THE COLLABORATION OF PATTERNS – A CASE STUDY

Ku-Yaw Chang[1], Lih-Shyang Chen[2], and I-Ning Chang[3]

Department of Electrical Engineering, National Cheng-Kung University
No. 1, University Rd., Tainan 701, Taiwan, R.O.C.
Email:{canseco[1], mol[3]}@mirac.ee.ncku.edu.tw, chens@mail.ncku.edu.tw[2]

ABSTRACT

Patterns are a promising technique for achieving widespread reuse of software architectures. They document existing, well-proven design experience and help you find appropriate solutions to design problems. Patterns, however, are usually interwoven with each other in a real-world software system. It is challenging and also needs some experience to compose patterns to a large structure in a meaningful way. In this paper, we take a medical imaging application as an example to describe how we combine a collection of patterns and organize them into a large software architecture. Instead of repeating each individual pattern in detail, we focus on how its constituent patterns are connected with each other. We also address some lessons learned from applying a pattern-based strategy to developing such a large application system.

1.INTRODUCTION

Patterns[1][2] are a promising technique for achieving widespread reuse of software architectures. Each pattern deals with a specific, recurring problem in the design or implementation of a software system. They document existing, well-proven design experience and help you find appropriate solutions to design problems. However, patterns do not exist in isolation. The architecture of a real-world software system usually consists of a collection of patterns – there are many interdependencies between them. Generally speaking, when we apply more patterns in a software system, the resulting design becomes more difficult to understand and to implement. To use patterns effectively is not a mechanical task. It needs some experience to compose them to a large structure in a meaningful way.

At National Cheng-Kung University, we have developed an interactive system called Discover[3] for scientific visualization. Although Discover is suitable for many areas, we have concentrated on medical imaging analysis and generation – one of the most rapidly growing applications for scientific visualization. Although our domain algorithms are very important, we believe that a well-structured system architecture is also a key element of continuous growth of our system. In fact, the system architecture of Discover consists of patterns. Over the past years, more and more patterns are integrated into our system and outdated patterns are removed if they are no longer used. This paper describes what patterns we use and how they connected together to complement each other. We also address some lessons learned from applying a pattern-based strategy to developing such a system.

The remainder of this paper is organized as follows: Section 2 gives a brief review of pattern categories and their relationships; Section 3 describes how we ties individual patterns together in Discover; Section 4 summarizes our experiences (both positive and negative) applying a pattern-based strategy to the development of Discover; and Section 5 presents our concluding remarks.

2.BACKGROUND

The idea of a design pattern originated with Christopher Alexander, an architect who documented a pattern language for the planning of towns and the construction of buildings within towns[4]. In 1987, several leading-edge software developers rediscovered Alexander's work and applied it to documenting design decisions in developing software. It was not until 1994 that design patterns entered the mainstream of the object-oriented software development community. Since then, there has been an exponential growth in the publication of design pattern literature. However, as the number of published patterns increases, it becomes another challenge for software developers to select a suitable design pattern or pattern language to meet their specific set of design concerns. If software developers must understand every pattern in detail to find the ones they need, the pattern system as a whole is helpless, even if its constituent patterns are useful.

2.1 Pattern Categories

One possible solution to the above problem is to categorize patterns by different criteria[5], such as granularity or level[2], purpose or scope[1], functionality or subject[6], and metalevel[7]. For example, according to different ranges of scale (granularity) and abstraction (level), patterns can be classified into three categories:

(1) *Architectural patterns*: Architectural patterns represent the highest-level patterns. They express the fundamental, system-wide structure for software systems. The selection of an architectural pattern is the first step when designing the architecture of a software system. However, a particular architectural pattern, or a combination of several, is not a complete software architecture. A detailed structural

framework must be further specified and refined.

(2) *Design patterns*: Design patterns are medium-level patterns and smaller in scale than architectural patterns. A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. Design patterns are not language specific and can be implemented in a variety of languages.

(3) *Idioms*: Idioms are the lowest-level and language-specific patterns. They describe how to implement particular aspects of components or the relationships between them using the features of the given language. In other words, they address both design and implementation issues.

In the above classification scheme, each category consists of patterns having a similar range of scale or abstraction. They are related to important software development activities. With the help of classification schemes like the above example, designers are able to search the currently existing catalogues for patterns in a more efficient way.

2.2 Pattern Relationships

Most well-structured systems are full of patterns that are tied together effectively. The relationships between patterns can put into the following categories[8]:

(1) *Refinement*: Applying a pattern solves a particular problem, but its application may raise new problems. Some of these can be solved by other patterns, which are usually smaller in scale. In other words, the latter pattern(smaller) refines the former(larger). Christopher Alexander puts this in somewhat idealistic terms: 'Each pattern depends on the smaller patterns it contains and on the larger patterns in which it is contained'[9]. For example, in the MVC (Model-View-Controller) pattern[2], the consistency between the components must be maintained: whenever the state of the model changes, all its dependent views and controllers must be updated. The Publisher-Subscriber pattern[2] can help us to solve this problem – the model takes the role of the publisher, while views and controllers play the roles of subscribers.

(2) *Variants*: A pattern may be a variant of another. In general, a pattern and its variants provide solutions to similar problems, which usually vary only in some of the forces involved. For example, the Document-View pattern [2] [10] is a variant of the MVC pattern. Its view component combines the view and controller functionality of the MVC pattern. The Document-View pattern is suitable for GUI (graphical user interface) platforms where window display and event handling are closely interwoven.

(3) *Combination*: Several patterns at the same level of abstraction can combine together to form a more complex structure. This happens when your original problem includes many forces that cannot be handled by a single pattern. This kind of relationship occurs often in practical complex systems.

3.SYSTEM ARCHITECTURE OF DISCOVER

3.1 The Architectural Pattern

3.1.1 Document-View-Presentation Pattern

The fundamental system-wide architecture of Discover is built on the Document-View-Presentation pattern (DVP) [11], which is a variant of the Document-View pattern[2] [10]. The DVP pattern is especially suitable for interactive application systems with computationally expensive rendering algorithms, such as computer graphics or image processing systems. This pattern divides an interactive system into the following three components.

(1) *Document*: The document component contains data and core functionality manipulating the data. It is independent of any specific input and output methods. The document component contains data and core functionality manipulating the data. It is independent of any specific input and output methods.

(2) *View*: View components contain the rendering algorithms of the application and maintain their respective rendering results. They are hidden behind the presentation components and do not interact with end users directly. They accept requests and fulfill the corresponding services by calling the core functionality provided by the document or its own rendering algorithm. They also obtain data from the document and use different algorithms to render (represent) the data. There can be multiple views of a document.

(3) *Presentation*: Presentation: Presentation components are the representatives of their views for input and output. They receive user events or messages from other components, and turn them into service requests. They do not implement these services directly. Instead, they forward these requests to their associated views. In addition, they also obtain the rendering result from their views and output the rendering result to different devices, such as the screen or the disk. Each presentation component could be for input only, output only, or both input and output simultaneously. They can be visible or invisible to users. The user can directly interact with the system through those visible presentation components.
The relationships between these components are illustrated in Fig. 1. Note that we adopt Unified Modeling Language (UML) [12] notation for all the figures in this paper.

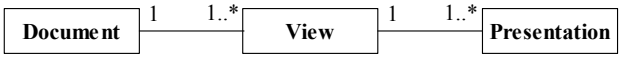

Fig. 1 The class diagram of the DVP pattern.

3.1.2 Publisher-Subscriber Pattern

The Publisher-Subscriber[2] pattern helps to keep the state of cooperating components synchronized. One dedicated component embodies the role of the publisher. All compo-

nents dependent on changes in the publisher are its subscribers. Whenever the publisher changes state, it sends a notification to all its subscribers. This pattern is also known as the Observer pattern[1].

In the DVP architectural pattern, when the data of the document changes, the document must notify all its dependent view components. Similarly, a view component needs to notify all its dependent presentation components whenever its rendering result changes. These two one-way change-propagation mechanisms can be specified with help of the Publisher-Subscriber pattern: a) the document plays the role of the publisher while its views are subscribers; b) the view plays the role of the publisher while the presentation components are subscribers. Note that the view plays both the roles – publisher and subscriber at the same time. Fig. 2 shows the collaboration between these two patterns.

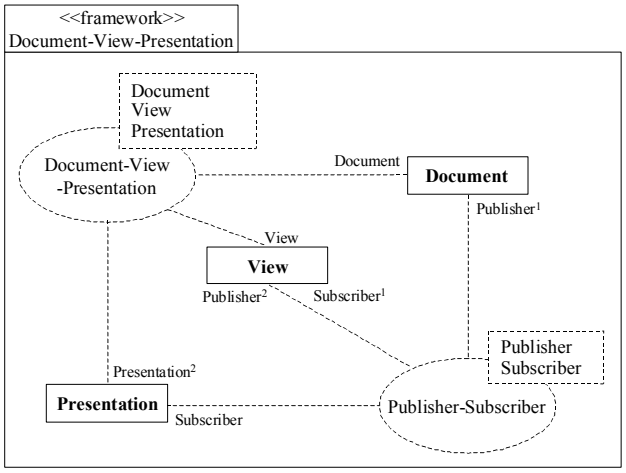

Fig. 2 The DVP pattern can be refined by the Publisher-Subscriber pattern.

In implementation, the change-propagation mechanism between the document and its views can be provided by the application framework. Please see Section 4 for more details.

3.2 Design Patterns

We further unfold the details of the above architectural skeleton with the following design patterns.

(1) *Command Processor Pattern*[2]: The command processor pattern builds on the Command design pattern in [1]. Its main idea is to separate a service request from its execution by encapsulating these requests into objects, known as command objects. A command processor component manages command objects, schedules their execution, and provides additional services such as the storing of command objects for later undo or redo.

As mentioned in Section 3.1, the views in the DVP pattern are in charge of both input and output simultaneously. As far as the input is concerned, views accept requests and fulfill the corresponding services by calling the core functions provided by the document. Such relationships between the document and its views can be refined by applying the Command Processor pattern[2]. As shown in Fig. 3, the document plays the role of the supplier and the view plays the role of the client, which sends requests to the controller. The more details can be obtained by further unfolding the Command Processor, as illustrated in Fig. 4. Instead of direct calling functions provided by the document to fulfill a request, the view component forwards the request to the controller. The controller creates a new concrete command object and then transfers the new command object to the processor for execution and further handling. Each command object delegates the execution of its task to the document, which provides most of the required functionality.

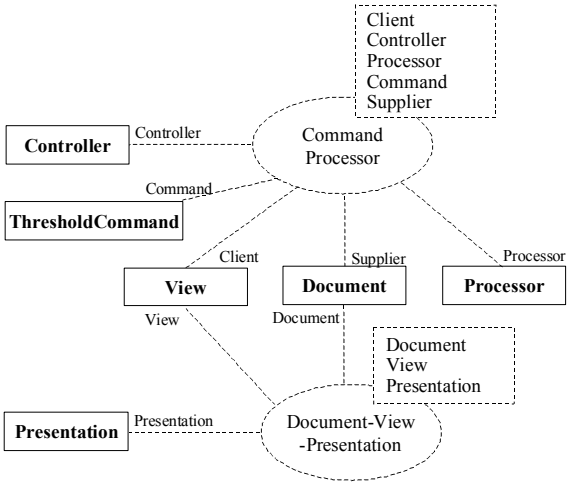

Fig. 3 The collaboration between the Command Processor pattern and the DVP pattern.

(2) *Singleton Pattern*[1]: This pattern ensures a class only has one instance and provides a global point of access to it. In order to have strict control over command objects and monitor how and when views deliver requests to the con-

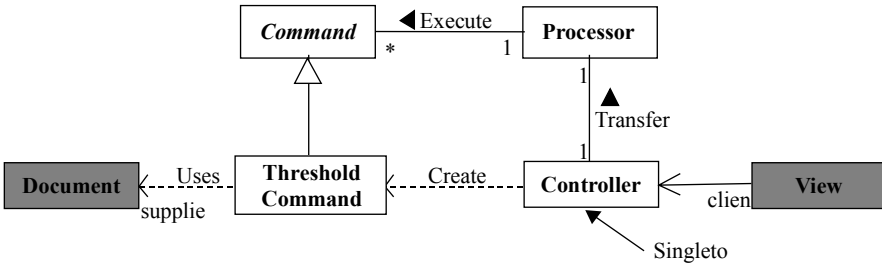

Fig 4. The class diagram of the Command Processor pattern with their collaborators.

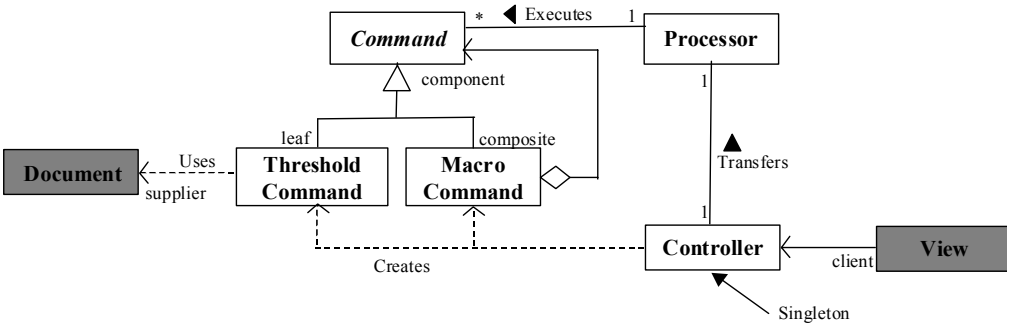

Fig. 5 The class diagram of the Command Processor pattern after applying the Composite pattern to the command class structure.

troller, we apply the singleton pattern for the controller.

(3) *Composite Pattern*[1]: The Composite pattern composes objects into tree structures to represent part-whole hierarchies. We apply this pattern to the command class hierarchy to provide macro commands, which combine several successive primitive commands. As shown in Fig. 5, a new macro command class is added to embody the role of the composite. The abstract command plays the role of the component and other primitive commands are the leaves. In this way, a macro command can contain several primitive commands or other macro commands. More importantly, the command processor can ignore the difference between compositions of command objects and individual command objects, and treat all command objects in the composite structure uniformly.

(4) *Memento Pattern*[1]: A memento is an object that stores a snapshot of the internal state of another object - the memento's originator. Only the originator can store and retrieve information from the memento. In other words, the memento is "opaque" to other objects. A caretaker will ask the originator to create a memento and is responsible for the memento's safekeeping. It never operates on or examines the contents of a memento. Later, the caretaker can use the memento to restore the originator's internal state. Thus when it is necessary to record the internal state of an object, we can capture and externalize the object's internal state without violating encapsulation by using a memento. This pattern is helpful in supporting the undo mechanism in an application.
In collaboration with the Command Processor pattern, the command object takes the role of the caretaker while the document plays the role of the originator, as shown in Fig. 6. The execution of a command object begins with requesting a memento from the document and then holds it for a time. When the undo function is applied, the command object will pass the memento back to the document to restore the internal state of the document.

(5) *Visitor Pattern*[1]: Visitor pattern allows us to define a new operation without changing the classes of the elements on which it operates. A visitor is a package of related operations from each class.

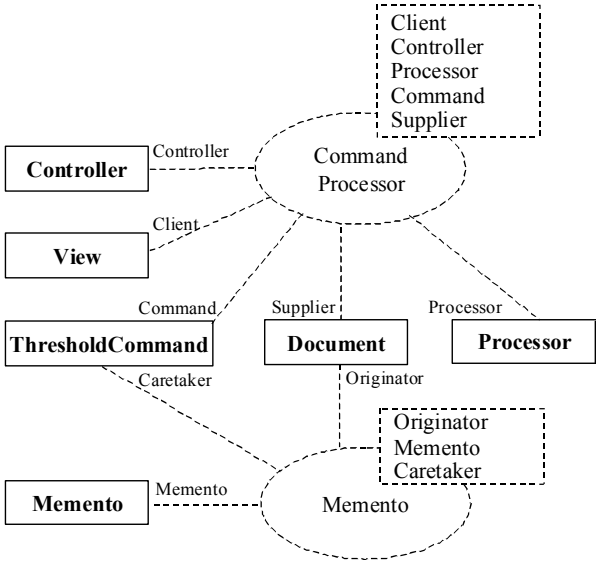

Fig. 6 The collaboration between the Command Processor pattern and the Memento pattern.

In our application, the structure of the document classes rarely changes, but we often need to add new operations over the structure. In order to avoid 'polluting' these document classes with new operations, we adopt the visitor pattern and package related operations from each document into a visitor. As shown in Fig. 7, the document takes the role of element and the command object plays the role of the client. When a command object starts its execution (usually after creating a memento), it first creates a visitor of the operation and passes it to the document. When the document "accepts" the visitor, it sends a request to the visitor. The visitor will then execute the operation for that document – the operation that used to be in the class of the document in a general object-oriented programming paradigm.

(6) *Wrapper Facade Pattern*[13]: The intent of this pattern is to encapsulate low-level functions and data structures within higher-level object-oriented class interfaces. As mentioned above, we package related operations into a visitor, but we do not implement their algorithms inside the visitor directly. Instead, we organize the algorithms into the

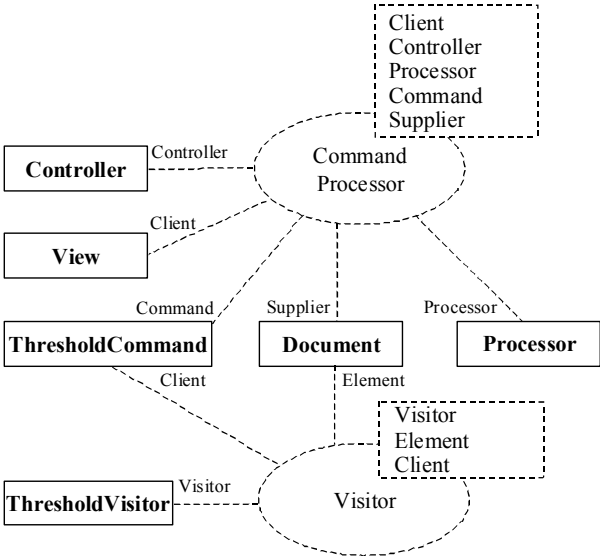

Fig. 7 The collaboration between the Command Processor pattern and the Visitor pattern.

generic functions. As shown in Fig. 8, the visitor functions as a wrapper facade, which generally forward client invocations (from the document) to one or more generic functions. Please see the section for more discussion about this arrangement.

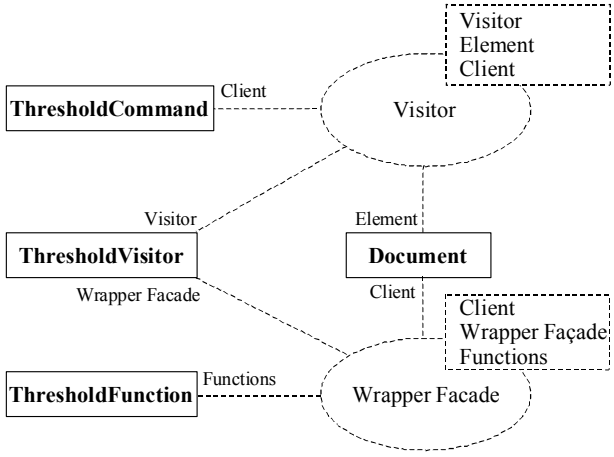

Fig. 8 The collaboration between the Visitor pattern and the Wrapper Facade pattern.

4.DISCUSSION

(1) *Patterns do more good than harm.* The introduction of patterns has a great impact on the development of Discover. One of the negative effects is the striking increase of system complexity - being full of much more classes than without patterns. To complete a specific task, the required steps and involved classes often duplicate or even more.
Although patterns complicate our system, the relationships between classes and their behavior (especially the invocation sequences) are well defined by patterns. Developers can capture the whole system easily if they know the constituent patterns well. If not, unfortunately, it is also possible to provide a guide to help them add new functions step by step. This is because the execution sequences of all operations follow a specific "pattern." We believe that it is important for medical imaging researchers to keep efforts on their expert domain, i.e. developing algorithms, without getting involved too much with the system architecture and underlying mechanisms.

(2) *Reuse of command objects.* In the medical imaging domain, we often need to process different image formats, such as gray-level images and true-color images. In terms of the DVP architecture, they represent different kinds of documents, which may need different arguments for the same operation. This makes it difficult for a command object to treat different documents uniformly. For example, when we apply a Threshold operation to a gray-level image, part of the execution codes inside the command object may look like this:

```
// pSupplier: a pointer to the document, i.e. the supplier
// nLow, nHigh: two threshold parameters
pSupplier->Threshold(nLow, nHigh);
```

or like this when applying to a true-color image:

```
// nRLow, nRHigh: threshold values for Red color
// nGLow,nGHigh: threshold values for Green color
// nBLow, nBHigh: threshold values for Blue color
pSupplier->Threshold(nRLow,  nRHigh,  nGLow,
                     nGHigh,  nBLow,  nBHigh);
```

In this case, we may need to provide more than one kind of command objects for the same operation, each kind for a document. A better solution to this problem is that a command object can be applied to different document types without any change, i.e. to reuse a command object.
The above goal can be achieved by using the Visitor pattern. The command object provides two different constructors: one for gray-level images and the other for true-color images. According to the different document types the command object is going to apply, different constructors will be invoked to create the corresponding visitors. When the execution function is actually invoked, the command object simply passes the visitor to its associated document. When the document "accepts" the visitor, it will automatically execute the correct operation of the visitor. The execution function of the Threshold command object may look like this (no matter what kind of the document is):

```
void CCmpCmdThreshold::Do()
{     // 1. get and store the memento
      m_pMemento = m_pSupplier->CreateMemento();
      // 2. apply the algorithm
      m_pSupplier->Accept(&m_VisitorThreshold);
}
```

(3) *Procedural paradigm is also important.* Over the last few years, object-oriented paradigm has been promoted as a major panacea for the software development. However, the object-oriented paradigm is a hybrid that builds on the paradigms that preceded it, among them modularity, abstract data types, procedures, and data structures. It is a hidden danger to regard the term "object-oriented" as a synonym for "good", and use the pure object-oriented paradigm only. Since not everything in nature or business is best seen as an object, we must look for opportunities to apply other paradigms. James Coplien called such a concept the "multi-paradigm design"[14].
In our medical-image analysis and generation domain, the development of domain algorithms also plays a vital role. Most of these algorithms exist better in the procedural form than as objects. In this way, we can develop and test these algorithms (functions) more independently. And further, we can also reuse these algorithms in other applications that do not necessarily use the Visitor pattern. The application of the Wrapper Façade pattern helps us obtain the advantages of the procedural programming without sacrificing benefits of the object-oriented paradigm.

(4) *Easy to implement DVP using modern window application frameworks.* The DVP is, in essence, a variant of the Document-View pattern. Several frameworks for developing window applications, such as MFC of Visual C++, adopt the Document-View pattern as their default application architecture[10]. The development platform of Discover is Microsoft Windows with the Visual C++ compiler. About the DVP architecture's implementation, we use all the Document-View related classes provided by the MFC framework, including their communication mechanism. The extra efforts we need to do include 1) suppressing the input/output functions of views in the framework; 2) providing additional presentation components and the communication mechanism between the view and its presentation components.

5.CONCLUSIONS

Patterns are already being successfully applied in many different domains. They are an important vehicle for constructing high-quality software architectures. Patterns are usually interwoven with each other in a real-world software system. In addition to knowing individual patterns to a certain extent, it is also important to know how to compose patterns to a large structure in a meaningful way. At the same time, object-oriented paradigm is not the silver bullet for the software development. We cannot ignore other paradigms such as the procedural paradigm that is possible more suitable for your application domain.

In this paper, we describe how we combine a collection of patterns and organize them into a large software architecture successfully. The patterns we use to construct the system and how they are tied together could be of help to those who are interested in developing other similar applications. Our experience shows that the application of patterns does increase the complexity of our Discover system. However, patterns also help us add new functions in a systematic way and reduce our maintenance efforts. This benefit is especially important to those development groups like our laboratory on campus whose members change constantly. In addition, we also propose the concept of reusing a command object and emphasize the importance of multi-paradigm design.

6.REFERENCES

[1] E. Gamma, E. Helm, R. Johnson and J. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995.

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

[3] P. W. Liu, L. S. Chen, S. C. Chen, J. P. Chen, F. Y. Lin, and S. S. Hwang, "Distributed Computing: New Power for Scientific Visualization," IEEE Computer Graphics and Applications, Vol. 16, No. 3, May 1996, pp.42-51.

[4] C. Alexander, A Pattern Language, Oxford: Oxford University Press, 1977.

[5] L. Rising, The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, 1998.

[6] J. O. Coplien and D. C. Schmidt, Pattern Languages of Program Design, Addison-Wesley, 1995.

[7] W. Pree, Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1994.

[8] W. Zimmer, "Relationships Between Design Patterns," Proceedings of PLoP'94, 1994.

[9] C. Alexander, The Timeless Way of Building, Oxford: Oxford University Press, 1979.

[10] D. Kruglinski: Inside Visual C++, Microsoft Press, 1996.

[11] K. Y. Chang, L. S. Chen and C. K. Lai, "Document-View-Presentation Pattern," Proceedings of PLoP'99, 1999.

[12] G. Booch, J. Rumbaugh and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[13] D. C. Schmidt, "Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes," C++ Report, Vol. 11, No 2, February, 1999.

[14] J. O. Coplien, Multi-Paradigm Design for C++, Addison-Wesley, 1999.