

Implementing Design Patterns Using Advanced Separation of Concerns

Natsuko Noda Tomoji Kishi
NEC Corporation
1753, Shimonumabe, Nakahara-Ku,
Kawasaki, Kanagawa 211-8666, Japan
+81-44-435-9491

n-noda@cw.jp.nec.com

kishi@aj.jp.nec.com

ABSTRACT

Design patterns have been recognized to be very important and useful. Especially, for software reuse and evolution, they have strong potential. In order to utilize design patterns in real software development, we need appropriate implementation techniques of them. Ordinary implementation of design patterns, however, is not always enough. One of the most typical ways is based on the inheritance mechanism. This makes applications quite dependent on patterns and reduces reusability of the application core part. In addition, it is difficult to exchange patterns once a system is designed. In this paper, we introduce a way to implement design patterns more flexibly, applying the concept of separation of concerns. It is realized by separating design patterns, as explicit concerns, from the application core that is responsible for the primary functionalities. This approach improves the reusability of both the design pattern part and the application core part of a system, and it provides the possibility to switch patterns using the very same application core classes. The new implementation technologies that support the advanced separation of concerns such as AspectJ and Hyper/J help with coding this kind of design. We show that the introduced implementation of design patterns can be coded in reality using these new language/environment, and compare the two language/environment. Furthermore, we show the applicability of this method to the GoF's patterns.

Keywords

Design pattern, advanced separation of concerns, AspectJ, Hyper/J, software reusability, software evolution

1. INTRODUCTION

Design patterns have been recognized to be very important and useful in real software development. They support developers in many ways: they give us useful vocabulary and they themselves are reusable software assets. Especially, for software reuse and evolution, they have strong potential: most of them intend to make systems more flexible and extensible. The importance of implementation of design patterns has been also pointed out [2]. Here, implementation means designing a system using design patterns and realizing the intention of the design in the code level.

Ordinary implementation, however, is not always enough. One of the most typical ways is to define concrete classes specific to each application, inheriting abstract classes that express design patterns. This way makes these application-specific concrete classes very dependent on design patterns, thus it reduces the reusability of these concrete classes. For example, consider a text editor application using the Observer pattern [3]. We may define abstract classes such as Observer and Subject to express the common structure of the pattern, and then define Text class inheriting the Subject and TextEditor class inheriting the Observer. The Text class and the TextEditor class may contain many application specific functions in them, such as retrieving text strings in the Text class and displaying a string in the TextEditor class. These functions could be independent of the Observer pattern: even if, for example, the timing of updating the TextEditor is changed, or the way to transfer data from the Text to the TextEditor, these application specific functions may not need to be changed. We want to reuse the classes Text and TextEditor that contain such application specific functions as they are. However, it is difficult at best, so long as the classes Text and TextEditor inherit the design pattern classes, Observer and Subject. The reusability of this kind of application specific classes is important. Especially, in a real large system, these classes could be big and contain many important functions, thus it is desirable to reuse these classes.

In addition, we may want to "switch" patterns. Because different design patterns have different advantage and/or drawbacks, even if they realize similar functionality, another design pattern would become suitable for a system as it evolves. Or different behavior that could not be realized by the original patterns might be needed later. However, we cannot exchange design patterns using same application specific classes, so long as these application specific classes are inherited from design patterns.

In order to avoid the problem described above, we apply the concept of separation of concerns to the implementation of design patterns: application classes are not designed to be inherited from design patterns, but design patterns and application core are made to be two different concerns. Especially, we make good use of technologies of a new generation of separation of concerns, advanced separation of concerns (ASOC) [7][8]. ASOC provides more flexible

concern encapsulation mechanisms and concern composing techniques. Using these mechanisms instead of the inheritance mechanism, an application can be composed of two separated concerns, the design pattern concern and the application core concern.

In this paper, we propose an implementation technique of design patterns that improves reusability of design pattern part and application core part both. For coding, we could use several ASOC language and/or environment. We implemented some design patterns on AspectJ [1] and Hyper/J [6]. We show these implementation examples and compare these two language/environment. Furthermore, we consider the applicability of this method to the GoF's patterns [3]. The major contributions of this paper are the following.

- It introduces an implementation technique of design patterns that makes an application independent of design patterns and improves reusability of both design pattern part and the application core part, using ASOC.
- It shows the possibility that the implementation can be coded in reality using typical ASOC languages/environments, AspectJ and Hyper/J.

Section 2 explains the design direction of our approach: how to separate application core part and design pattern part as different concerns. Section 3 describes how to realize the design direction in AspectJ and Hyper/J, and compares the implementations in two language/environment. Section 4 shows that patterns could be switched in an application using the very same application core in our approach. Section 5 provides consideration to applicability of our approach to the GoF's patterns. Section 6 discusses the contributions of our approach comparing it with related works. Section 7 concludes the paper.

2. DESIGN PATTERN AS CONCERN

In this section, we explain how to separate application core part and design pattern part as different concerns. In order to show the detail concretely and clearly, we take the Observer pattern and a small application using this pattern as an example.

2.1 Observer Pattern

The Observer pattern is to define a dependency between objects so that when one object changes state, its dependents are notified and updated automatically [3]. Figure 1 shows the structure of the pattern.

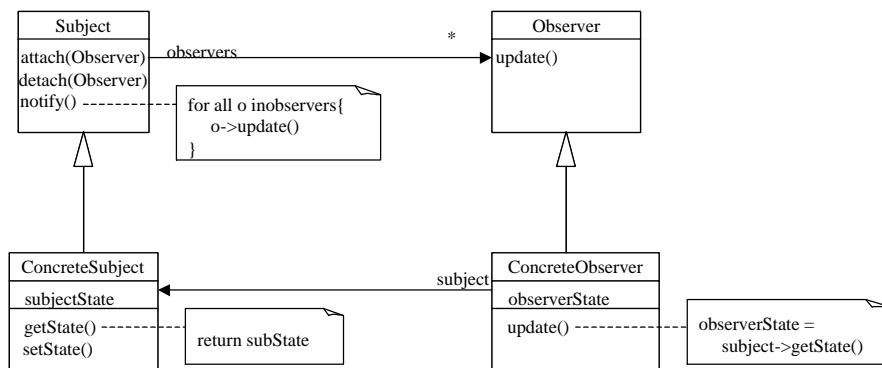


Figure 1 Structure of Observer Pattern

Consider a small application as follows:

A counter object contains a number and the value of the number is increased over time. A display object contains buttons and manages their allocations. Each button object displays a text that shows the number contained by the counter, when it is active. The state of active and non-active of each button is switched by a click.

If we apply the Observer pattern to this application, one ordinary object-oriented design is that a class Button is a subclass of the Observer class and a class Counter is a subclass of the Subject class.

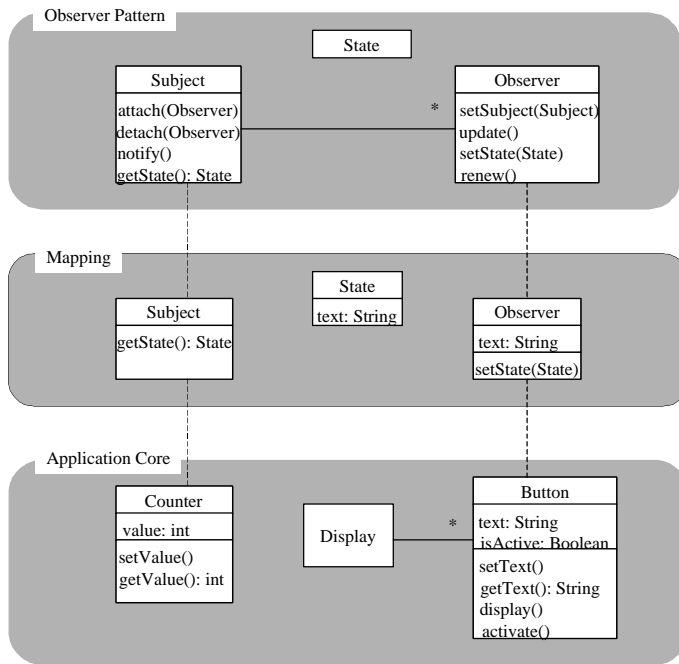


Figure 2 Design of Example Application – Using Observer Pattern

2.2 Separating Design Pattern and Application Core

As we described before, the typical design of this application described in 2.1 makes the Button class and the Counter class strongly dependent on the Observer pattern. We separate this application into two parts that are manipulated by different concerns: one is responsible to realize application core functionality, the concern of the application core, and the other is relevant to the Observer pattern, the concern of the design pattern. In the following sections, we show the detail of each concern.

2.2.1 Concern of the Application Core

The concern of the application core is responsible to realize application core functionality. Many application specific functions could be independent of design patterns. For example, how the counter counts up the number is irrelevant to the Observer pattern. How the button display texts is also irrelevant to the pattern. We encapsulate these functions in the application core concern, defining classes in this concern to contain just these application specific functions. This makes the application core part to be reused regardless of patterns

Table 1 shows the classes contained in the application core part. These classes have just only methods which are independent of the Observer pattern, such as setValue(int) and display(). The bottom of Figure 2 also shows the concern of the application core.

Table 1 Application Core Classes

Counter	has a number and increases the number automatically.
Display	manages the allocation of the buttons.
Button	has a text and sets and displays the text.

2.2.2 Concern of the Design Pattern

The concern of the design pattern describes the abstract structure and behavior that is determined by the design pattern, and it does not define any application specific behavior. In the structure of the Observer pattern described in Figure 1, class Subject, class Observer and the association between them are determined independent of the application. Operations that the Subject has, such as attach, detach, and notify, are also independent of the application: these are responsible for the pattern-specific functions. It is also the intention of the pattern that the Observers query the Subject for its current state and update their own states, thus this interaction is also contained in the concern of the design pattern. However, the actual data transferred in this interaction is specific to the application. So we define an abstract data type for this interaction in the design pattern concern.

Table 2 Observer Pattern Classes

Observer	is notified of changes in a subject, then query the subject state and updates its own state.
Subject	notifies its observers when a change occurs.
State	defines a data type of the subject's state transferred between the subject and the observers. Just the name declaration and it is empty class.

Table 2 shows the classes contained in the concern of the design pattern. In Figure 2, this concern is showed at the top of it.

Figure 3 shows a part of pseudo-codes of the Observer class and the Subject class. The code tells that, for example, it is defined that the Subject notifies its all Observers of its change, but it is not defined what the Observer does after it changed its state.

```
Class Observer {
    ...
    void update() {
        setState(subject.getState());
        renew();
    }
    abstract void setState(State state);
        // not define concrete behavior
    abstract void renew(); // not define concrete behavior
}
Class Subject {
    ...
    void notify() {
        for all o in observers {
            o.update();
        }
    }
    abstract State getState(); // not define concrete behavior
}
```

Figure 3 Pseudo-code of Observer and Subject

2.3 Mapping Problem

The application core and the design pattern are separated concerns. We have to compose these concerns to make a meaningful application. To do this, we define the mapping between two concerns. The mapping means the correspondence between each element (class, operation and attribute) to merge the correspondent elements into one element. There are a couple of options of designs to support this mapping. It partly depends on which language is used in coding. In this section, we explain the very general idea in the basis of the design options, and the detail dependent on languages is described in 3.1 and 3.2. In this example, the Counter is mapped to the Subject and the Button is mapped to the Observer in order to realize the desired behavior. To do this, data conversion between the abstract data types used in the Observer pattern part and the concrete data used in the application core part is also needed. For this mapping we define classes that manage the mapping between each pattern class and each application class and convert the data. Figure 2 shows the entire design of this application using the Observer pattern. In this diagram, a dotted line means that classes connected by this line are mapped each other.

3. CODING EXAMPLE

In this section, we shows two implementation examples of the observer example described in the section 2, one is in AspectJ and the other is in HyperJ.

3.1 Observer Example in AspectJ

AspectJ is an extension to the Java programming language supporting the ASOC. AspectJ provides a way to encapsulate concerns that crosscut a system's structure. A concern can be encapsulated in the *aspect* construct. This construct provides a means of grouping together code related to a concern. To control system's behaviors that divided into different aspects, there are several constructs in AspectJ. Among these, we use the following constructs for the implementation of design patterns. The *advice* construct groups together code that is to be executed at some defined point in the system's execution in an *aspect* construct. The defined point may be *before*, *after*, or *around* the execution of a method. For this study, we used Version 0.7beta11.

The *aspect* construct has similar structure to the Java class construct. It may be defined according to a class: an aspect may be aspect of each object that is instance of a class. Therefore, instead we define an aspect according to each concern (a concern of application core or a concern of design pattern), we define an aspect according to each class in each concern. To be exact, we define classes in the application core concern as standard Java classes, and classes in the design pattern concern and the mapping part are defined as aspects. In order to show clearly each concern and to make it easy to reuse each concern, we make all classes in one concern to be in a package.

```

package observerdemo.observer;

public abstract aspect Observer {
    protected Subject subject;
    public void setSubject(Subject s) {
        subject = s;
        subject.attach(this);
    }
    public void update() {
        setCurrentState(subject.getState());
        renew();
    }
    abstract protected void setCurrentState(State state);
    abstract protected void renew();
}

```

Figure 4 Observer Aspect

```

package observerdemo.observer;

import java.util.Vector;

public abstract aspect Subject {
    protected Vector observers = new Vector();
    public void attach(Observer obs) {
        observers.addElement(obs);
    }
    public void detach(Observer obs) {
        observers.removeElement(obs);
    }
    abstract public pointcut stateChanges();

    after(): stateChanges() {
        for (int i = 0; i < observers.size(); i++) {
            ((Observer)observers.elementAt(i)).update();
        }
    }
    abstract public State getState();
}

```

Figure 5 Subject Aspect

Figure 4 and Figure 5 show the code of the design pattern concern. These definitions of aspects are very similar to the definition of abstract classes. In Figure 5, *pointcut* names a system's execution point. Every time after the system reaches this pointcut `stateChanges()`, `update()` of all observers are called. Using the *before* or *after* advice, we can declare the interaction between classes (in this case, aspects) more clearly.

Figure 6 and Figure 7 show the code for the mapping part between the application core concern and the design pattern concern. We define aspects for the classes in the mapping part. These aspects inherit abstract aspects in the design pattern concern, and show to which class in the application core concern they are applied. For example, aspect `ButtonAsObserver` is for the mapping between the `Observer` and the `Button`. This aspect inherits aspect `Observer` in the design pattern concern and is applied to the `Button` class in the application core concern. In these mapping aspects, it is defined which concrete function is actually executed to realize the behavior the pattern determines. See the operation `renew()` of aspect `ButtonAsObserver` (Figure 6). The operation `renew()` is called from the operation `update()` of the aspect `Observer` (Figure 4), and the operation `display()` of the `Button` is actually executed in the `update()`.

```

package observerdemo.converter;

import java.util.Vector;
import java.awt.Color;

import observerdemo.observer.*;
import applicationdata.*;

public aspect ButtonAsObserver extends Observer
  of eachobject(instanceof(Button)) {
  protected void setCurrentState(State state) {
    thisButton.setTextString(((TextColor)state).text);
  }
  protected void renew() {
    thisButton.display();
  }
  // The Button object that this aspect is of
  private Button thisButton;

  after(Button bn) returning (): instanceof(bn) && receptions(new(..)){
    thisButton = bn;
  }
}

```

Figure 6 Mapping of Observer and Button

```

package observerdemo.converter;

import java.util.Vector;
import java.awt.Color;

import observerdemo.observer.*;
import applicationdata.*;

public aspect CounterAsSubject extends Subject
  of eachobject(instanceof(Counter)) {
  public State getState() {
    return (State)(new Data(getText()));
  }
  public String getText() {
    return (new Integer(thisCounter.getValue())).toString();
  }
  pointcut stateChanges():
    (receptions(void setValue(..)));

  // The Counter object that this aspect is of
  private Counter thisCounter;

  after(Counter cn) returning ():
    instanceof(cn) && receptions(new(..)){
    thisCounter = cn;
  }
}

```

Figure 7 Mapping of Subject and Counter

3.2 Observer Example in Hyper/J

Hyper/J is an environment supporting the advanced separation of concerns in standard Java software. When using Hyper/J, developers provide three inputs: 1) a *hyperspace* file that describes the Java class files being composed, 2) a *concern mapping* file that describes pieces of Java within those files, e.g. packages, classes, and operations, that map to different concerns, and 3) a *hypermodule* file that describes how integration between concerns should be done. We can implement each class of each concern in standard Java. We made a separated Java class package for each concern

respectively. In this implementation, we made the mapping part between the application core concern and the design pattern concern another concern and made package for it also.

Figure 8 shows the mapping class Subject in the mapping concern. Here it is defined how the abstract data type used in the design pattern concern is converted to the concrete data in the application core part. Notice which class in the application core concern is mapped to a class in the design pattern concern is not specified here. That is defined in a hypermodule file, which is explained later.

```
public abstract class Subject {
    public State getState() {
        return (new State(getText()));
    }
    public String getText() {
        return (new Integer(getValue())).toString();
    }
    public abstract int getValue();
}
```

Figure 8 Class Subject in Mapping Concern

Figure 9 shows the concern mapping file for the Observer pattern implementation.

```
-concerns
package applicationdata : Feature.Kernel
package observerdemo.mapping : Feature.Mapping
package observerdemo.observer : Feature.Pattern
```

Figure 9 A Concern Mapping File for The Observer Pattern Implementation

Figure 10 is a portion of the hypermodule file for the Observer pattern implementation. The *equate* relationship specifies to which classes in the other concerns a class maps to. It also defines which concrete application method and abstract method in the pattern are integrated. As the *mergeByName* relationship is designated in this file, all methods that have the same name are merged in one. Therefore, there are many other methods integrating with others than the methods specified explicitly by *equate* relationship. The line beginning with *bracket* specifies that the *notify()* method of the Observer pattern concern is always called after the *setValue()* method of the Counter class is called.

```
-hypermodules
...
relationships:
mergeByName;
equate class Feature.Kernel.Button,
           Feature.Mapping.Observer,
           Feature.Pattern.Observer;
equate class Feature.Kernel.Counter,
           Feature.Mapping.Subject,
           Feature.Pattern.Subject;
bracket "Counter"."setValue" with
after Feature.Pattern.Subject.notify;
equate operation Feature.Kernel.display,
                Feature.Pattern.renew;
end hypermodule;
```

Figure 10 A Hypermodule File for the Observer Pattern Implementation

3.3 Variation

One of the important intentions of the Observer pattern is that the dependency between Subject and Observer may be one-to-many, and multiple types of concrete Observers can be handled as the same Observer by the Subject. The example described in the previous sections contains just one concrete Observer. In this section, we extend the example and consider the case that contains two different types of Observer.

We extend the example as follows: in addition to the button, a text field display a text that shows the number contained by the counter at the same time.

The design of this extended example is shown in Figure 11.

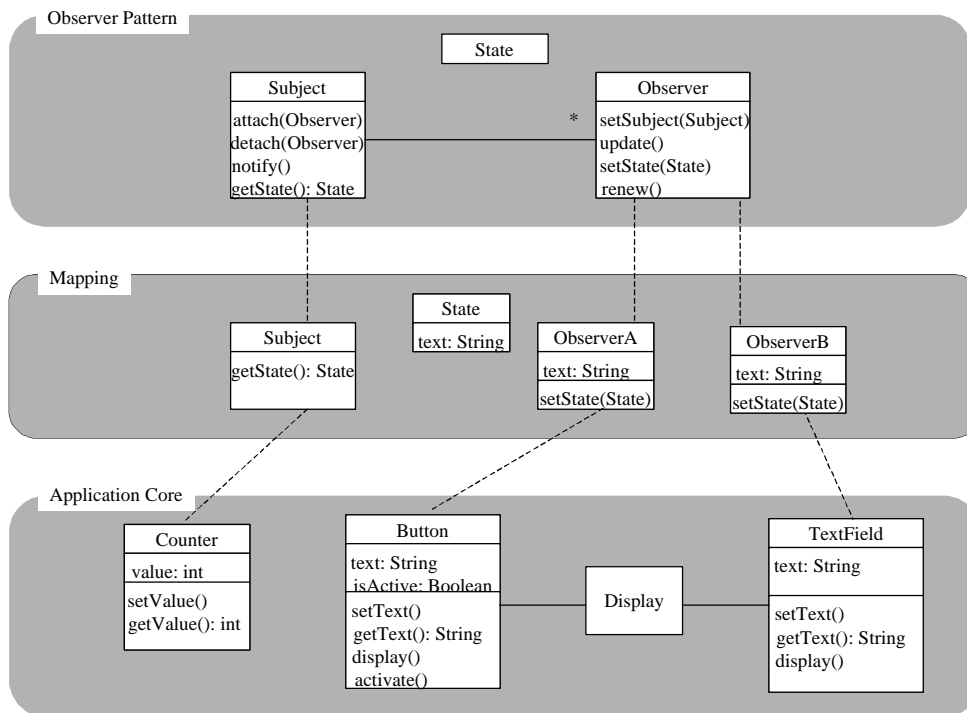


Figure 11 Variation of the Observer Pattern Example

Consider implementing this example in AspectJ. In this case, we can easily apply the technique described in 3.1 to this extended example. We just define only one more mapping aspect: the aspect mapping the Observer to the TextField. We show the code of this aspect in Figure 12.

Implementing this example in Hyper/J is not so simple. Currently, Hyper/J does not support to compose one class in a concern with several classes in another concern. Thus, we cannot map the classes ObserverA and ObserverB in the mapping concern to the class Observer in the design pattern concern. A current solution for this problem is to make copies of the classes Observer, Subject and State in a different package and map the second Observer in the mapping concern to the copied Observer in this different package.

```
import java.util.Vector;
import observerdemo.observer.*;
import applicationdata.*;

public aspect FieldAsObserver extends Observer
of eachobject(instanceof(TextField)) {
protected void setCurrentState(State state) {
thisField.setString(((TextColor)state).text);
}
protected void renew() {
thisField.display();
}
// The Field object that this aspect is of
private TextField thisField;
after(TextField fd) returning () : instanceof(fd) && receptions(new(..)){
thisField = fd;
}
}
```

Figure 12 Mapping of Observer and TextField

3.4 Comparison Between Implementations in AspectJ and Hyper/J

Basically, the introduced implementation technique of design patterns can be coded similarly in both AspectJ and Hyper/J. We point some differences out.

- In Hyper/J, every concern is coded with normal Java class construct. It makes the program easy to understand. On the other hand, in AspectJ, every class in the design pattern concern is described by an aspect. It could increase difficulty to understand the program for some developers.
- In Hyper/J, both the application core part and the design pattern part are clearly described as concern. In other words, every concern is equal. In AspectJ, only the application core part is written in normal classes and the other parts are written with aspects. This would imply that the design pattern part is a concern but the application core part is not a concern but the entity of the application.
- One of the biggest differences is the code for the mapping part. In AspectJ, the data conversion for the mapping and the correspondence between classes in the application core and classes in the design pattern are both described as aspects. This makes aspects in the mapping concern complicated. And we would have to write many similar codes for the mapping part. On the other hand, in Hyper/J, the correspondence between classes is described in a hypermodule file. This simplifies the mapping part and the classes in the mapping concern could be reused.
- For this implementation technique of design pattern, one of the biggest drawback of Hyper/J is that it lacks the ability to compose one class with several classes in another concern¹. If there are multiple kinds of concrete classes inheriting an abstract class in original design patterns, AspectJ would provide a simpler solution.

4. SWITCHING PATTERNS

In this section, we explain that we could “switch” design patterns using the same application core in this implementation technique.

Consider implementing the same application described in 2.1 using the Composite pattern [3]. The application core part described in 2.2.1 is independent of the pattern. Therefore, the very same classes in the application core concern can be used in the design of the case where the Composite pattern is used. The concern of the composite pattern is separated from the application core concern in the same way we described in 2.2.2.²

For the realization of the desired behavior, the Counter is mapped to the Client, the Display is mapped to the Composite, and the Button is mapped to the Leaf.

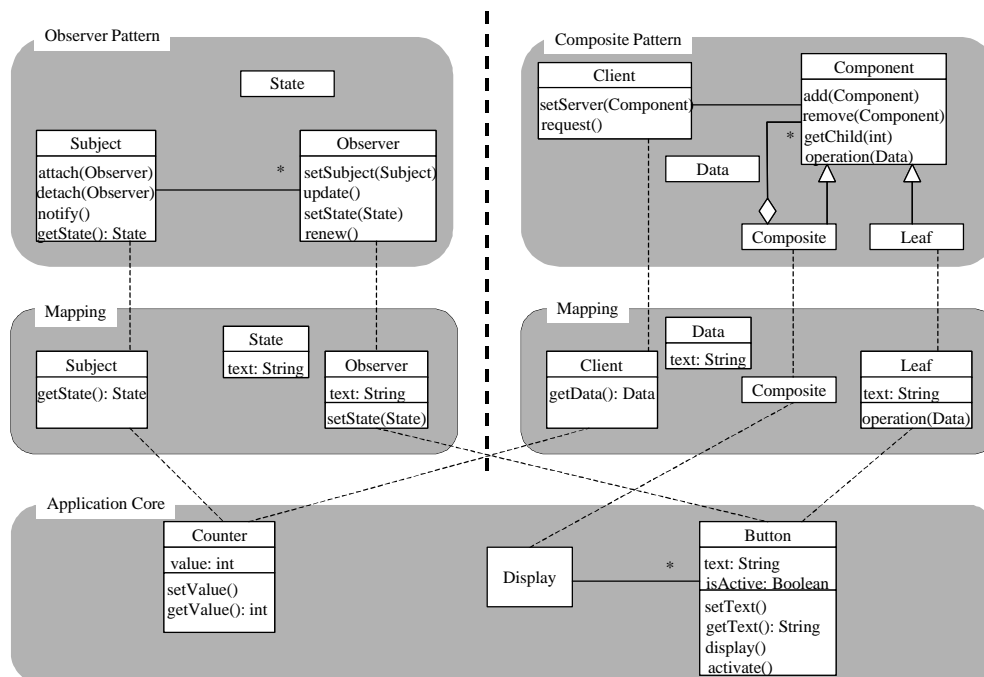


Figure 13 Application Using Observer Pattern And Composite Pattern

¹ We understand this lack is the temporal problem. The original intention of Hyper/J would contain this kind of composition. We hope a future version of Hyper/J support this.

² The main purpose of the Composite pattern provides the part-whole structure. However, it is often used to make a chain of operations according to the structure. We include this chain of operations in the pattern part.

The right part of Figure 13 shows the designs of the application using the Composite pattern. The left side of Figure 13 is as same as Figure 2, which is presented here again to show clearly how we switch the pattern using the same classes of the application core concern.

5. APPLICABILITY TO OTHER DESIGN PATTERNS

We have examined implementations of GoF's design patterns [3] according to our approach. Many of them can be implemented with our approach, and we did actual coding in AspectJ and Hyper/J. On the other hand, we have found that not every design pattern is suitable to this style of implementation.

The GoF's patterns include some patterns for the creational purpose. We don't think our approach is suitably applied for them. In our approach, the concerns of design patterns could be recognized to provide some views on the entities. On the other hand, the creational patterns concern the process of object creation. If we separate the design patterns and the application and handle the object creation issues in the concern of design patterns, what does it mean? The meaning of creating objects as views is unclear.

Some of design patterns concern very low level issues, such as memory management. These issues would be suitably solved by languages. Thus, we have not examined the implementation of this kind of patterns. The patterns such as Prototype, Singleton, Flyweight and Proxy are this kind of patterns.

On the other hand, some patterns are very abstract. They just describe design concept or design guideline. Façade is the typical pattern of this kind. It is difficult to implement these patterns using our approach.

Sometime we might want to convert or change the interface of a class to another one to provide collective interface of a class group. For this purpose, patterns would use delegation and/or inheritance. Because ASOC has ability to compose two different modules (classes or aspects) into one module, we do not have to use delegation or inheritance in ASOC. Thus, for a pattern simply using delegation or inheritance, ASOC would remove necessity of the pattern. Typical example is Adapter.

Figure 14 represents a part of codes of a simple example in Hyper/J. In this example, when Adapter.request() is called, Adaptee.specialRequest() is actually executed, without using the Adapter pattern.

```
class Adapter {
    virtual void request();
    ...
}

class Adaptee {
    void specialRequest() {
        // definition of specific behavior
        ...
    }
}

// Hypermodule file
-hypermodules
...
relationships:
mergeByName;
equate class Feature.Kernel.Adapter,
           Feature.Implementation.Adaptee;
...
equate operation Feature.Kernel.request,
                 Feature.Implementation.specialRequest;
...
end hypermodule;
```

Figure 14 Adapter Implementation in Hyper/J

However, many other patterns than the pattern described above do not use just one delegation or inheritance, but also make structure composing these techniques, in order to realize specific behavior. To these patterns, our approach is suitably applied. For example, consider the Decorator pattern. This pattern uses inheritances and associations to realize a behavior that makes a chain of functions. The structure making a chain of functions is realized by the pattern concern, and each concrete function is realized by the application core concern. Figure 15 shows the design of a GUI application using the Decorator pattern according to our implementation technique. This application would decorate a textview with borders and scrollbars.

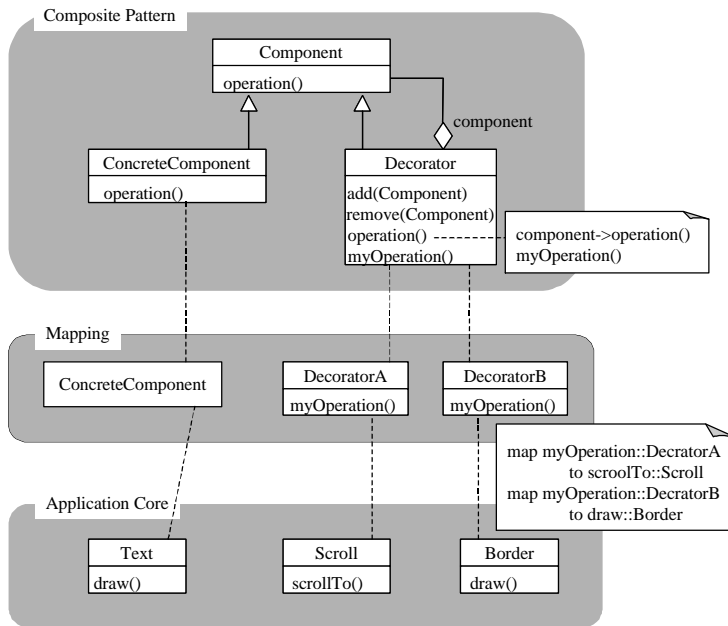


Figure 15 GUI Application Using Decorator Pattern

6. DISCUSSION

The importance of design patterns is broadly recognized. On the other hand, several problems of implementing design patterns have been pointed out [2], for example, ordinary object-oriented style implementations reduce the traceability of design patterns and the reusability of the implementation of design patterns. Our approach is one of the solutions for these problems. Separating design patterns as concerns enhances the traceability of the patterns. And some examples in the paper show concretely that the code for a design pattern concern can be reused. Although there is some related work for these problems [9], enhancing the reusability of both the design pattern part and the application core part is the characteristic and the advantage of our approach.

For the implementation of design patterns, the design policy to consider the patterns as concerns is important. At the same time, it is to be desired that we have effective languages and tools supporting the advanced separation of concerns. As we showed in this paper, AspectJ and Hyper/J have the power to support our approach. These are being improved and getting more functionalities. Even though they still have a couple of limitations, we believe they have strong potential to the implementation of design patterns. And as we described in 3.4, each of them has different advantages and drawbacks. However, we should not decide which one is suitable for the implementation right now, because they are growing. We hope such experiment of using these languages and environments as our approach gives important information to the developers of these languages and environments and improves them.

Although many researches about ASOC are going on [8], it is still vague what is a concern and what concerns are useful to encapsulate. Our approach shows concrete examples of concerns: design patterns. We studied about GoF's design patterns and implemented them. There are more patterns other than GoF's patterns and our approach would be applicable to many of those patterns. Studying those patterns and considering to which patterns our approach is applied, as we described in 5, would help us to understand more precisely what is a concern.

Another challenge about ASOC is understanding how to separate and encapsulate concerns in the design level and the code level. There is an empirical study about this issue [5]. They rewrote existing systems with a couple of ASOC languages and showed examples of how to structure the code to enable a concern to be separated. Our study is also another example for this kind of issue. While they focus on the code level mainly, we consider both the design level and the code level.

7. CONCLUSION

In this paper, we introduced a new technique for implementation of design patterns. Design patterns are important, and we need appropriate implementation techniques of them in order to utilize them in real software development. Ordinary implementation technique is not enough, especially for the reusability. We applied the advanced separation of concerns to the implementation of design patterns and have introduced the technique that enhances the reusability of both the design pattern part and the application core part. We applied our technique to the implementations of the GoF's design patterns and showed the availability of our technique. We used AspectJ and Hyper/J for the coding and showed the possibility that the implementation can be coded in reality.

8. ACKNOWLEDGMENTS

We would like to thank Harold Ossher for his help with our applications of the Hyper/J, and David Notkin for useful comments and discussion on our approach.

9. REFERENCES

- [1] AspectJ. <http://www.aspectj.org>.
- [2] Bosch, J. Design Patterns & Frameworks: On the Issue of Language Support, In Proceedings of ECCOP'97 (Position Paper for LSDF'97), 133-136.
- [3] Gamma, E. et.al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
- [4] Kiczales, G., et.al. Aspect-oriented programming. In Proceedings of ECOOP'97, 220-242.
- [5] Murphy, G., et.al. Separating Features in Source Code: An Exploratory Study. In Proceedings of ICSE, 2001, 275-284.
- [6] Ossher, H. and Tarr, P. Hyper/J: Multi-dimensional separation of concerns for Java. In Proceedings of ICSE, 2000, 734-737.
- [7] Workshop on Advanced Separation of Concerns at OOPSLA 2000.
- [8] Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001.
- [9] Workshop on Language Support for Design Patterns and Frameworks (LSDF'97) at ECOOP'97.