patterns 190 through 196 are for fine-tuning the shapes and sizes of rooms, and alcoves to make them precise and feasible to construct.

- Patterns 205 through 253 are detailed-level patterns that define how to construct a building in detail. For example, Patterns 221 through 225 define how to fix the exact positions for openings - the doors and windows, and frame these openings within the main structure of the building. There are 8 sub-groups in this last category, such as
Patterns 249 through 253 are used for completing the building with ornaments, lights, and colors; and Patterns 233 through 240 are for putting in the surfaces and indoor details.

By combining patterns from these 253 patterns, one can form a language to design his/her own favorite project. For example, the following ten patterns (which are by themselves design languages) are sample designs chosen from a combination of a thousand possible languages for a porch in front of a house:

- Private terrace on the street (Pattern 140)
- Sunny place (Pattern 161)
- Outdoor room (Pattern 163)
- Six-foot balcony (Pattern 167)
- Paths and goals (Pattern 120)
- Columns at the corners (Pattern 212)
- Front door bench (Pattern 242)
- Raised flowers (Pattern 245)
- Different chairs (Pattern 251)

The pattern language, written by Alexander, is similar to natural languages. There are many ways we can use the language. For example, we can write either an ordinary sentence or a poem in which each word may have more connotations. Similarly, to design a building with pattern language, one can design in a loose manner, or
refined to a more formal fashion. Alexander argued that patterns are well-researched solutions to recurring design problems. Designs that violate the derived patterns were noticeably less successful than those that followed them.

Over the last decade, after object-oriented design was widely accepted as a good technology for software development, people from the software community discovered Alexander's concept of pattern language. Although software design is supposedly a different domain from architectural design, software engineers found it helpful to have an abstract method, similar to that of Alexander's pattern language, to express the core solutions and the rationale behind software system designs.

Gamma et al. [1995] define design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." (p.3) They name and identify the key aspects of a common design structure such as factory method pattern or singleton pattern, provides a good way for creating a reusable object-oriented design. They assist a software engineer in the maintenance of software product, modification of existing programs, removal of programming bugs, and addition of new features.

According to Gabriel [1996], other benefits of design patterns are:
1) Design patterns identify what is important in the programming. With design patterns, programmers will be able to solve problems more rapidly by using solutions inherited in design patterns.

2) They provide a way of communication between software engineers and programmers in the abstract level.

3) They enable inexperienced programmers learn from the patterns directly without years of training or field experience.

Design patterns are not usually invented, instead they are derived empirically from observations by experienced software engineers and programmers. Therefore, the use of design patterns ensures the successful application of other programmers in different situations.

Although there has been research on software design patterns [e.g. Gamma et al. 1995; Eden 1999; Keller 1999], there has been a lack of scientific theory that explains why particular patterns work in certain designs. One of the objectives of this research is to propose a model of using design patterns. With this model, which illustrates the use of design patterns, we may enhance the understanding in the field of software reusability.
2.2 Design Patterns

2.2.1 Description of design patterns

Design patterns are core solutions to recurring problems in software design. They capture both static and dynamic structures and collaborations of components in successful solutions to problems, and give software engineers an easier way to communicate designs on a higher level of abstraction. Otherwise, for each new design, the designers have to think in terms of individual classes and their behaviors.

The solutions in the abstract level described by design patterns are different from the solutions in the implementation level. For example, an algorithm such as linked list or bubble sort could be used directly in source codes. But solutions for problems in design patterns are only concerned about core elements in the solution, but not a piece of code to be used directly. In different contexts or applications, the solution could be implemented in a different way. This is the reason why design patterns could only be given in the form of descriptions in natural languages.

Gamma et al. [1995] also describe each design pattern as including the following elements: pattern name, intent, motivation, applicability, structure, participant,
collaboration, consequence, implementation, sample code, known uses, and related patterns. These elements provide information such as where to apply the pattern, how to use it, and what the results will be, etc. Table 2.1 [Gamma et al. 1995] in below provides an explanation of each parts in a design pattern.

In addition to the parts in Table 2.1, there are other ways to describe a design pattern. For example, in Grand [1998], descriptions of each design pattern include:

Pattern name.

Synopsis that conveys the essence of the solution provided by the pattern.

Context that includes problems that the pattern addresses.

Forces that summarize the considerations that lead to a general solution presented in the solution section.

Solution.

Consequences.

Implementation.

Java API usage.

Code examples.

Related patterns.
<table>
<thead>
<tr>
<th>Elements</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent or purpose</td>
<td>A short statement that explains: what does the design pattern do?</td>
</tr>
<tr>
<td></td>
<td>What is its rational and intent?</td>
</tr>
<tr>
<td></td>
<td>What particular design issue or problem does it address?</td>
</tr>
<tr>
<td>Motivation</td>
<td>A scenario that illustrates a design problem and how the structure of</td>
</tr>
<tr>
<td></td>
<td>class and object in the pattern solve the problem.</td>
</tr>
<tr>
<td>Applicability</td>
<td>A list of situations that includes when the design pattern can be</td>
</tr>
<tr>
<td></td>
<td>applied, examples of poor designs that the pattern can address, and</td>
</tr>
<tr>
<td></td>
<td>how the user can recognize these situations.</td>
</tr>
<tr>
<td>Structure</td>
<td>A graphical presentation of the classes in the pattern by a notation on</td>
</tr>
<tr>
<td></td>
<td>OMT (Object Modelling Technique).</td>
</tr>
<tr>
<td>Participants</td>
<td>A list of classes and/or objects that participate in the design pattern</td>
</tr>
<tr>
<td></td>
<td>and their responsibilities.</td>
</tr>
<tr>
<td>Collaborations</td>
<td>A description that explains how the participants collaborate in order</td>
</tr>
<tr>
<td></td>
<td>to carry out their responsibilities.</td>
</tr>
<tr>
<td>Consequences</td>
<td>An explanation of how the pattern supports its objectives, what are the</td>
</tr>
<tr>
<td></td>
<td>trade-offs and results of using the pattern, and what aspects of the</td>
</tr>
<tr>
<td></td>
<td>system structure that may change independently.</td>
</tr>
<tr>
<td>Implementation</td>
<td>A list of descriptions about pitfalls, hints, or techniques that could</td>
</tr>
<tr>
<td></td>
<td>cause problems when implementing the pattern, as well as other language-</td>
</tr>
<tr>
<td></td>
<td>specific issues.</td>
</tr>
<tr>
<td>Sample Code</td>
<td>An illustration of how to implement the pattern in C++, Java, or Small-</td>
</tr>
<tr>
<td></td>
<td>talk.</td>
</tr>
<tr>
<td>Known Uses</td>
<td>An introduction of where to successfully use the pattern in specific</td>
</tr>
<tr>
<td></td>
<td>systems.</td>
</tr>
<tr>
<td>Related patterns</td>
<td>A list of related design patterns and their differences.</td>
</tr>
</tbody>
</table>

Table 2.1 Parts in a Design Pattern
Code examples in Grand [1998] are given in Java language. Although design patterns are described in a different way, key elements in the solution of the same design pattern remain the same as those in Gamma [1995].

2.2.2 An example of a design pattern.

One essential pattern given in Gamma et al. [1995] is called Factory Method pattern. Corresponding to Table 2.1, this pattern works in the following way:

1. The intent of Factory Method pattern is to provide an application-independent object. It lets a class defer instantiation to subclasses. The interface defined in this pattern allows an object's subclasses to decide which class to instantiate.

2. The motivation of the pattern is how to keep a framework from creating its application-specific subclasses directly. If a framework creates every application-specific subclass, it will be difficult to have flexibility in the system.

3. The applicability of this pattern. According to Gamma et al. [1995, p108] Applicability includes the following three situations:
(1) A class can't anticipate the class of objects it must create.

(2) A class wants its subclasses to specify the objects it creates.

(3) Classes delegate responsibility to one of several helper subclasses, and it is necessary to localize the knowledge of subclass to the delegated subclass itself.

4. The structure of the pattern is in Figure 2.1.

![Diagram](image)

**Figure 2.1 Structure of the Factory Method design pattern**

5. The solution of this pattern is to encapsulate the knowledge of application-specific subclasses and move this knowledge out of the framework. Using the solution in Figure 2.1, the system is flexible and reusable.
In Figure 2.1, the Product class defines the interface of objects that are created by the Factory Method. The Concrete Product implements the Product interface.

The Creator class declares the FactoryMethod. It may call the FactoryMethod to create a Product object. The ConcreteCreator overrides the FactoryMethod in the Creator class to return an instance of a ConcreteProduct.

6. The consequences of the Factory Method pattern are:

(1) The Creator class is independent of the creation of concrete product classes. This is more flexible than creating an object directly in the Creator class.

(2) The set of product classes that can be instantiated may change dynamically.

7. When implementing this pattern, the following issues should be considered:

(1) There are two cases in the Factory Method pattern: (a) the Creator class is an abstract class and does not provide an implementation for the FactoryMethod it declares.

(b) the Creator provides a default implementation for the FactoryMethod.
If there are multiple products that need to be created by the pattern, the FactoryMethod takes a parameter that identifies the kind of object to create.

In different languages, the implementation of this pattern may also vary. For example, in C++, FactoryMethod is always virtual function and is often pure virtual.

In C++, it may use templates to avoid subclassing.

In all applications, it uses correct naming conventions.

Code examples of using Factory Method pattern are in Appendix A.

Related Patterns are Abstract Factory, Template Method, Prototype, and Hashed Adapter Objects.

### 2.2.3 Organization of design patterns

There are a total of 23 patterns described in Gamma et al. [1995]. The patterns are grouped into the three categories: creational patterns, structural patterns, and behavioral patterns. Table 2.2 shows how the 23 patterns are grouped into different abstract levels according to Gamma et al. [1995].
Table 2.2: Design pattern space

2.2.4 Different kinds of design patterns

In addition to the widely accepted 23 patterns in Gamma et al. [1995], there are also other design patterns as well as different ways of grouping these design patterns. For example, Grand [1998] lists 41 design patterns. Parts of these patterns are from the 23 patterns in Gamma et al. The 41 design patterns are organized into the following six categories:

1. Fundamental design patterns: Patterns that are most likely to be used extensively in other design patterns. For example, the Proxy pattern.
2. **Creational patterns**: Patterns that provide guidance on how to create objects and the creation needs dynamic decision making. For example: the Singleton pattern.

3. **Partitioning patterns**: Patterns that describe guidance on how to partition complex factors and concepts into multiple classes. For example, the Layered Initialization Pattern.

4. **Structural Patterns**: Patterns that describe common ways of how different types of objects can be organized to work with each other. For example, the Bridge Pattern.

5. **Behavioral patterns**: Patterns that can be used to organize, manage, and combine behavior. For example, the Observer Pattern.

There are also design patterns in the same abstraction level that may be applicable in specific domains, such as design patterns for network services in distributed systems [Schmidt 1996], mobile agents [Lange 1998], and building simulators' modelling [PsiGene 1999]. These patterns are domain-specific design patterns.

There are many pattern discussion groups. In these groups, experienced software engineers and programmers
discuss new patterns developed from their experiences [Hillside 2000].

As mentioned above, design patterns are in an abstract level. Inside the abstract level, there are three different levels: architectural patterns, design patterns, and idioms. Since design patterns have been discussed, the following is a brief explanation of architectural patterns and idioms:

(1) Architecture patterns provide solutions to specify the fundamental structure of a software system. Buschmann [1996] defined architectural patterns as "... fundamental structural organization schema for software systems." Table 2.3 below shows the four categories of architectural patterns.

(2) Idioms describe how to solve implementation-specific problems in a programming language. A collection of related idioms defines a programming style [Buschmann 1996].
Table 2.3. Architectural Patterns

2.3 Current Applications of Design Patterns

According to Agerbo [1998], when compared with traditional software designs, the application of design patterns in software development has the following advantages:

1) They encapsulate experience.

2) They provide a common way for computer scientists to communicate with each other coming from different background.
3) They enhance the documentation of software designs.

To take advantage of design patterns in software development, several efforts have been made on applications of design patterns. The following sections introduce three of those projects:

2.3.1 The SPOOL project

The SPOOL (Spreading Desirable Properties into the design of Object Oriented, Large Scale Software Systems) project [Keller 1999] is aimed at "... devising a systematic transition approach among analyses, designs, and implementations with the objective of improved software maintainability." SPOOL provides a reverse-engineering environment that helps to understand software by its organization around patterns. This project is a collaboration between Bell Canada and GELO [GELO 2000] group at the University of Montreal. The collaboration is motivated by the difficulties when large telecommunications companies have to adapt their software systems with millions of lines of code to rapidly changing requirements. In such systems, maintenance is very difficult when there are no methodological support during the software development.

The SPOOL environment is composed of source code capturing, design representation, design repository, and
pattern-based design recovery. Their functions are as follows:

The SPOOL model supports C++ at this time.

<table>
<thead>
<tr>
<th>Composition</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code capturing</td>
<td>Extracts an initial model from the existing source code.</td>
</tr>
<tr>
<td>Design representation</td>
<td>Represents interactive visualization and refinements of source code models, abstract design components, and implemented components.</td>
</tr>
<tr>
<td>Design Repository</td>
<td>Provides centralized storage, manipulation, and queries of the source code models.</td>
</tr>
<tr>
<td>Pattern-Based Design Recovery</td>
<td>Helps to structure parts of class diagrams and to resemble pattern diagrams.</td>
</tr>
</tbody>
</table>

Table 2.4 Composition of SPOOL environment

2.3.2 Tool support for design patterns

Another project at the Computer Science Department of Utrecht University (the Netherlands) is concerned about using design patterns at the beginning of software design by forward engineering. They created a first prototype of an environment that aimed at developing object-oriented programs with design patterns. Florign [1999] suggested that pattern-based development environment consists of three mutually consistent views in different levels of abstraction:
(1) **Pattern level**: The places where patterns are put into the system, are linked together, or are replaced by another pattern.

(2) **Design level**: The place where the structure of designs in terms of classes, methods, association and inheritance relationships.

(3) **Code level**: The place where the source codes a particular programming language is stored.

This tool assists software developers using patterns in the following three ways:

(1) Generating program elements (e.g. classes, hierarchies) for a new instance of a pattern, taken from an extensible collection of "template" patterns.

(2) Integrating pattern occurrences with the rest of the program by binding program elements to a role in a pattern (e.g. indicating that an existing class plays a particular role in a pattern instance).

(3) Checking whether occurrences of patterns that still meet invariants governing the patterns and repairing the program in case of problems.

The structure of this tool environment is shown in Figure 2.2.
The design of this tool supports the use of patterns both in forward engineering (i.e. documenting occurrences of patterns in existing programs) and in reverse engineering (i.e. modifying the program to better reflect the pattern's structure.)

In this project, there is a sophisticated Graphic User Interface (GUI) for large system designs that provide a friendly interface. The GUI has two components, the system browser and the fragment inspector. There are also
two other layers in the system, which are shown in Figure 2.3.

<table>
<thead>
<tr>
<th>GUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser, inspector</td>
</tr>
<tr>
<td><strong>Fragment Model</strong></td>
</tr>
<tr>
<td>design pattern, fragment, role actor</td>
</tr>
<tr>
<td><strong>Implementation Level</strong></td>
</tr>
<tr>
<td>fragment, slot, message</td>
</tr>
</tbody>
</table>

Figure 2.3. System layers of the Tool support for object-oriented (design) patterns

2.3.3 Automatic code generation from design patterns

IBM implemented the automatic code generation for the application of design patterns in 1996. Budinsky [1996] reported that the system is designed based on the following three goals:

1. Fast turnaround: Because most of this work is new and experimental, the system can be modified as quickly as possible. The designers intended to avoid any delays in implementing or testing new functionality.

2. Flexibility: Any major changes should be well-contained. For example, supporting generating code in a different programming language should not force an overhaul of the entire system.
3. *Ease of specification:* There is a higher-level way to specify how code gets generated without limiting flexibility.

They intended not to use conventional development tools, user interface development tools, or support libraries. Their interface with end-users is simply a World Wide Web (WWW) browser that displays pages specified in hypertext markup language (HTML). All the descriptions of the design patterns from Gamma et al. [1995] are provided in a template. Whenever the user enters information into a Code Generation page, the browser transmits the information to the Mapper through the Web-standard Common Gateway Interface (CGI). The mapping descriptions are Perl scripts; hence the Mapper is a Perl interpreter. The Perl scripts invoke a Code GENeration Template (COGENT) interpreter, which serves as the Code Generator. COGENT is a simple code generation specification language developed by IBM in 1996.

2.3.4 Research on design patterns

According to Eden [1999], there are the following three types of activities in the research of design patterns:

1. The "discovery" or invention of new patterns.
2. The classification and description of design patterns.
3. The application of design patterns.

The previous sections introduced current projects that are in the area of application of design patterns. The automation of code generation in this area is helpful to software developers and the process of using design patterns may not be time consuming, boring and error-prone. As the SPOOL team pointed out, without methodological support during the software development, it is hard to maintain the complex software product, therefore this thesis is proposing the ADAP model in this research.

To summary this chapter, this thesis contributes to the research effort in the following two areas:

(1) Defining and proposing a model of applying design patterns, ADAP, so that an automated tool environment could be implemented.

(2) Documenting the application process.

Chapter 3 continues with the model definition and the process.
Chapter Three

A Component-Based Model for Applying Design Patterns (ADAP).

3.1 Motivation

Design patterns are descriptions of solutions to recurring problems in an abstract level. The descriptions in natural languages, as introduced in Chapter Two, explain why a design pattern exists, where, when and how to use it, and the trade-offs when using it. Therefore, in various contexts, the solution can be applied in a different way. The following are two examples of the implementation of a simple design pattern - Singleton pattern in Gamma [1995].

Singleton is a creational design pattern, which ensures that a class only has one instance in the system. It also provides a global point of access. The key in this pattern is to keep a private and static variable - singletonInstance, a private getInstance() method that ensures the instance of this class to be controlled by itself. Figure 3.1 is Singleton's structure presented in UML (United Modelling Language).
A sample code in C++ that is one way to apply the Singleton pattern for the MazeFactory class is:

```cpp
class MazeFactory{
    public:
        static MazeFactory* Instance();
        // existing interface goes here
    protected:
        MazeFactory();
    private:
        static MazeFactory* _instance;
};
```

The following is another way for subclassing the Singleton class.

```cpp
class Singleton {
    public:
        static void Register(char* name, Singleton*);
        static Singleton* Instance();
        static Singleton* Lookup (const char* name);
    protected:
        static Singleton* _lookup (const char* name);
    private:
```
In addition to the above two situations, this pattern could be used to control the number of instances in an application as well as to implement the pattern with other programming languages such as Java [Grand 1998].

The above examples indicate that, for different applications, the detailed implementation can be varied a lot. Even though this is a simple design pattern, there are still several ways to implement it. Only the core elements in the solution do not change.

As Chapter Two introduced, there are research groups studying and grouping common design patterns, and putting efforts in automating the use of design patterns. But there are only few research on the methodology and the process of applying design patterns. This is a reflection of the analysis in Chapter One, that there is no transition between design patterns in the abstract level and the programming code in the implemented level (see Figure 1.1). As Khriss [1999] pointed out, there needs to have a methodology directing software developers in a step-by-step procedure of how to apply design patterns, particularly for the automation of using design patterns. Therefore, this research proposes a clearly-defined process for using design patterns.
A model directing the whole process of applying design patterns, which defines how they evolve from the abstract level to the implemented code level, makes the best use of design patterns. The documentation of the whole process will store the rationale behind the software design, which will make it easier for maintaining the software product.

This thesis proposes a model that details and defines the process between the abstract level and the implemented level. The model is called the component-based model for applying design pattern (ADAP). This model also describes a step-by-step guidance for applying design patterns.

The next section introduces a process suggested by Gamma et al. [1995]. Section 3.3 analyzes the characteristics of design patterns. Section 3.4 introduces the goal of the model that directs how to apply design patterns to software development. Section 3.5 explains the details of the proposed model. Section 3.6 discusses the representation of design patterns. Sections 3.7 through 3.10 are detail guidance of each step in the ADAP model.
3.2 The process suggested by Gamma et al. [1995]

There is a brief suggestion by Gamma et al. [1995] on how to select a design pattern and how to use it. The following are the steps of how to select a design pattern:

1. Consider how design patterns solve design problems.
2. Scan intent section.
3. Study how patterns interrelate.
4. Study patterns of like purpose.
5. Examine a cause of redesign.
6. Consider what should be variable in your design.

These steps are guidance for selecting a design pattern correctly. This issue depends on developers' experiences, their understanding of design patterns, and their analysis of applied systems. The model proposed in this thesis is based on the assumption that all design patterns are selected correctly by software engineers.

Gamma et al. [1995] also suggested how to use a design pattern:

1. Read the pattern once through Structure, Participants, and Collaborations sections.
2. Look at the Sample Code section to see a concrete example of the pattern in code.
3. Choose names for pattern participants that are meaningful in the application context.
(4) Define the classes.
(5) Define application-specific names for operations in the pattern.
(6) Implement the operations to carry out the responsibilities and collaborations in the pattern.

As Gamma et al. [1995] pointed out, these steps "are just guidelines to get you started." They are helpful when applying design patterns. But software design requires more than simple guidelines.

Before proposing a model of applying design patterns, the next section analyzes the characteristics of design patterns.

3.3 The characteristics of design patterns

Design patterns describe core solutions of recurring problems in the abstract level. When a design pattern is used in a specific application, there are many ways to implement the design patterns in programming languages. Keller [1999] has proven that design patterns are a great help in understanding the complexity of large software systems.

Unlike a hardware product or a completed building, an implemented software product still needs to be upgraded, maintained, added new features, or deleted old features.
This makes the documentation of the software design very important. In a large software system with hundreds of thousands or even millions of lines of source code, it is difficult to understand the system by reading them. Because design patterns tell rationales behind software designs, including why and how the software was structured, the proper documentation of applying design patterns can ease the system understanding and maintenance in the future.

By applying design patterns, the software development needs not to be done from scratch. It results in savings in development time and in the software life cycle.

The characteristics of design patterns are:

(1) Design patterns do not give solutions in a specific application environment, but provide detail descriptions of where, when, why and how the pattern should be used.

(2) They give key elements such as participants, structure, and collaborations that contribute to the solution of the problem.

(3) They include discussions of consequences and implementation when using the design patterns, and suggestions about considerations in different applied situations.
3.4 Goals of the ADAP model

As previous sections suggested that there is a need for a methodology to direct the process of applying design patterns from the descriptions to the implemented code. This thesis proposes such a methodology - a Model for Applying Design Patterns (ADAP). Research objectives of this thesis are as follows:

(1) To provide a step-by-step process of applying the solutions in design patterns in the abstract level to the programming language code in the implemented level. All steps in the ADAP model should be easily followed by software engineers.

(2) To provide a way of tracing the design process. Every step in the ADAP model should be documented to make it possible of tracing back to the original design patterns.

(3) To establish guidance for the future automation of the ADAP model that will be a tool with easy access and usage of design patterns for software designs.

3.5 ADAP Model - The process of applying design patterns.

After an extensive literature review on related subjects such as using, automating, analysing and formalization of design patterns, the ADAP model proposes
five states when applying design patterns in a specific software design. These five states are generic and domain-specific, concrete, specific, integrated, and implemented design patterns. In contrast, there are only two states in the existence literature. The following sections define the five states in the ADAP model.

3.5.1 Generic design patterns

Generic design patterns are descriptions of solutions in natural languages. They are generally published design patterns (e.g. Gamma [1995], Grand [1998]).

According to Gamma, elements of design patterns include intent or purpose, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses and related patterns. There are also other ways to describe design patterns. The purpose of the descriptions is to help understanding all aspects of the design patterns. Since the solutions given by design patterns are in the abstract level, they can not be used directly without considering the context in different applications. Only with a thorough understanding of design patterns and why they be used in a specific context can the design pattern be effectively and correctly used.
In the proposed model - ADAP, the original design patterns with descriptions of solutions are called generic design patterns.

3.5.2 Domain Specific Design Patterns

Domain specific design patterns are core solutions to problems in a specific domain. They are similar to generic design patterns and are at the same level of abstraction as generic design patterns. The difference between the two is that generic patterns are applicable in all areas of applications, while domain specific patterns may only be applicable in a certain domain application area.

For example, the Master-Slave pattern, found in the development of mobile agents, is a domain specific design pattern [Lange 1998]. This pattern defines a scheme whereby one agent, called master, can delegate a task to a slave agent. Figure 3.2 shows the structure of this pattern. Further details of this pattern will be provided in Chapter Four.
Figure 3.2 Structure of master-slave pattern

The description of the Master-Slave pattern consists of same elements as those of generic design patterns: intent, applicability, participants, collaboration, consequences, implementation, and sample code. But this pattern is only for applications in mobile agents, and it may not be applicable beyond this domain.

There are also other design patterns for different domain specific applications. For example, patterns for network and communication [Schmidt 1996] and for the modelling of building simulators [PsiGene 1999].

The reason why we put domain specific patterns in the same abstract level with generic patterns is that domain specific patterns are also core solutions in a description of natural languages. Although a domain specific design pattern may not be used in other applications, with modifications, it is still possible to apply it in the
domain. The process of applying the domain specific patterns is similar to that of generic design patterns.

3.5.3 Concrete design patterns

Concrete design patterns eliminate the ambiguities in the generic/domain-specific level. They are the first step in applying design patterns.

To eliminate the ambiguities, decisions must be made based on the considerations of trade-offs, pitfalls, hints, and techniques stated in the generic design pattern for an applied context. In the description of generic design patterns, the consequences and implementation parts help understanding the pattern and give clues to concretize patterns.

Here is an example of several possible concrete design patterns derived from one generic observer pattern:

The Observer pattern [Gamma 1995] defines a one-to-many dependency between objects so that when a subject changes state, all its dependants are notified and updated automatically. Figure 3.3 is the structure of Observer pattern:
Figure 3.3 Structure of the Observer pattern

In this pattern, when a subject changes its states, it will notify all its observers. The observers will update their states after receiving the notification. The benefit of this pattern is to keep abstract coupling between the Subject and the Observer, and to avoid being tightly-coupled. The Subject keeps a list of observers but do not know details of any Observers.

There are several ways to concretize this original pattern to a concrete observer pattern:

(1) An observer may depend on several subjects. In such case, an update interface should include the observer. When the observer updates its state, it will know which subject has changed its state.
(2) The update method may be triggered by a subject or by a client. The difference is that the client may wait until a series of states being changed before notifying the observer.

(3) When the subject broadcasts additional information about changes, the subject may send observers detailed information about the changes. This process is called the push model (Gamma [1995]). Otherwise the subject may send only notifications, and observers ask for details explicitly thereafter. This later process is called the pull mode (Gamma [1995] p298).

(4) The observers may register in a subject as only interested in specific events. In such case, the Attach and Update methods must have a parameter indicating the observers' interests. There may also be other ways to concretize the Observer pattern in other contexts. However, core solutions of this pattern remain the same.

This example also illustrates that, when applying a generic/domain-specific design pattern, the concrete state of the pattern is unavoidable. Without concreteness with considerations and decision-making based on generic design
patterns, there is no way to eliminate the ambiguities in the generic design patterns.

The concrete design patterns will not be represented by descriptions in natural languages as in generic design patterns. We will discuss how to represent concrete design patterns in the ADAP model in Section 3.6.

3.5.4 Specific design patterns

Specific design patterns include not only key solutions from concrete design patterns, but also detail designs for specific applications.

After the concrete design patterns eliminate ambiguities in the generic design states, the next step is to consider the specific requirements and situations for specific applications.

The solutions in the concrete design pattern contain only key elements including classes, methods, and relationships among classes that have contributions to solving the problem. They are not directly related to the current application. For different applications, the concrete design patterns must be modified to fit into the environment of the specific system. This is a process of renaming those elements in concrete design patterns, and adding new elements for system requirements. After the specific design states, the design patterns are ready to be implemented.
The following is an example of applying the observer pattern in a design for a gas station system. In a gas station, pumps control the dispensing of petrol, and screens show a volume of the petrol delivered. Figure 3.4 [Khriss 1999] is a UML diagram for a specific design pattern of the observer pattern.

![UML Diagram](image)

Figure 3.4 UML diagram for a specific Observer pattern.

In Figure 3.4, Pump class and Screen class are concrete classes or subclasses of Subject and Observer class respectively. Compared to Figure 3.3, some of the classes and methods in the concrete design pattern are
renamed and some application specific methods are added to the classes.

The above figure is a UML class diagram. The specific design pattern may also include sequence diagrams for the specific application that is based on the concrete design patterns. These diagrams form the analysis part of the specific design pattern.

The specification of design patterns includes renaming of classes and methods, cloning of classes, changing and adding attributes, methods, and classes. The specific design patterns are detailed software designs before implementation, and the design patterns are embedded in the applied environment. Section 3.8 will thoroughly discuss the specification in the ADAP model.

3.5.5 Integrated design patterns

Integrated design patterns combine more than one specific design pattern. One design pattern may only provide a solution to a problem. In designing for a complex system, there may be more than one applicable design pattern. Integration is a combination of all design patterns.

Varies ways to integrate design patterns may be applicable, including aggregation, composition, and containment. Fowler and Scott [2000] describe aggregation as the part-of relationship, composition as the part
object may belong to only one whole. Containment incorporates layers of design patterns into a particular design pattern.

After the concreteness and the specification of generic design patterns, the integration needs only to consider the interfaces and relationships among specific design patterns. The integrated design pattern is a completed software design. Section 3.9 will provide an example of an integrated design pattern using two specific design patterns.

3.5.6 Implemented design patterns

The implemented design patterns are programming codes. The implementation is based on the integrated design patterns. The system may be implemented in C++, Java, or other object-oriented programming languages. It will be hard to recognize the original design patterns in the programming code. But with the documentation of the whole process in the ADAP model, it will be easier to understand and maintain the implemented product.

3.5.7 The model for applying design patterns (ADAP)

Figure 3.5 shows the ADAP model and the relationships among the above defined states of design patterns.
3.6 Representations of design patterns

3.6.1 Representation of design patterns

The generic design patterns are described in natural languages with ambiguities. In the proposed ADAP process, different states of design patterns must be expressed in certain specifications. The model is not only a tool for
guidance, but also a method to document every step in the process. This representation is also essential when the model is to be developed as an automating tool for applying design patterns.

Although there has been prior research that intends to come up with a general method of formalizing design patterns, there is not a general agreed representation. For example, Free [1994] introduced the concept of meta-patterns. He used seven basic meta-patterns to represent design patterns on a meta-level. Florijn [1999] proposed fragment model that uses fragments to represent the structure of design patterns. Mikkonen [1998] discussed how to formalize design patterns by using DisCo Language. Eden [1999] used meta-language and proposed a pattern-wizard of transforming design patterns from one language to another language. However, none of the above research settles the formal representation issue.

According to Eden [1999], there are following arguments for the formalization of design patterns:

(1) Existing specifications contribute little or nothing to the understanding of when and how to use a design pattern.

(2) Patterns are abstractions, or generalizations, and therefore should be vague, ambiguous, and imprecise.
(3) Formalization is impossible because there is no fixed elements in patterns, and everything can be changed.

(4) Patterns are core solutions or concepts whose essence is intangible, elusive, and hence beyond the scope of a literal expression.

These arguments maybe reasonable if the formalization of design patterns aims to be the only method to represent design patterns instead of descriptions in natural languages.

In the ADAP model, design patterns are refined through concrete state where existing ambiguities have been removed, and through specific state where more application details are added, therefore the formal representation of design patterns are more rigid than the aforementioned representations in prior research.

The ADAP model is guidance of a procedure of clarify the ambiguities and imprecision in the original design patterns. Without eliminating ambiguities, the generic design patterns can not be applied in a specific context. Therefore, formalization of design patterns in the concrete, specific, and integrated design patterns in the ADAP model will not conflict with the current arguments about formalization of generic design patterns.
Generic/domain-specific design patterns are kept as descriptions in the ADAP model, and is available with helpful information at all times.

The ADAP model adopts the following two methods in design pattern representation: the Distributed Co-operation (DisCo) language for concrete design patterns, and the United Modelling Language (UML) for specific and integrated design patterns. This adoption is based on extensive research or reports in design pattern representations as well as on characteristics of the different states in the ADAP Model.

The following section affirms the adequacy of adopting the two languages in the ADAP model.

3.6.2 Adequacy of DisCo Language

The DisCo language, a specification method [Mikkonen 1998], is used to represent concrete design pattern in the ADAP model. There are six reasons the ADAP model adopts DisCo language:

1. DisCo is a specification language for distributed systems. This language "... is capable of describing a system together with its environment at a high level of abstraction" [DisCo 2000]

2. DisCo has a special feature that is different from other methods of formalization of design
patterns - DisCo has a formal basis in Temporal Logic of Actions [Lamport 1994].

3. DisCo was proved to be able to formalize design patterns by Mikkonen [1998]. Mikkonen also showed how to represent design patterns in DisCo. There is no real implementation in his paper. As he concluded in his paper (p124) "In brief, the DisCo method offers object-oriented modeling capabilities that can be used for developing specifications at a high level of abstraction, as well as well-defined semantics that enable rigorous reasoning."

4. DisCo may eliminate ambiguities in generic design pattern, while keeping key elements in expressions for concrete design patterns. Section 3.7 will show how to express concrete design patterns in DisCo language.

5. DisCo language is developed for distributed systems. The ADAP model is intended mainly for applications in E-Commerce. The distributed nature of the Internet ensures that DisCo language is appropriate to express concrete design patterns.

6. DisCo language focuses on interactions between components instead of individual objects. This
is coincidental with design pattern's concentrating on solutions with relationships among classes.

Section 3.6.3 is a brief introduction of DisCo language. The advantages of expressing concrete design patterns by DisCo language are as follows:

1. DisCo language can help software developers focus on the structure or relationships among classes, instead of behaviors of individual classes in the early stage of software design.

2. DisCo language can provide a clear documentation of how generic design patterns are concretized.

3. DisCo language's formal basis in Temporal Logic of Actions makes it possible to check errors in the design by technologies in artificial intelligent.

3.6.3 DisCo Language

The intent of DisCo method is for specification and modelling of interactions at a high-level of abstraction [DisCo 2000]. The meaning of a DisCo specification is the set of execution sequences it allows. Only exchanged information among participants is important, not who initiates such exchange. The formal basis of the method is in temporal logic of actions [Lamport 1994].
A DisCo specification is a definition of a system. In each system, the developer can introduce classes, relations and actions [DisCo 2000].

1. Definition of class:

The following is definition of a class. New classes can be defined, and imported classes can be extended in a system part:

```
class = class_definition | class_extension
class_definition = Class simple_name
                   [( formals_with_defaults_list) ] is
                   class_body
                   end [simple_name ];
class_body = { class_component }
class_component =
                 state_definition | variable_definition |
                 state_extension | inheritance |
                 local_assertion | local_initial_condition
```

(The detailed definition of the state_definition and can be found in [DisCo 2000]).

The class definition could also be defined simply as the following:

```
class_definition = Class class_name { attributes }
```

One simple example of a class definition is:

```
Class C={x}, defines class C, where x is an untyped variable.
```

2. Definition of Relation R
Relation (n) • R • (m): C X D, defines Relation R, where n is the number of instances of class C, and m is the number of instances of class D.

3. Definition of action or method

\[
\text{action} = \text{action}\_\text{definition} \mid \text{action}\_\text{transformation}
\]

\[
\text{action}\_\text{definition} =
\begin{align*}
\text{Action} &\ 
\text{simple}\_\text{name} \left[ \left( \text{formals}\_\text{list} \right) \right] \\
\text{by} &\ 
\text{formals}\_\text{list} \ 	ext{is} \\
\text{when} &\ 
\text{boolean}\_\text{expression} \ 	ext{do} \\
\ 
\text{[body]} \\
\text{end} &\ 
\text{[simple}\_\text{name}];
\end{align*}
\]

body = statement { statement }

\[
\text{statement} =
\begin{align*}
\text{multiple}\_\text{assignment} &\mid \\
\text{multiple}\_\text{state}\_\text{transition} &\mid \\
\text{conditional}\_\text{statement} &\mid \\
\text{local}\_\text{assertion} &
\end{align*}
\]

In the ADAP model, the action is simplified as the following:

\[
\text{Action} \ 	ext{simple}\_\text{name} \ 	ext{(role}\_\text{name}: \text{Class}\_\text{name}; \ 	ext{parameter}); \\
\text{enabling}\_\text{conditions} \rightarrow \text{result}\_\text{states}
\]

An example of Action definition is:

\[
\text{Action} \ A(c:C; \ I): \ I \neq c.x \rightarrow c.x' = I,
\]

It defines an action A, where c is a role for an object in class C, and I denotes an untyped value given as a parameter. Expression \( I \neq c.x \) is the enabling condition under which the action can be executed. Unprimed and primed variables refer to the values of variables before
and after the execution of the action, respectively, thus defining the state change caused by an execution of the action.

Each DisCo specification describes the temporal behavior of a closed system. It includes classes and the relationships between classes. The following is an example of expressing a design pattern in DisCo language.

Figure 3.6 is an illustration of the Observer pattern [Mikkonen 1998].

![Diagram of Observer pattern]

**Figure 3.6 An illustration of the Observer pattern**

The Observer pattern can be expressed in DisCo language as

```plaintext
class Subject = {Data},
class Observer = {Data}.

Relation (0..1) • Attached • (*): Subject X Observer.
Relation (0..1) • Updated • (*): Subject X Observer.

Attach(s:Subject; o:Observer):
    ¬s.Attached.o
```
\[ \rightarrow s.\text{Attached}.\ o \]

\text{Detach}(s: \text{Subject}; \ o: \text{Observer}):
\begin{align*}
& s.\text{Attached}.\ o \\
& \rightarrow \neg s.\text{Attached}.\ o
\end{align*}
\[ \wedge \neg s.\text{Updated}.\ o \]

\text{Notify}(s: \text{Subject}, \ d):
\begin{align*}
& \rightarrow \neg s.\text{Updated}.\ o \\
\text{class Observer} & \\
& \wedge s.\text{Data} = d
\end{align*}

The parameter \(d\) models the new value, set upon notification, and \text{class Observer} denotes all instances of the class Observer.

\begin{align*}
\text{Update}\ (s: \text{Subject}; \ o*: \text{Observer}; \ d):
& s.\text{Attached}.\ o \\
& \neg.\text{Updated}.\ o \\
& \wedge d = s.\text{Data}
\end{align*}
\[ \rightarrow s.\text{Updated}.\ o \\
\wedge o.\text{Data}=d. \]

The asterisk for the participant \(o\) denotes a fairness requirement, which means that if an object could repeatedly take this role in this action, the action will be executed for the object.

3.6.4 Unified Model Language (UML)

The Unified Model Language is a widely-used system design language. It is a suitable language in software design, particularly in the object-oriented paradigm. The UML is used to represent specific design pattern and integrated design patterns in the ADAP model.

After getting concrete design pattern in the ADAP model, more details of classes and methods relating to
application requirements need to be added to the design. The UML, which focuses on details of classes is suitable to represent the classes, attributes, and methods in the specific design patterns.

3.7 From generic/domain-specific design pattern to concrete design pattern - Concreteness

Concrete design pattern is the first step to apply generic design patterns. The concrete design pattern only focuses on those elements that contribute to solutions in generic design/domain-specific patterns. The elements are:

- Classes in the design patterns.
- Relationships between the classes in the design patterns.
- Methods and attributes which contribute to the purpose of the design patterns.

The concreteness of generic/domain-specific design patterns needs software developer's understanding of the generic/domain-specific design pattern, and the context in the target application systems. Generic/domain-specific design patterns include all helpful information for the concreteness. The following elements are related information for consideration:
• Applicability: Lists the situations of when the pattern should be used.

• Structure: Provides structures and key elements that must be kept in concrete design patterns.

• Participants: Explains contributions of each class in the pattern.

• Collaborations: Provides interactions between classes.

• Consequences: States trade-offs and some considerations when applying the pattern.

• Implementation: Explains all considerations during the implementation phases. Even though it is not necessary to implement the pattern at this stage, it will give some hints for the concrete design pattern.

In the ADAP model, the DisCo language is used to represent concrete design patterns. DisCo is a clear, easy to use language for representing concrete design patterns. With DisCo, the concrete design patterns can extract all key elements from the generic design pattern without ambiguities for the concrete context. When the ADAP model is to be automated in the future, the graphical animation tool based on DisCo language could be used to provide the visualization of the concrete design patterns.
Table 3.1 is expressions representing elements in the concrete design patterns:

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Class class-name = { class-attributes}</td>
</tr>
<tr>
<td>Extended-class</td>
<td>class class-name = class-name + {class-attribute}</td>
</tr>
<tr>
<td>Relation</td>
<td>Relation (n-instance-class1) • Relation-name • (m-instance-class2) : class-name1 X class-name2</td>
</tr>
<tr>
<td>Method</td>
<td>Method-name(role-name: class-name; parameter): enabling_conditions (\rightarrow) result_states</td>
</tr>
<tr>
<td>Extended-method</td>
<td>Method-name(role-name: class-name; parameter): Refines method-name(role-name: class-name; parameter): enabling_conditions (\rightarrow) result_states</td>
</tr>
<tr>
<td>Inheritance class</td>
<td>class subclass-name = superclass-name + { class-attributes}</td>
</tr>
<tr>
<td>Method in Inheritance class</td>
<td>Method-name (role-name, class-name; parameter): Refines method-name1(role-name1: class-name1; parameter1) for role-name1 \in\ class-name</td>
</tr>
</tbody>
</table>

Table 3.1 Expressions in a Concrete design pattern

As a middle state of design patterns in the ADAP model, the concrete design pattern does not have ambiguities comparing with the generic design pattern. After considering the applicability, structure, participants, collaborations, consequences, and implementation in the generic design patterns, decisions concerning trade-offs, hints, suggestions, and the applied system were made. The resulting concrete design pattern is a concrete solution for the application system being developed. The concreteness completes the following two tasks:
(1) Keeping all the key information in generic design patterns.

(2) Choosing a concrete solution for the applied system based on considerations of the trade-offs, hints, and suggestions in generic design patterns.

The concrete design patterns in the ADAP model have the following two properties:

(1) Give the reasons of the existence of all classes and methods.

(2) Provide the first documentation in the abstract level in the software design.

Without the concreteness of eliminating the ambiguities in generic design patterns, it is unfeasible to apply the generic design pattern to software design.

The following is an example of concrete design patterns of the observer design pattern introduced in Section 3.6.3:

From the action Notify\(s:Subject, d\) in the formalizing expression; we have

Expression 1.

\[
\text{Notify}(s:Subject, d) : \rightarrow \neg s.\text{Updated} . \text{class Observer} \\
\wedge s.\text{Data}' = d
\]

The subject is responsible to trigger the update.

The Update\(s:Subject, o:Observer, d\)
Update (s:Subject; o:Observer; d):
s.Attached.o
^\neg. Updated.o
\land d = s.Data
\rightarrow s.Updated'.o
\land o.Data'=d.

This expression indicates that this is a pull model.

First, the subject sends the notification, then the observer asks for details later.

The following is another way to concretize the observer pattern: the concrete solution is that the observer is attached to the subject according to the specific events of interest,

Expression 2.

class Subject = {Data},
class Observer = {Data}.

Relation (0..1) . Attached . (*): Subject X Observer.
Relation (0..1) . Updated . (*): Subject X Observer.

Attach(s:Subject; o:Observer; interest):
\neg s. Attached.o ^ o.interest
\rightarrow s.Updated'.o

Detach(s:Subject; o:Observer; interest):
s.Attached.o ^ o.interest
\rightarrow \neg s.Attached'.o
\land \neg s.Updated'.o

Notify(s:Subject, o.Observer; d; interest):
o.interest
\rightarrow \neg s.Updated'. oobserver
\land s.Data' = d,

Update (s:Subject; o*:Observer; d: interest):
s.Attached.o ^ o.interest
^\neg. Updated.o
\land d = s.Data
\rightarrow s.Updated.o
\land o.Data'=d.
As we have introduced in section 3.5.6 about the concrete design pattern, there are many choices to concretize the generic/domain-specific pattern. The above examples are only two ways to concretize the observer design pattern. With DisCo language, the concrete design pattern can express the solution clearly without ambiguities, and emphasis on the behaviors among classes. The concrete design patterns will then evolve to the specific design pattern with details in the individual classes.

3.8 From concrete design patterns to specific design patterns - Specification

The specific design pattern is the next step after the concrete design patterns with solutions for a problem. The specific design patterns direct software developers to focus on applying the concrete design patterns to the specific environment and adding other feathers for the applied system.

The considerations in the specific design patterns are different from those in the concrete design patterns. While concreteness focuses on eliminating ambiguity in the generic/domain-specific design pattern, the specification concentrates on the application of the concrete design
pattern to the system being designed. Compared to the more decision-making oriented activities - based on the references to the generic design patterns - in the concreteness, the specification of concrete design patterns is more application-oriented, including changes in classes, attributes, and methods - based on the application.

Modifications included in the specification are:

- Rename elements in the concrete design pattern, such as class, method, attributes, and parameters.
- Extend the classes and methods, when it is necessary for the applied system.
- Add classes or methods for other system requirements.
- Add more details in the classes or methods according to the specific applied system.

After the modification, the concrete design pattern will be embedded in the specific design pattern, and it will be hard to distinguish key elements originated from the concrete design pattern. To keep documentation of the evolution of a generic design pattern, the ADAP model keeps a transition table between the concrete design pattern and the specific design pattern.
The transition table is a mapping from class names, method names, and attribute names in the concrete design pattern to those in the specific design pattern. This table helps tracing back from the specific design pattern to the concrete design pattern.

In the ADAP model, the specific design pattern is expressed by UML because UML emphasizes on the individual classes and their behaviors. After detailed design for the application requirements is established, the specific design pattern is ready to be implemented.

To use UML language, there is a transformation from the DisCo language to the UML. Table 3.2 gives more details:

<table>
<thead>
<tr>
<th>DisCo Language</th>
<th>UML language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class-name</td>
<td>Class-name</td>
</tr>
<tr>
<td>Method-name</td>
<td>Operation-name</td>
</tr>
<tr>
<td>Attribute, parameter</td>
<td>Parameter</td>
</tr>
<tr>
<td>Conditions and results</td>
<td>Pseudo code</td>
</tr>
<tr>
<td>Role name</td>
<td>Role name</td>
</tr>
</tbody>
</table>

Table 3.2 Transformation from DisCo language to UML

Figure 3.4 in Section 3.5.4 is an example of a specific design pattern.

After this transformation, an object-oriented methodology for the software design can be used for more
detailed considerations for the specific design patterns relating to the applied system.

In a specific design pattern, the structure given in the generic design patterns has been embedded. By keeping the concrete design pattern in the DisCo language and the transition table, the contribution of the classes, methods, and why they exist in the specific design pattern are clearly documented. The documentation provides a method to understand the design during and after the software development.

The evolution of a design pattern from the generic, to the concrete, and then to the specific state is a process for applying all individual design patterns. The integration of more than one design pattern in the system development is the next step in the ADAP model.

3.9 From specific design pattern to integrated design pattern - Integration

Design patterns provide core solutions for some recurring problems. However, most of the time, there are many problems to be solved in a software design. After the concreteness and specification of the individual generic design pattern, integration of the specific design patterns is the last step to form a component of the system being designed.
During the software development, design patterns can be used to solve various problems. Whichever pattern is used in solving a problem, the ADAP model is a methodology for software engineers to apply a design pattern -- from generic/domain-specific design patterns, concrete design patterns, to specific design patterns -- until it is time to consider how to integrate all specific design patterns.

Figure 3.7 shows an example of integrating Template Method and Builder pattern in an application of the data set construction. Both the Template Method and the Builder pattern are generic design patterns in Gamma et al. [1995]. The structure of the Template Method pattern and the Builder pattern are given in Appendix B. The purpose of this design is to create a data set independent of the data source. [Masuda 1998]

In Figure 3.7, the Builder pattern solves the problem of separating the construction of a complex object from its representation for the creation of Dataset. This solution makes it possible to change the internal representation of a data set object. The Template pattern is applied to solving the problem of creating the Dataset from different data sources. It reuses the skeleton of the data creation algorithm.
By following the ADAP model, the software design is a process of problem solving with available solutions in generic design patterns. In the integration phase, all problems have been solved and the software design is ready to be implemented.

### 3.10 Implementation

After integrating all the design patterns, the next step is to implement the design in object-oriented programming code. The implementation can be coded in specific programming languages such as C++ or Java.

With the guidance in the ADAP model, and the documentation of each step, software developers have
understood why and how the system is designed before implementation.

Next chapter is a demonstration of a complete application of the ADAP model for an E-commerce application, including the implemented code in Java language.
Chapter Four

Using ADAP in a an E-commerce Application

This chapter demonstrates how to use the proposed ADAP model in an actual software design for an E-commerce application. The demonstration will go through the evolution of design patterns from its generic level, to the concrete level, then the specific level, and finally integrating all specific patterns together. This design is implemented in Java Language. The source code can be found in Appendix C.

4.1 E-commerce Applications in Mobile Agents

Ma [1999] estimated that Internet-generated electronic transactions will grow exponentially, and there are at least $313 billion of profits out there in the next few years. The delivery of consumers' orders on the Internet is an acceptable way of doing business. However, current online purchases are still not automated. People must perform and involve in every step in the buying process, such as collecting information, making decisions about merchants and products, and entering purchase and payment information.

Among E-commerce technologies, such as network security, transaction processing, mobile agents, instant messaging, online advertisements, micro-browser for mobile users, and so on, Mobile agent is an emerging technology
that performs the buying process automatically instead of manually. Mobile agent is one of the most attractive technologies in E-commerce.

Lange [1998] defines a mobile agent as:

"... A mobile agent is not bound to the system where it begins execution. It has the unique ability to transport itself from one system in a network to another. The ability to travel allows a mobile agent to move to a system that contains an object with which the agent wants to interact and then to take advantage of being in the same host or network as the object."

He also provided details of background, evolution and mechanisms of the mobile agent. According to Lange, comparing with other network technologies, mobile agents have the following advantages:

1. They reduce the network load.
2. They overcoming network latency.
3. They encapsulate protocols.
4. They execute asynchronously and autonomously.
5. They adapt dynamically.
6. They are naturally heterogeneous.
7. They are robust and fault-tolerant.

There are three reasons why this thesis uses ADAP model in Mobile Agents:
(1) Mobile agents have great potentials in the booming E-commerce. By following the ADAP model when using design patterns, the demonstration shows the advantages of applying design patterns and the effectiveness of the ADAP model in this relatively new area.

(2) There are domain-specific design patterns in mobile agents. They are good examples to demonstrate how to apply domain specific design patterns.

(3) The Aglets Software Development Kit (ASDK) [Lange 1998], a framework supporting mobile agents that provides environment in our project, is light-weighted compared to other mobile agents environment, and it could be downloaded from the Internet. [http://www.trl.ibm.co.jp/aglets]

The implementation of mobile agents is developed under the Aglets Software Development Kit (ASDK). The name Aglets came from the combination of Agents and applets. ASDK is a Java-based framework for implementing mobile agents [Lange 1998] that is developed by IBM. According to Dasgupta [1998] “ASDK provides an object-oriented programming interface, mechanisms for moving,
data and state information from one machine to another, a platform-independent development and runtime environment, and security mechanisms."

Figure 4.1 shows the classes and interfaces in the Aglet API.

![Diagram of Aglet API classes and interfaces]

**Figure 4.1 Classes and interfaces in the Aglet API [Lange 1998]**

In Figure 4.1:

- An aglet is a mobile Java object. An aglet can visit other aglet-enabled hosts in a computer network.

- An aglet proxy is a representation of an aglet in the server. A proxy works as a shield that protects the aglet from direct access to its public methods.

- A context is an aglet's workplace. A context is a stationary object that provides a means for
maintaining and managing running aglets in a uniform execution environment.

The aglet programming model is event-based. There are three kinds of listeners catching every event in the life cycle of an aglet: clone listener, mobility listener, and persistence listener. Aglets communicate with each other by exchanging messages. An aglet can reply to a message in two ways: synchronous and asynchronous.

Figure 4.2 is a life cycle of an aglet.

Figure 4.2 A life cycle of an aglet [Lange 1998].

An aglet can be created by instantiating directly from an aglet class or cloning from an available aglet. The aglet can then dispatch or retract itself to a remote host. After the aglet completes its tasks, it disposes itself and releases all resources. The Tahiti server in
the ASDK provides an aglet runtime environment with a graphically enhanced interface to create, dispatch, and dispose aglets. (The interface of Tahiti server can be found in Appendix D).

The security model in the aglet system identifies several fundamental principals: the aglet, the aglet owner, the aglet manufacturer, the context, the domain, and the domain authority. There is a set of security services aimed at dealing with the following security threats: remote host threatens agent, agent threatens another agent, unauthorized third parties threaten agent, incoming agent threatens host, unauthorized third parties threaten host, and incoming agent threatens the network. The detail of the services could be found in [Lange 1998].

4.2 Demonstration Description

Based on the ASDK, this research developed a simple simulation of a supply chain. The following is the scenario of this demonstration: At a host server, a user inputs requirements of a product including name, price, and amount. Then, a mobile agent is dispatched to the network together with the requirements and an address list of remote hosts, visiting a remote host, and checking if the remote host keeps products that meet the requirements. This mobile agent then dispatches itself and visits all
remote hosts one by one, and comes back to the original host with the search result.

Figure 4.3 shows the architecture of the demonstration:

Figure 4.3 Architecture of the Mobile Agent Demonstration

There are three parts of the demonstration.
(1) The Graphic User Interface (GUI): This is an interface for receiving the user's requirements and the address list of destination servers. The GUI also shows the results of visiting the remote hosts and the status of the mobile agent.

(2) The stationary agent in the host server: This agent takes care of the host GUI and prepares data for creating and dispatching a mobile agent to the network. This stationary agent also receives and shows the results and status from the mobile agent in the GUI.

(3) The mobile agent: The mobile agent is running on the remote hosts. After arriving at the remote server, the mobile agent checks the local information saved in the remote server, and sees if the user's requirements are satisfied. The mobile agent keeps the result, then moves to the next server, until it goes back to the original host server.

4.3 Generic/Domain-specific design patterns

Based on the scenario of the demonstration, the software design must solve two problems:

(1) How to create a stationary agent and the mobile agent, and how to keep them working together.
(2) How to check the local data in a flexible way.

The following two design patterns have solutions for the two problems:

(1) Master-Slave Pattern [Lange 1998]: a domain-specific pattern.

(2) Strategy Pattern [GOF 1995]: a generic design pattern.

Because the two design patterns cover both generic pattern and domain-specific pattern, the design process of this demonstration can cover both of them and show the effectiveness and applicability of the ADAP model.

As introduced in Chapter Three, the generic design patterns have detailed descriptions of the solution in the length of several pages. This section will only introduce an abbreviated version, which includes key aspects of the design patterns. The detailed descriptions of each design pattern can be found in Gamma [1995] and Grand [1998].

The Master-Slave pattern is a domain-specific design pattern that typically solves the problem of stationary and mobile agents. Lange [1998] defines Master-Slave pattern as schemes whereby a master aglet can delegate a task to a slave aglet.

Figure 4.4 shows the structure and participants of the Master-Slave pattern.
sendMessage-Result Slave;

\[ \text{abstract initializeTask()}
\]
\[ \text{abstract doTask()}
\]

Master

createAglet()

createAglet()

ConcreteSlave

\[ \text{initializeTask()}
\]
\[ \text{doTask()}
\]

Figure 4.4 Structure and Participants of the Master-Slave Pattern

In the solution, the master aglet creates the slave aglet. The Slave aglet initiates the delegated tasks and completes the tasks inside the slave class. After the slave visits all remote hosts, it will send results back to the master.

The Strategy Pattern is a generic design pattern [Gamma 1995] that provides a flexible solution when several algorithms are included. The Strategy pattern defines a family of algorithms, encapsulates each one of them, and makes them interchangeable. Strategy pattern allows the algorithms vary independently from clients that use it. In this demonstration, this pattern provides the solution for checking if remote hosts keep the product
that meets the user's requirements. Figure 4.5 shows the structure and participants of a strategy pattern.

![Diagram of Strategy Pattern](image)

**Figure 4.5 Structure and Participants of Strategy Pattern**

The Strategy pattern defines a family of algorithms for context to reuse. There are several results of using Strategy pattern. The main advantage we use Strategy pattern in this demonstration is its flexibility of varying the algorithms independently by encapsulating algorithms in a separate Strategy class from the context. The trade-off of such adoption is that it may increase the number of objects and more communication overhead between Strategy and Context.
4.4 Evolution of Master-Slave design pattern

4.4.1 Concrete Master-Slave design pattern

According to the ADAP model, the concrete pattern will eliminate ambiguities in the generic pattern.

As a domain specific pattern, the Master-Slave pattern is written for the applications in mobile agents where the context is limited to the situations of mobile agents. In general, there are less ambiguities in domain specific patterns comparing with those in solutions given in a generic design pattern. There are two choices to concretize a Master-Slave pattern:

(1) The Slave class performs same tasks in all visited hosts. In this case, the task of the slave is determined at the design time thus cannot be changed at the run time.

(2) The tasks of the Slave could be different according to the hosts it visited. This is a delegation-based model.

In the design of this demonstration, the mobile agent does the same tasks when travelling to remote hosts. Therefore the domain-specific design patterns are concretized, as the first of the two methods listed above.

The Concrete Master-Slave Pattern in DisCo for this demonstration is as follows:

class Master = {Data}
class Slave = {Data}
Relation(0..1).HasAglet.(\*): Master X Slave
Relation(\*).SendMessageResult.(0..1): Slave X Master

HasAglet(m:Master, d, s:Slave):
  m.HasAglet.s
  \rightarrow m.HasAglet'.s
  s.Data = d
SendMessageResult(m:Master, d, s:Slave):
  \rightarrow s.SendMessageResult.m
  m.Data = d
Retract(m:Master, s:Slave):
  m.HasAglet.s \rightarrow \neg m.HasAglet.s

This concrete pattern extracts the solution for this demonstration from the Master-Slave design pattern. Following the ADAP model, the next step is the specification of this concrete pattern.

4.4.2 Specific Master-Slave design pattern

Based on the concrete Master-Slave pattern, the specification in the ADAP model needs the following modifications:

1. Rename elements in the concrete pattern for this demonstration: The Master class is called SupplyChain. Other changes are listed in Table 4.1 below.

2. Extend classes and methods for the specific application: The master class must be initiated through the onCreation method. It also receives results via the callback method. When the master creates the slave classes, it also takes
two Vectors with data of user's requirements and a list of remote servers.

(3) Add classes or methods for other system requirements: The master class stays in the host server and creates a GUI before it creates a slave aglet. We add a SupplyChainWindow class for the GUI interface.

Table 4.1 shows the transition table between concrete and specific design patterns:

<table>
<thead>
<tr>
<th>Elements in Concrete</th>
<th>Elements in Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>SupplyChain</td>
</tr>
<tr>
<td>Slave</td>
<td>Slave</td>
</tr>
<tr>
<td>InitializeTask()</td>
<td>initializeJob()</td>
</tr>
<tr>
<td>DoTask()</td>
<td>doJob()</td>
</tr>
<tr>
<td>ConcreteSlave</td>
<td>SupplyChainSlave</td>
</tr>
<tr>
<td>InitializeTask()</td>
<td>initializeJob()</td>
</tr>
<tr>
<td>DoTask()</td>
<td>doJob()</td>
</tr>
</tbody>
</table>

Table 4.1 Transition table of Master-Slave pattern.

Figure 4.6 below shows the specific Master-Slave pattern together with other added methods.

This specific Master-Slave pattern includes the solution of creating stationary and mobile agents, and the GUI interface. The above evolution of Master-Slave design pattern indicates that the process in the ADAP model is easy to follow, and the whole process is well-documented. With the available solutions in the Master-Slave design pattern, it is also easy to tell why the demonstration is
designed with four classes - SupplyChain, Slave, SupplyChainSlave, and SupplyChainWindow, as in Figure 4.6.

4.5 Evolution of Strategy design pattern

4.5.1 Concrete Strategy design pattern

The Strategy pattern is a generic pattern. According to the descriptions of Strategy pattern by Gamma [1995], there are many ways to concretize this pattern. Based on the situation of the demonstration, the following are possible ways to concretize the Strategy pattern:
(1) The Context class passes the data as parameter to the Strategy class. Each ConcreteStrategy class fetches those algorithms it needs to use. In this case, the context class does not need to know details of different algorithms.

(2) The Context passes requests from its clients to its Strategy. Or the Strategy class keeps a reference of the Context class. This is a solution that makes the Context and Strategy class more closely coupled.

(3) Making the Strategy class a template if the strategy needs to be selected at compile-time (C++). Or making the strategy class an interface or an abstract class (Java).

In this demonstration, the scenario of our concrete Strategy pattern consists of two classes: the Context class and the Strategy class. The context class passes all data as a parameter to the Strategy class. The Strategy class is an abstract class that keeps common attributes and methods. The ConcreteStrategy classes implement the same methods of the abstract class for different algorithms.

The following is the concrete Strategy pattern in Disco for this demonstration:

class Context = {Data}
class Strategy = {Data}
class concreteStrategy = {Data}

Relation (1): strategy :: Context X Strategy
strategy(c:Context, s:Strategy, d):
  \[ c \cdot \text{strategy} \cdot s \]
  \[ s.Data = d \]

Inherit Strategy as concreteStrategy.
Based on this concrete pattern, the specification will consider adding details for the requirements of this demonstration.

4.5.2 Specific Strategy design pattern

Figure 4.7 shows a specific Strategy pattern in this demonstration. Class names are changed for the application. Attributes and methods are then added to the class.

Figure 4.7 Specific Strategy pattern for the demonstration

Table 4.2 shows the transition table between the concrete level and the specific level.
<table>
<thead>
<tr>
<th>Elements in Concrete</th>
<th>Elements in Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>CheckLocal</td>
</tr>
<tr>
<td>ContextInterface</td>
<td>readlocal</td>
</tr>
<tr>
<td>Strategy</td>
<td>Check</td>
</tr>
<tr>
<td>AlgorithmInterface</td>
<td>issatisfied</td>
</tr>
<tr>
<td>ConcreteStrategy</td>
<td>CheckName</td>
</tr>
<tr>
<td>ConcreteStrategy</td>
<td>CheckPrice</td>
</tr>
<tr>
<td>ConcreteStrategy</td>
<td>CheckAmount</td>
</tr>
</tbody>
</table>

Table 4.2 Transition table of Strategy pattern.

From the above transition table, it is easy to distinguish key elements from the concrete pattern to the specific pattern. In the design of complex systems, it is easier to understand how the problems are solved and the rationale of why each class exists in the design.

By using the solution in the Strategy pattern, the designed system is more flexible.

4.6 Integration

When integrating the specific patterns, we need to consider the relationships and the interface between the two specific patterns. In this demonstration, the Strategy pattern is only related to the Slave class. The Slave class associates with the CheckLocal class. Figure 4.8 shows the integrated design pattern.

4.7 Implementation

The integrated pattern is the final design for this application. Based on this design, the project was implemented in Java language. Table 4.3 lists the
functions for each method in the design. The source code of this demonstration is in Appendix D. Sample run of this demonstration is in Appendix E.

4.8 Other E-commerce Applications

In addition to the mobile agents, the ADAP model can also be adopted in other E-commerce applications such as network security, transaction processing, instant messaging or micro-browser for mobile users.

To conclude this chapter, with domain-specific design patterns, software developers may reuse the solutions by following the ADAP model, and build up new systems promptly. The ADAP model is also a good communication tool for all team members to understand the design during the software development.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Method Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SupplyChain</td>
<td>onCreation</td>
<td>Initializes an aglet and creates a GUI window.</td>
</tr>
<tr>
<td></td>
<td>Callback</td>
<td>Displays the results returning from Slave.</td>
</tr>
<tr>
<td></td>
<td>Go</td>
<td>Creates a mobile agent.</td>
</tr>
<tr>
<td></td>
<td>HandleMessage</td>
<td>Receives and handles messages sent from Slave.</td>
</tr>
<tr>
<td>Slave</td>
<td>initializeJob</td>
<td>Defines interfaces for initialization</td>
</tr>
<tr>
<td></td>
<td>doJob</td>
<td>Defines interfaces for job.</td>
</tr>
<tr>
<td>SupplyChainSlave</td>
<td>initializeJob</td>
<td>Initializes a job</td>
</tr>
<tr>
<td></td>
<td>doJob</td>
<td>Checks local data with the user requirement.</td>
</tr>
<tr>
<td>SupplyChainWindow</td>
<td>go()</td>
<td>Starts SupplyChainSlave</td>
</tr>
<tr>
<td>CheckLocal</td>
<td>readlocal</td>
<td>Reads local data</td>
</tr>
<tr>
<td>Check</td>
<td>issatisfied</td>
<td>Defines interface</td>
</tr>
<tr>
<td></td>
<td>getnoMatch</td>
<td>Gets content in the noMatch vector.</td>
</tr>
<tr>
<td>CheckName</td>
<td>issatisfied</td>
<td>Checks Name</td>
</tr>
<tr>
<td>CheckAmount</td>
<td>issatisfied</td>
<td>Checks Amount</td>
</tr>
<tr>
<td>CheckPrice</td>
<td>issatisfied</td>
<td>Checks Price</td>
</tr>
</tbody>
</table>

Table 4.3 Functions for each method in the design.
Figure 4.8 Integrated pattern for the demonstration.
Chapter Five

Conclusions and Future Directions

5.1 Contribution of the Research

This thesis proposed a component-based model of applying design patterns - ADAP. The model defined the design process from design patterns in the abstract level to the source codes in the implementation level. The purpose of this model is to take advantage of design patterns during the process of software development, and to provide documentation for the maintenance after the software is developed.

This research is done in the following way: Chapter One introduced the concept of reusability and the focus of the research. Chapter Two provided a brief literature review concerning design patterns and their applications to software design. Chapter Three detailed the ADAP model, including system architecture and application processes. Chapter Four demonstrated an example of using the ADAP model in Mobile Agents. This chapter concludes the research with future directions.

Figure 5.1 in below reiterates the proposed ADAP model as follows: three new states of design patterns - concrete, specific, and integrated - are inserted into the traditional two levels: generic/domain-specific and
implementation design patterns. The three added levels in the ADAP model are intended to direct the process of applying design patterns in a more refined and step-by-step way.

1. The concrete design level removes the ambiguities in the original solution from the generic/domain-specific level.

2. The specific pattern level adds design details to the concrete pattern level for the environment under which they apply.

3. The integrated level combines the design patterns from the specific pattern level.

Current industry practices may have applied design patterns with insufficient clarity in the application process. The three added levels in the ADAP model refined the process of applying design patterns. It also provided additional documentation for software designers for their communications.

In addition, the ADAP model suggested the representations for the three added levels.

1. In the concrete level, the ADAP model utilizes the concrete design pattern with DisCo language that is developed for specifications at a high
level of abstraction in the object-oriented environment.

(2) In the specific and integrated levels, the model uses the Unified Modelling Language (UML) that is widely used in software design, particularly in class and object specifications.

![Diagram of ADAP model]

**Figure 5.1 The ADAP model**

Furthermore, this research developed functioning software in mobile agents following the ADAP model. The software development process illustrated each step in the model, which includes the evolution from the generic/domain-specific level to the implemented level. It
also proved that the ADAP model is effective, which meets the goals set for the thesis.

5.2 Future Directions

This research has achieved the set goals specified in Chapter One - to develop a model directing the process of applying design patterns and to provide documentation of the software development. Based on the ADAP model, there are several future directions:

(1) Artificial intelligence (AI) technology, such as expert systems, could provide an automated help assistant to guide a novice software engineer on how to select the appropriate generic/domain-specific design patterns. Presently design patterns are still not frequently used in software design. Tools such as automated assistants developed with AI technologies would be very helpful for most software engineers.

(2) Temporal logic, which is the basis of DisCo language, could be used to prove the correctness of the design, and to fix errors as early as possible in the software development process.

(3) Code generator, which may be developed in C++ or Java after developing the integrated design patterns. Automating the whole process of ADAP
together with a code generator in the software development would improve on software development time.

(4) New applications of the ADAP model in E-commerce, such as network security, transaction processing, mobile agents, instant messaging, online advertisements, micro-browser for mobile users require additional research efforts in the future.

(5) The complexity metrics in software design, such as the level of difficulties in design processes, may be combined with the ADAP model to test the applicability of software reusability.
APPENDIX A

Sample code of the Factory Method in C++:

Definition of the factory methods in MazeGame for creating the maze, room, wall, and door objects [Gamma et al. 1995]:

class MazeGame {
public:
    Maze* CreateMaze();

    // factory methods;
    
    virtual Maze* MakeMaze() const
    { return new Maze; }
    virtual Room* MakeRoom(int n) const
    { return new Room(n); }
    virtual Wall* MakeWall() const
    { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new Door(r1, r2); }

The CreateMaze method:

Maze* MazeGame::CreateMaze() {
    Maze* aMaze = MakeMaze();
    Room* r1 = makeRoom(1);
    Room* r2 = makeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);
    
    aMaze -> AddRoom(r1);
    aMaze -> AddRoom(r2);
    
    r1 -> SetSide(North, MakeWall());
    r1 -> SetSide(East, theDoor());
    r1 -> SetSide(South, MakeWall());
    r1 -> SetSide(West, MakeWall());
    
    r2 -> SetSide(North, MakeWall());
    r2 -> SetSide(East, MakeWall());
    r2 -> SetSide(South, MakeWall());
    r2 -> SetSide(West, theDoor());

    return aMaze;
}
Different games can subclass MazeGame to specialize parts of the maze. MazeGame subclasses can redefine some of all of the factory methods to specify variations in products. For example, a BombedMaze Game can redefine the Room and Wall products to return the bombed varieties:

```cpp
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();
    virtual Wall * MakeWall() const
    { return new BombedWall; }
    virtual Room * MakeRoom(int n) const
    { return new RoomWithABomb(n); }
}
```

An EnchantedMazeGame variant might be defined like this:

```cpp
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();
    virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom(n, CastSpell()); }
}
```

```cpp
virtual Door* MakeDoor(Room* rl, Room* r2) const
{ return new DoorNeedingSpell(rl, r2); }
protected:
    Spell* CastSpell() const;
};
```
APPENDIX B:

Structure of the Builder Pattern and the Template Pattern:

![Diagram of the Builder Pattern]

Figure B.1 Structure of the Builder Pattern

![Diagram of the Template Pattern]

Figure B.2 Structure of the Template Pattern
APPENDIX C: Tahiti server’s interface:

[Image of the Tahiti server's interface]

Tahiti: The Aglet Viewer

[Options and tools]

SupplyChain, SupplyChain Fri May 12 13:12:24 PDT 2000 Starting...

Create: SupplyChain, SupplyChain from file://u/grad/yji/aglets1.0.3/public/

[Image of the Aglets Property File]

Reading aglets property from file:////u/grad/yji/aglets/aglets.properties
Reading ATP property from file:////u/grad/yji/aglets/atp.properties

Licensed materials - Property of IBM
4647394 (C) Copyright IBM Corp. 1996, 1998 All rights reserved.
US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

[IBM Aglets Class Library 1.03 Revision: 8]
Reading property for tahiti from file:///u/grad/yji/aglets/tahiti.properties
[Creating com.ibm.aglets.tahiti.Tahiti]
No audio device
Audio player exit
APPENDIX D:

Source Code of the demonstration

/*========================================================================
* SupplyChain.java
* 
* Katrina
* 3/20/00
* 
* This program is the main program in the application.
* Class SupplyChain is used to
* 1. Ask input of requirements and remote servers from user.
* 2. Dispatch a slave to find those servers who holds inventory
* that meets the requirements of user.
* 3. Retrieve local user information from a remote aglet server
* and display the result.
* 
* Functions in SupplyChain class:
private Vector _resultList = null;
public void onCreate ( Object o)
public boolean handleMessage(Message msg)
protected synchronized void callback(Object arg)
protected synchronized void statusReport(String text)
synchronized void showPartOfResult(URL url)
void go(Vector destinations, Vector requirement)
void getNewFieldValues(Properties p)
* 
* @see SupplyChainWindow
* @see SupplyChainSlave
* 
*******************************************************************/

package SupplyChain;

import com.ibm.aglet.*;
import com.ibm.agletx.patterns.*;
import com.ibm.aglet.util.*;

import java.net.URL;
import java.net.MalformedURLException;
import java.io.IOException;

import java.util.Vector;
import java.util.Properties;
import java.util.Enumeration;


public class SupplyChain extends SampleAglet {

    //-- name of the slave's class
    private final static String SlaveClassName = "SupplyChain.SupplyChainSlave";

    //-- the result of the search
    private Vector _resultList = null;

    public void onCreation (Object o) {
        super.onCreation(o);
        try {
            _msw = new SupplyChainWindow(this);
            updateWindow();
        } catch (Exception e) {
            inError(e.getMessage());
        }
    } // onCreation

    //-- Message handler.
    public boolean handleMessage(Message msg) {
        if (msg.sameKind("status report")) {
            statusReport((String)(msg.getArg()));
            return false;
        } else {
            return super.handleMessage(msg);
        }
    } // handleMessage

    /**
     * The master's callback. The entry point for the returning slave.
     * This method is a part of the Master-Slave usage pattern.
     * @param s the slave.
     * @param arg the result of the slave's work.
     */
    protected synchronized void callback(Object arg) {
        String ni;
        // Slave is back...
        _resultList = (Vector)arg;

        for (int i = 0; i < _resultList.size(); i++) {
            SupplyChainInfo si = (SupplyChainInfo)_resultList.elementAt(i);
            Vector v = si.getResultList();
            String URLString = si.getURLString();
        }
    }
}
for (int j = 0; j < v.size(); j++) {
    ((SupplyChainWindow)_msw).addResultList(
        SupplyChainInfo.getFullPathName(URLString,_resultList.elementAt(j)));
}

((SupplyChainWindow)_msw).addResultList((String)valueOf(_resultList.size()));
}

} // callback

protected synchronized void statusReport(String text) {
    setTheMessage(text);
} // statusReport

//-- display the results found for a specific URL.  
//-- synchronized void showPartOfResult(URL url) {
    for (int i = 0; i < _resultList.size(); i++) {
        SupplyChainInfo si= (SupplyChainInfo)(_resultList.elementAt(i));
        try {
            if (url.toString().equalsIgnoreCase(si.getURLString())) {
                Vector v = si.getResultList();
                for (int j = 0; j < v.size(); j++) {
                    ((SupplyChainWindow)_msw).addResultList(SupplyChainInfo.getFullPathName(url.toString(),
                        (String)(v.elementAt(j))));
                }
            } catch (Exception e) {
                // not yet implemented
            }
        }
    }
} // showPartOfResult

/**
 * Dispatch a slave.
 * @param destinations: Vector of destination URLs.
 * @param requirement: Vector including user requirement.
 */
void go(Vector destinations, Vector requirement) {
    super.go((URL)(destinations.firstElement()));
    try {
        Slave.create(null, SlaveClassName, getAgletContext(),
try {
    for (Enumeration e = p.propertyNames(); e.hasMoreElements();)
    {
        String key = (String)e.nextElement();
        getAgletContext().setProperty(key, p.getProperty(key));
        getAgletContext().multicastMessage(new Message("updateWindow"));
    }
} catch (AgletException ae) {
    inError(ae.getMessage());
} // getNewFieldValues
*/

@ =: SupplyChainSlave.java
*

* Katrina
* 3/20/00
* The SupplyChainSlave class is a aglet for the SupplyChain.
* This slave class is part in the Master-Slave pattern
* Member functions:
    private Vector resultList = null;
    private Vector requirement = null;
    protected void initializeJob()
    protected void doJob() throws AgletException
    private void reportMaster(String msg)
*

* @see SupplyChain
*********************************************************************
package SupplyChain;

import com.ibm.aglet.*;
import com.ibm.agletx.patterns.*;
import java.util.Enumeration;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.Date;
import java.net.URL;
import java.net.MalformedURLException;
import java.io.*;
import java.lang.System;

public class SupplyChainSlave extends Slave {
    private Vector resultList = null;
    private Vector requirement = null;
    private CheckLocal _check = null;

    protected void initializeJob () {
        RESULT = new Vector();
    } // initializeJob

    protected void doJob() throws AgletException {
        String filename;

        Vector requirement = new Vector(); // requirement from master
        resultList = new Vector(); // fileList for RESULT
        requirement = (Vector)ARGUMENT;

        //-- get context of this slave
        AgletContext ctx = getAgletContext();

        String hostname;
        if (ctx.getHostingURL() == null) {
            hostname = "Unknow";
        } else {
            hostname = ctx.getHostingURL().getHost().toString();
        }
        resultList.addElement( "This is from the server: " + hostname);
        resultList.addElement( (new Date()).toString());

        // check existence of the local file
        File file;
        String inventname = "/u/grad/yji/Inventory";
if (hostname.equals("gemini.csci.csusb.edu")) {
    inventname = "/u/grad/yji/Invent-gemini";
}
if ((file = new File(inventname)) == null) {
    throw new AgletException("AgletException:" + "Null file object error");
}
if (!file.exists()) {
    throw new AgletException("Try to access non-existing file");
}
_check = new CheckLocal(requirement, inventname);
for (int i = 0; i < _check.getResult().size(); i++)
    resultList.addElement(_check.getResult().elementAt(i));
((Vector)RESULT).addElement(new SupplyChainInfo(getAgletContext().getHostingURL(), resultList));
reportMaster("Completed search");
} // doJob

//-- report the Master
//--
private void reportMaster(String msg) {
    AgletContext ac;
    try {
        ac = getAgletContext();
        Messenger.create(ac,
            new URL(getOrigin()),
            getMaster(),
            new Message("status report",
                ac.getHostingURL().toString() + ":" + msg));
    } catch (IOException ae) {
        //-- we give up this report.
    } catch (AgletException ae) {
        //-- we give up this report.
    }
} // reportMaster
} // SupplyChainSlave


/******************************************************************************
 * @(#)SupplyChainInfo.java
 * **********************************************************************/
package SupplyChain;

import java.util.Vector;
import java.net.URL;
import java.io.*;

/**
 * Class SupplyChainInfo abstracts the information returned from the
 * mobile searcher to its master.
 * @see SupplyChain
 */

final class SupplyChainInfo implements Serializable {
    private String _url;
    private Vector _resultList;

    SupplyChainInfo(URL url, Vector v) {
        _url = url.toString();
        _resultList = v;
    } //SupplyChainInfo

    String getURLString() {
        return _url;
    } // getURLString

    Vector getResultList() {
        return _resultList;
    } //_getResultList

    private static final String DELIMITER = "::";

    static String getFullPathName(String URLString, String fileName) {
        return URLString + DELIMITER + fileName;
    } // getFullPathName

    static String extractPathName(String fullPathName) {
        int i = fullPathName.indexOf(DELIMITER);
        if (i > -1)
            return fullPathName.substring(i+2);
        else
            return fullPathName;
    } // extractPathName

    static String extractHostURL(String fullPathName) {
        int i = fullPathName.indexOf(DELIMITER);
        if (i > -1)
            return fullPathName.substring(0,i);
        else
            return fullPathName;
    } // extractHostURL
}
SupplyChainWindow.java

Katrina Ji
3/19/00

SupplyChainWindow.java is a GUI for asking input from user
and display result from other servers. The slave will
return with the information
to be displayed by the master class.

Attributes:
private List _locationList = new List(5, true);
private List _resultList = new List(8, false);

Member functions:
public SupplyChainWindow(SupplyChain aglet)
private void makeMainPanel() throws AgletException
void addResultList(String s)
public void clearResult()
private Panel makeLocalButtonPanel(Button b1, Button b2, Button b3)
protected void go()
private synchronized void addLocation()
protected boolean popUpHandleButton (Button button)
private boolean handleList(List list)

package SupplyChain;

import com.ibm.aglet.*;
import com.ibm.agletx.patterns.NetUtils;
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Choice;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Event;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.Frame;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Label;
import java.awt.List;
import java.awt.Panel;
import java.awt.TextField;
import java.awt.TextArea;
import java.awt.Color;
import java.awt.Insets;
import java.net.URL;
import java.net.MalformedURLException;
import java.util.Vector;
import java.util.Properties;
import java.util.Enumeration;

public class SupplyChainWindow extends SampleWindow {
    private static final String TITLE = "Welcome to SupplyChain GUI! ";
    private static final String TITLE1 = "Please input your requirement: ";
    private static final String TITLE2 = "Please input addresses: ";
    private static final String NO_DESTINATIONS_MSG = "Please input at least one destination URL.";
    private static final String NO_NAME_MSG = "Please input Inventory Name!";
    private static final String NO_LOWER_LIMITATION_MSG = "Please input lowest price!";
    private static final String WRONG_LOW_LIMITATION_MSG = "Please check lowest price!";
    private static final String NO_UPPER_LIMITATION_MSG = "Please input highest limitation!";
    private static final String WRONG_HIGH_LIMITATION_MSG = "Please check highest price!";
    private static final String NO_AMOUNT_MSG = "Please input amount!";
    private static final String WRONG_AMOUNT_MSG = "Please input an integer in amount!";
    private static final String NO_UPPER_LIMITATION_MSG = "Please input highest limitation!";
    private static final String WRONG_RANGE_MSG = "Please make sure your lowest price is smaller than the highest price!";

    private PopUpMessageWindow _noDestinationsWindow = null;
    private PopUpMessageWindow _noInputName = null;
    private PopUpMessageWindow _noLowerLimit = null;
    private PopUpMessageWindow _wrongLowLimit = null;
    private PopUpMessageWindow _noUpperLimit = null;
    private PopUpMessageWindow _wrongHighLimit = null;
    private PopUpMessageWindow _noAmount = null;
    private PopUpMessageWindow _wrongAmount = null;
    private PopUpMessageWindow _wrongRange = null;

    //-- Constructs the dialog window. //
    public SupplyChainWindow(SupplyChain aglet) throws AgletException {
        super(aglet);
    }
}
_noDestinationsWindow = new PopUpMessageWindow(this, "DESTINATIONS ARE MISSING",
NO_DESTINATIONS_MSG);

_noInputName = new PopUpMessageWindow(this, "INVENTORY NAME IS MISSING",
NO_NAME_MSG);

_noLowerLimit = new PopUpMessageWindow(this, "LOWER LIMITATION IS MISSING",
NO_LOWER_LIMITATION_MSG);

_wrongLowLimit = new PopUpMessageWindow(this, "LOW LIMITATION IS WRONG",
WRONG_LOW_LIMITATION_MSG);

_noUpperLimit = new PopUpMessageWindow(this, "UPPER LIMITATION IS MISSING",
NO_UPPER_LIMITATION_MSG);

_wrongHighLimit = new PopUpMessageWindow(this, "HIGH LIMITATION IS MISSING",
WRONG_HIGH_LIMITATION_MSG);

_noAmount = new PopUpMessageWindow(this, "AMOUNT IS MISSING",
NO_AMOUNT_MSG);

_wrongAmount = new PopUpMessageWindow(this, "AMOUNT IS WRONG",
WRONG_AMOUNT_MSG);

_wrongRange = new PopUpMessageWindow(this, "Price IS WRONG",
WRONG_RANGE_MSG);

makeMainPanel(); displayFrame(this);

//-- Main panel
private List _locationList = new List(5, true);
private List _resultList = new List(10, false);

private void makeMainPanel() throws AgletException {
 Component comp;
// title
constraints.anchor = GridBagConstraints.CENTER;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.BOTH;
constraints.insets = new Insets(0,0,5,0);
constraints.weightx = 1.0;
constraints.weighty = 2.0;
comp = new Label(TITLE);
comp.setFont(new Font(getFont().getName(),
Font.BOLD,
getFont().getSize()-f2));
layout.setConstraints(comp, constraints);
add(comp);

// title1
constraints.anchor = GridBagConstraints.CENTER;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.BOTH;
constraints.weightx = 1.0;
constraints.weighty = 2.0;
comp = new Label(TITLE1);
comp.setFont(new Font(getFont().getName(),
Font.BOLD,
getFont().getSize()+1));
layout.setConstraints(comp, constraints);
add(comp);

// file name panel
addLabeledComponent("Stock Name :", _name);
addLabeledComponent("Lowest Price preferred :",
_lowerlimitation);
addLabeledComponent("Highest Price preferred :",
_upperlimitation);
addLabeledComponent("Amount needed:", _amount);

// title2
constraints.anchor = GridBagConstraints.CENTER;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.BOTH;
constraints.weightx = 1.0;
constraints.weighty = 2.0;
comp = new Label(TITLE2);
comp.setFont(new Font(getFont().getName(),
Font.BOLD,
getFont().getSize()+1));
//layout.setConstraints(comp, constraints);
add(comp);

// add button for host address
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.BOTH;
constraints.weighty = 1.0;
comp = makeLocalButtonPanel(_addLocation, _removeLocation,
                        _showFiles);
layout.setConstraints(comp, constraints);
add(comp);

// information settings
// addLabeledComponent(URLLabel, _URLString);
// initURLFields(getHotlist(_aglet));  // hotlist & URL field
// addLabeledComponent(hotlistLabel, _hotlist);

// show address book
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.insets = new Insets(0,0,1,0);
constraints.weightx = 1.0;
comp = _addressChooser;
layout.setConstraints(comp, constraints);
add(comp);

// show host location
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.BOTH;
comp = _locationList;
layout.setConstraints(comp, constraints);
add(comp);

//Label = Result
//constraints.anchor = GridBagConstraints.CENTER;
//constraints.gridwidth = GridBagConstraints.REMAINDER;
//constraints.fill = GridBagConstraints.BOTH;
comp = new Label(" Result ");
//layout.setConstraints(comp, constraints);
add(comp);

// add button to send out aglets
constraints.anchor = GridBagConstraints.CENTER;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.BOTH;
constraints.weightx = 1.0;
comp = makeMainButtonPanel();
layout.setConstraints(comp, constraints);
add(comp);

// panel for showing the result
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.BOTH;
constraints.weighty = 1.0;
comp = _resultList;
_resultList.addItem("The result will be listed there!");
layout.setConstraints(comp, constraints);
add(comp);
// area for error messages
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.fill = GridBagConstraints.BOTH;
constraints.weighty = 1.0;
initMessagePanel();
comp = _msgLine;
layout.setConstraints(comp, constraints);
add(comp);
} // makeMainPanel

//-- Result panel

void addResultList(String s) {
    boolean found = false;
    int count = _resultList.countItems();
    String item;
    for (int i = 0; i < count; i++) {
        item = _resultList.getItem(i);
        if (item.equals(s)) {
            return;
        }
    }
    _resultList.addItem(s);
    return;
} // addResultList

public void clearResult() {
    _resultList.clear();
}

//-- Location panel
private Button _addLocation = new Button("Add");
private Button _removeLocation = new Button("Remove");
private Button _showFiles = new Button("Show");

private Panel makeLocalButtonPanel(Button bl, Button b2, Button b3) {
    Panel p = new Panel();
    p.setLayout(new FlowLayout(FlowLayout.RIGHT));
    p.add(bl);
    p.add(b2);
    p.add(b3);
    return p;
} // makeLocalButtonPanel

//-- The call back methods

116
protected void go() {
    String item;
    Vector requirement = new Vector();
    Vector itin;

    // Check valid input
    if (!CheckInput()) return;
    String filename = _name.getText().trim();
    String lowlimitation = _lowerlimitation.getText().trim();
    String upperlimitation = _upperlimitation.getText().trim();
    String amount = _amount.getText().trim();

    // Prepare requirement for slave
    requirement.addElement(new String(filename));
    requirement.addElement(new String(lowlimitation));
    requirement.addElement(new String(upperlimitation));
    requirement.addElement(new String(amount));

    // Prepare destination for slaves
    itin = new Vector();
    int size = _locationList.countItems();
    try {
        for (int i = 0; i < size; i++) {
            item = _locationList.getItem(i).trim();
            itin.addElement(new URL(item));
        }
        ((SupplyChain)_aglet).go(itin, requirement);
    } catch (MalformedURLException e) {
        _malFormedURLWindow.popup(this);
    }
} // go

// check invalid input
private boolean CheckInput() {
    // check invalid input
    String filename = _name.getText().trim();
    String lowlimitation = _lowerlimitation.getText().trim();
    String upperlimitation = _upperlimitation.getText().trim();
    String amount = _amount.getText().trim();
    float inputlow = 0;
    float inputhigh = 0;

    if (filename.equals("")) {
        _noInputName.popup(this);
        return false;
    }
    if (lowlimitation.equals("")) {
        _noInputLowLimitation.popup(this);
    }
_noLowerLimit.popup(this);
return false;
}
if (upperlimitation.equals('')) {
    _noUpperLimit.popup(this);
    return false;
}
if (amount.equals('')) {
    _noAmount.popup(this);
    return false;
}
try {
    inputlow = Float.valueOf(lowlimitation).floatValue();
} catch (NumberFormatException e) {
    _wrongLowLimit.popup(this);
    return false;
}
try {
    inputhigh = Float.valueOf(upperlimitation).floatValue();
} catch (NumberFormatException e) {
    _wrongHighLimit.popup(this);
    return false;
}
if (inputlow >= inputhigh) {
    _wrongRange.popup(this);
    return false;
}
try {
    int inputamount = Integer.parseInt(amount);
} catch (NumberFormatException e) {
    _wrongAmount.popup(this);
    return false;
}
// check invalid destination
int size = _locationList.countItems();
if (size == 0) {
    _noDestinationsWindow.popup(this);
    return false;
}
return true;

// CheckInput

//-- add location to the list
private synchronized void addLocation() {
    int count;
    String newLocation;
    URL item, newURL;
    int port = -1;

    // new URL string
newLocation = _addressChooser.getAddress();

try {
    newURL = new URL(newLocation.toString());
} catch (MalformedURLException e) {
    _malFormedURLWindow.popup(this);
    return;
}

count = _locationList.countItems();
for (int i = 0; i < count; i++) {
    try {
        item = new URL(_locationList.getItem(i));
    } catch (MalformedURLException e) {
        return;
    }

    if (NetUtils.sameURL(newURL, item)) {  // -- compare in the
        // level of URLs.
        return;
    }
}

// if there is no matched URL, add a new item
_locationList.addItem(newURL.toString().trim());  // addLocation

// remove selected item from the list
private void removeLocation() {
    for (int i = _locationList.countItems(); i >= 0; i--) {
        if (_locationList.isSelected(i)) {
            _locationList.delItem(i);
        }
    }  // removeLocation

//-- show files found for the selected host in the list
private void showFiles() {
    URL url = null;
    boolean found = false;
    for (int i = _locationList.countItems(); i >= 0; i--) {
        if (_locationList.isSelected(i)) {
            try {
                url = new URL(_locationList.getItem(i));
            } catch (MalformedURLException ae) {
                continue;
            }
            if (!found) {  // -- first time...
                found=true;
                clearResult();
            }
        }
    }
}
```java
((SupplyChain)_aglet).showPartOfResult(url);
}
} // showFiles

//-- Event handler methods

//-- The event handler.
//--
public boolean handleEvent(Event event) {
    if (event.target instanceof List)
        return handleList((List)event.target);
    else
        return super.handleEvent(event);
} // handleEvent

//-- Handles button events.
//--
protected boolean handleButton(Button button) {
    if (button == _addLocation) {
        addLocation();
    } else if (button == _removeLocation) {
        removeLocation();
    } else if (button == _showFiles) {
        showFiles();
    } else
        return super.handleButton(button);
    return true;
} // handleButton

protected boolean popUpHandleButton (Button button) {
    if (button == _malFormedURLWindow.getButton(PopUpMessageWindow.OKAY) &&
        "Okay".equals(button.getLabel()) ) {
        _malFormedURLWindow.setVisible(false);
        return true;
    }
    if (button ==
        _noDestinationsWindow.getButton(PopUpMessageWindow.OKAY) &&
        "Okay".equals(button.getLabel()) ) {
        _noDestinationsWindow.setVisible(false);
        return true;
    }
    if (button == _noInputName.getButton(PopUpMessageWindow.OKAY) &&
        "Okay".equals(button.getLabel()) ) {
        _noInputName.setVisible(false);
        return true;
    }
```

120
if (button == _noLowerLimit.getButton(PopupMessageWindow.OKAY))
   &&
   "Okay".equals(button.getLabel()) ) {
   _noLowerLimit.setVisible(false);
   return true;
}
if (button == _wrongLowLimit.getButton(PopupMessageWindow.OKAY))
   &&
   "Okay".equals(button.getLabel()) ) {
   _wrongLowLimit.setVisible(false);
   return true;
}
if (button == _noUpperLimit.getButton(PopupMessageWindow.OKAY))
   &&
   "Okay".equals(button.getLabel()) ) {
   _noUpperLimit.setVisible(false);
   return true;
}
if (button == _wrongHighLimit.getButton(PopupMessageWindow.OKAY))
   &&
   "Okay".equals(button.getLabel()) ) {
   _wrongHighLimit.setVisible(false);
   return true;
}
if (button == _wrongRange.getButton(PopupMessageWindow.OKAY))
   &&
   "Okay".equals(button.getLabel()) ) {
   _wrongRange.setVisible(false);
   return true;
}
if (button == _noAmount.getButton(PopupMessageWindow.OKAY))
   &&
   "Okay".equals(button.getLabel()) ) {
   _noAmount.setVisible(false);
   return true;
}
return false; //-- should not reach here.
} // popupHandleButton

--- Handles List field events.

private boolean handleList(List list) {
   if (list == _resultList) {
      String str = list.getSelectedItem();
      Properties p = new Properties();
      p.put("filename", SupplyChainInfo.extractPathName(str));
      p.put("location", SupplyChainInfo.extractHostURL(str));
      ((SupplyChain) _aglet).getNewFieldValues(p);
   }
}
CheckLocal.java
Katrina
4/6/00
Client class in Strategy pattern
Read local data and check if the data meets
user's requirement

package SupplyChain;

import java.util.Vector;
import java.io.*;

final class CheckLocal implements Serializable{

    private static final Vector noMatch = new Vector();
    private static Check firstcheck ;

    private static CheckName fname = new CheckName();
    private static CheckPrice fprice = new CheckPrice();
    private static CheckAmount famount = new CheckAmount();

    public CheckLocal(Vector requirement, String str) {
        Vector localdata = new Vector();
        readlocal(str, localdata);

        firstcheck = fname;
        firstcheck.ifsatisfied(requirement, localdata);

        firstcheck = fprice;
        firstcheck.ifsatisfied(requirement, localdata);

        firstcheck = famount;
        firstcheck.ifsatisfied(requirement, localdata);
    }

    protected Vector getResult(){
        Vector result = new Vector();

        if (firstcheck.getnoMatch().size() == 0){
            result.addElement( "++++ Requirement satisfied!");
        }
    }
}
else {
    for (int i = 0; i < firstcheck.getNoMatch().size(); i++)
        result.addElement(firstcheck.getNoMatch().elementAt(i).toString());
}
return result;
}

// read local data
private static void readLocal (String filename, Vector localdata) {
    File file;
    try {
        file = new File(filename);
        FileInputStream inFile = new FileInputStream(file);
        BufferedReader indata = new BufferedReader(new InputStreamReader(inFile));

        int inBytes = inFile.available();
        byte inBuf[] = new byte[inBytes];
        int bytesRead = inFile.read(inBuf, 0, inBytes);
        
        /*
         * read info from local file
         * FileReader inFile = new FileReader(file);
         * BufferedReader indata = new BufferedReader(inFile);
         * s = indata.readLine();
         */

        String str = new String(inBuf, 0);
        String s = new String(inBuf, 0);

        int firstpos = 0;
        for (int pos = str.indexOf(' '); pos < str.length() && pos != -1;
            pos = str.indexOf(' ', firstpos)) {
            s = str.substring(firstpos, pos).trim();
            localdata.addElement(s);
            firstpos = pos + 1;
            if (firstpos == str.length()) break;
        }
        firstpos = str.lastIndexOf(' ');
        int pos = str.indexOf("+");
        s = str.substring(firstpos, pos).trim();
        localdata.addElement(s);

        inFile.close();
    }
}

} catch (Exception e) {
    // error
}  // catch
This class is an abstractStrategy class

package SupplyChain;

import java.util.*;
import java.util.Vector;

public abstract class Check{
    protected final static Vector noMatch = new Vector();
    // Return boolean after check ifSatisfied.
    abstract public boolean ifSatisfied(Vector requirement, Vector Idata);
    
    public Vector getNoMatch()
    {
        return noMatch;
    }
}

// class Check

// Check if two name are identical
package SupplyChain;
import java.util.Vector;

public class CheckName extends Check{
    
    public boolean ifSatisfied(Vector req, Vector ldata)
    {
        if (req.elementAt(0).equals(ldata.elementAt(0)))
    }
return true;
noMatch.addElement("---- Name not Match!");
return false;

} // ifsatisfied

} // CheckName

/***************************************************************************
CheckPrice.java
Katrina
4/5/00
This class inheritant from Check class. It check if the price satisfied
the requirement.
***************************************************************************/

package SupplyChain;

import java.util.Vector;

// Check if the lowest price meets.
public class CheckPrice extends Check{

   public boolean ifsatisfied(Vector req, Vector ldata){
      if ( (Float.valueOf(ldata.elementAt(1).toString()).floatValue()
        < Float.valueOf(req.elementAt(1).toString()).floatValue() )
        &&
        (Float.valueOf(ldata.elementAt(2).toString()).floatValue() >
        Float.valueOf(req.elementAt(2).toString()).floatValue() ) )
      return true;
      noMatch.addElement("---- Price not satisfied!");
      return false;
   }

} // CheckPrice

/***************************************************************************/
**
CheckAmount.java
Katrina
4/5/00
***************************************************************************/

package SupplyChain;

import java.util.Vector;

// Check if the amount match.
public class CheckAmount extends Check(
    public boolean ifSatisfied(Vector req, Vector Idata){
        if ( Integer.valueOf(req.elementAt(3).toString()).intValue() < 
            Integer.valueOf(Idata.elementAt(3).toString()).intValue() )
            return true;
        noMatch.addElement(" Not enough in Stock!");
        return false;
    } // ifSatisfied
} // CheckAmount
APPENDIX E: Sample Run of the demonstration.

SupplyChain Interface

Welcome to SupplyChain GUI!

Please input your requirement:

- Stock Name:
- Lowest Price preferred:
- Highest Price preferred:
- Amount needed:

Please input addresses:

Address Book | Address:
--- | ---
[Input Address]

--- Result ---

Get Qualified Resources | Quit

--- Not enough in Stock ---

Nodile Agent is Ready

From: [Input Address]

going to: [Input Address]

Completed Search

SupplyChainSlave returned.
Bibliography


<http://www.serc.nl/people/florijn/work/patterns.html> 1999


[GELO] GELO Project 

[Gamma 1995] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. 1995


[Lange 1998] Danny B. Lange, Mitsuru Oshima Programming and Deploying Java Mobil Agents with Aglets Addison-Wesley 1998


