Sustaining Composability of Aspect-Oriented Design Patterns in Their Symmetric Implementation

Jaroslav Bálik and Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 3, 84216 Bratislava 4, Slovakia
balki.balkovic@gmail.com, vranic@fiit.stuba.sk

Abstract. In aspect-oriented programming, it may be distinguished between asymmetric and symmetric implementation. While asymmetric approach dominates in programming, symmetric approach shows its importance in modeling. Aspect-oriented design patterns are known almost exclusively in their asymmetric, AspectJ-like implementation. This paper presents three aspect-oriented patterns—Director, Cuckoo's Egg, and Border Control—described in Coplien's form and implemented symmetrically in Hyper/J. Sustaining composability of the symmetric implementations of these patterns has been demonstrated by a small study that required the composition of these patterns. Although it has been possible to develop a fully functional composition of symmetric pattern implementations, there have been some differences with respect to the asymmetric composition worth mentioning related to the way Cuckoo's Egg and Border Control are composed and limitations of the last available Hyper/J version. Beyond the patterns presented here, some other patterns could not be implemented in Hyper/J: Exception Introduction, Worker Object Creation, Policy, nor Wormhole.

Keywords: intrinsic aspect-oriented design patterns, symmetric aspect-oriented programming, asymmetric aspect-oriented programming

1 Introduction

Apart from object-oriented design patterns that can be used or reimplemented in aspect-oriented programming languages that are realized as extensions of object-oriented programming languages, there are also design patterns intrinsic to aspect-oriented programming. Aspect-oriented design patterns are mainly related to AspectJ [10, 11]. Some of them have been implemented in CaesarJ, too [4], proving that they are more than just AspectJ idioms. However, all these implementations are related to what is known as PARC aspect-oriented approach which is known to be *asymmetric*.

Symmetry of aspect-oriented approaches is based on the dichotomy of PARC AOP [9], with AspectJ as its language representative, and subject-oriented programming, represented by Hyper/J. PARC AOP is considered to be asymmetric, while subject-oriented programming to be symmetric, and this is most apparent in element symmetry. Simply stated, asymmetric aspect-oriented approaches distinguish between so-called basic elements and aspects that affect them or other aspects. Symmetric aspect-oriented approaches treat all elements equally. These elements are composed according to the composition rules that are usually introduced separately.

Beside element symmetry, a complex view of symmetry includes relationship and join point symmetry [7]. While PARC AOP appears to be fully asymmetric, and subject-oriented programming fully symmetric, other approaches may exhibit mixed symmetry. Thus, composition filters are asymmetric with respect to elements: the main concern is implemented in classes and crosscutting concerns are implemented in input and output filters. With respect to relationships, composition filters are asymmetric because only filters can affect other concerns.

It is worth noting that in the beginning, subject-oriented programming was even doubted to belong to aspect-oriented area [9, 13], but now it is recognized as its fundamental part despite lacking industry-strength language support. Both perspectives are valuable and this is probably most remarkable in Jacobson's and Ng's work on aspect-oriented software development with use cases [8] in which peer use cases represent symmetric decomposition, while the extend relationship represents asymmetric decomposition. It is also applied in the Theme approach to aspect-oriented modeling [8, 2].

An aspect-oriented design pattern should be valid for both asymmetric and symmetric aspect-oriented languages. Of course, this is not a proof that a pattern can be realized in all aspect-oriented languages (and it actually doesn't have to be realizable in all aspect-oriented languages), but constitutes a fundamental diversity in applicability. The aim of the study reported in this paper was to explore whether selected aspect-oriented design patterns can be implemented in a symmetric way and whether their composability sustains among their symmetric implementations. We used Hyper/J for symmetric pattern implementation despite the fact that this is a dead programming language. In our opinion it is still interesting as a promoter of the idea of symmetric aspect-oriented programming.

The rest of the paper is structured as follows. Section 2 presents symmetric implementations of selected aspect-oriented design patterns. Section 3 presents the composition of symmetric pattern implementations as a way of evaluating their validity. Section 4 discusses the symmetric implementations and their composition. Section 5 discusses related work. Section 6 concludes the paper.

2 Expressing Aspect-Oriented Design Patterns in a Symmetric Way

Aspect-oriented patterns have been defined virtually by their implementations in AspectJ. This is not an appropriate way to define a pattern and we have to

step back from the implementation details and define the pattern in a general manner. One way to do this is certainly to use Coplien's form [3] as an adaptation of the original, Alexander's form of pattern description [1].

Three aspect-oriented patterns—Director, Cuckoo's Egg, and Border Control—have been analyzed and expressed in Coplien's form [3]. Based on this, their symmetric implementation in the last available Hyper/J version[1] has been developed. Concern Manipulation Environment that was supposed to replace Hyper/J (actually, it included Hyper/J2) never became publicly available[2])

2.1 Director

In its AspectJ realization, the Director pattern defines additional roles as interfaces that are enforced onto the existing types by the declare parents intertype declarations. A general description of Director in Coplien's form is as follows:

Problem: Additional roles have to be defined in application.
Context: A type hierarchy that defines the roles.
Forces: The application has to be extended with additional roles, but the original class hierarchy in the source code has to remain free from these roles.
Solution: Introduce the additional roles as types and enforce their implementation by the corresponding types externally.
Resulting Context: The type hierarchy preserved in the source code, but extended with new roles in execution.
Rationale: Director provides two main benefits: the application behavior can be easily changed by replacing a particular concern and the core functionality is less complicated.

Let us demonstrate how Director may be implemented in Hyper/J. In Hyper/J, partial class definitions that belong to one application view—or aspect—are grouped into so-called *hyperslices*. In the Java part of the language, hyperslices are represented by packages. Hyperslices contain partial class and interface definition to be composed into so-called *hypermodules* that form complete, runable versions of the application.

For simplicity, suppose that the application in which Director is to be applied is defined in a single hyperslice named objects. The StateChanger class changes its state, while the StatePrinter class prints it if changed:

```
package objects;
public class StateChanger {
   int state;
   public StateChanger() { state = 0; }
   public int getState() { return state; }
   public void changeState(int i) { state = i; }
}
public class StatePrinter {
   StateChanger sch;
```

[1] `http://www.alphaworks.ibm.com/tech/hyperj/`
[2] `http://www.research.ibm.com/cme/hyperj.html`

```
    public StatePrinter(StateChanger s) { sch = s; }
    public void printState() {
        System.out.println("State changed to: " + sch.getState());
    }
}
```

We will use Director to enforce the Observer roles onto the existing classes. The observer hyperslice contains the participant roles of the Observer pattern:

```
package observer;
public class Subject {
    ArrayList<Observer> observers;
    public Subject() { observers = new ArrayList<Observer>(); }
    public void attach(Observer o) { observers.add(o); }
    public void detach(Observer o) { observers.remove(o); }
    public void notif() { for(Observer o : observers ) { o.update(this); } }
}
public interface Observer {
    void update(Subject subject);
}
```

The role enforcement itself is achieved in the dummy hyperslice by redeclaring the StateChanger and StatePrinter class with the addition of the appropriate **extends** or **implements** clause:

```
package dummy;
public abstract class StateChanger extends Subject {
    public StateChanger() { }
    public void changeState(int i) { this.notif(); }
    public abstract int getState();
}
public abstract class StatePrinter implements Observer {
    public StatePrinter(StateChanger s) { s.attach(this); }
    public void update(Subject subject) { printState(); }
    public abstract void printState();
}
```

If not instantiated in a hyperslice where they are redeclared, the classes should preferably be abstract there [12]. All the methods used in the hyperslice have to be at least declared, so it could be compiled [12]. They don't have to implement a correct behavior if it is supposed to come with a composition.

In the composition file, the hyperslices are composed by the mergeByName statement introduced in the relationships section. This simply merges equally named elements (classes) from different hyperslices [12]:

```
−concerns
    package observer: Feature.f1
    package objects: Feature.f2
    package dummy: Feature.f3
    class Main: Feature.f4
−hypermodules
    hypermodule BorderDemo
```

```
    hyperslices: Feature.f1, Feature.f2, Feature.f3, Feature.f4;
    relationships: mergeByName;
end hypermodule;
```

2.2 Border Control

In its AspectJ implementation, the Border Control pattern defines an alternative application partitioning view by a set of pointcuts. By addressing these pointcuts in advices, the application functionality may be altered according to this new partitioning. A general description of Border Control in Coplien's form is as follows:

Problem: There is a need to operate upon the application partitioning other than the existing one.

Context: Crosscutting concerns that define the functionality and the unsatisfactory existing application partitioning.

Forces: A new application partitioning is needed, but the existing one may not be altered.

Solution: To introduce sets of join points that can specify an application partitioning that is needed.

Resulting Context: The existing partitioning is preserved, while crosscutting concerns may operate upon the new partitioning.

Rationale: This pattern allows to develop different views of the application and to design concerns at early stages of development when its structure is not yet fully known.

In Hyper/J, standard means can be used to explicitly define the concerns. Hyperslices and hypermodules are intended to define regions that crosscutting concerns can operate on. In the example that follows, classes are added into the partitions defined by hyperslices called Feature.f1 and Feature.f2:

```
−concerns
    class package1.Class1: Feature.f1
    class package1.Class3: Feature.f1
    class package2.Class2: Feature.f2
    class Main: Feature.f3
    package package3: Feature.f4
    package package4: Feature.f5
```

The bracket command in the composition file plays a similar role as an advice in AspectJ. The additional someMethod() method is executed after the execution of methods that match the pattern. An additional method is executed only within methods which belong to the Feature.f1 hyperslice:

```
bracket "∗"."exe∗"
    from hyperslice Feature.f1
    after Feature.f5.Add.someMethod;
```

2.3 Cuckoo's Egg

The AspectJ implementation of the Cuckoo's Egg pattern captures a constructor call by an around advice and creates and provides an object of another type. A general description of Cuckoo's Egg in Coplien's form is as follows:

Problem: Instead of an object of the original type, under certain conditions, an object of some other type is needed.
Context: The original type may be used in various contexts. The need for the object of another type can be determined before the instantiation takes place.
Forces: An object of some other type is needed, but the type that is going to be instantiated may not be altered.
Solution: Make the other type subtype of the original type and provide its instance instead of the original type instance at the moment of instantiation if the conditions for this are fulfilled.
Resulting Context: The original type remains unchanged, while it appears to give instances of the other type under certain conditions. There may be several such types chosen for instantiation according to the conditions.
Rationale: The other type has to be a subtype of the original type.

In Hyper/J, the Cuckoo's Egg pattern can be implemented by the standard concern manipulation features. The change is done by physical replacement of the original class in composition. The specification of concerns in composition file is as follows:

```
−concerns
  class Egg: Feature.egg
  class CuckoosEgg: Feature.cuckoo
  class Nest: Feature.nest
```

Since the names of classes to be merged are different, the replacement must be defined explicitly:

```
override class Feature.egg.Egg with class Feature.cuckoo.CuckoosEgg;
```

As we can see, the shortcoming of this approach is that the conditions of replacement can't be as easily refined as in the asymmetric approach. However, this shortcoming can be reduced by a smart definition of hyperslices.

3 Pattern Composition

The previous section described a successful individual symmetric implementation of three aspect-oriented patterns: Border Control, Director, and Cuckoo's Egg. To check whether the pattern composability sustains among their symmetric implementations, a small study based around a simple Swing application has been developed. First, an asymmetric version has been implemented in AspectJ with all three patterns applied in a composition. Subsequently, the patterns have been implemented symmetrically in Hyper/J demonstrating the sustaining ability of their symmetric implementations to be composed.

The application consists of two types of widgets: a frame with a slider and a frame with text labels. The functionality is very simple: as a user moves the sliders, the text labels display their status by a number within 0–100. The SlFrame class represents the basic frame with a slider inside of it:

```
public class SlFrame extends JFrame {
   private SlPanel SlPanel;
   public SlFrame() {
      this.getContentPane().add((JPanel) getSlPanel());
      ...
   }
   public abstract JPanel getSlPanel();
}
```

SlPanel is a simple panel with a horizontal slider. AnotherPanel is panel with a vertical slider. The Display class (implementation omitted here) shows the status of sliders. There are three packages named red, green, and blue, each of which contains a class derived from SlFrame: RedFrame, GreenFrame, and BlueFrame.

3.1 Asymmetric Implementation

An asymmetric implementation in AspectJ has been developed first. The Border Control pattern has been used to define an application partitioning according to red, green, and blue package.

The Cuckoo's Egg pattern has been used to swap an SlPanel instance by an AnotherPanel instance in the BlueFrame class (supposedly needed there) using the pointcut defined by the Border Control pattern instance.

The Director pattern has been used to enforce the Observer pattern onto the Display class and panel classes without having to alter their code. Since the Director instance affects also AnotherPanel, it forms a composition with a Cuckoo's Egg instance.

3.2 Symmetric Implementation

A symmetric implementation has been performed in Hyper/J. The Border Control pattern is realized simply by concern mappings. Partitionings are divided into hyperslices:

```
package red: Feature.red
package green: Feature.green
package blue: Feature.blue
package main: Feature.main
package gui: Feature.gui
package panelswap: Feature.panelswap
```

The Cuckoo's Egg pattern swaps the Feature.blue hyperslice defined by a Border Control instance using the **override** statement:

```
override hyperslice Feature.blue with hyperslice Feature.panelswap;
```

The additional roles for the Director pattern are defined in separate hyperslices in a similar manner as in Sect. 2.1:

```
package observer: Feature.observer
package dummy: Feature.dummy
```

The observer hyperslice contains the subject and observer interface:

```
public interface Subject {
   public void attach(Observer o);
   public void detach(Observer o);
   public void notify();
}
public interface Observer {
   void update(Subject subject);
}
```

The dummy hyperslice redeclares the BlueSlPanel, SlPanel, and Display class so that BlueSlPanel and SlPanel implement the Subject interface, while Display implements the Observer interface (the same way as explained in Sect. 2.1).

4 Discussion

Although it has been possible to develop a fully functional composition of symmetric pattern implementations, there are some differences with respect to the asymmetric composition worth mentioning. One of them is related to the way the Cuckoo's Egg and Border Control pattern are composed. As has been explained in Sect. 2.3, the symmetric implementation of Cuckoo's Egg replaces the whole class, unlike the asymmetric implementation in which only class instances are replaced at the instantiation time.

In its composition with Border Control, Cuckoo's Egg was intended to replace the corresponding class in one hyperslice only with its former version in other hyperslices preserved. In the asymmetric composition, the class to be replaced was shared between two areas of interest defined by pointcuts, but since in Hyper/J it is not possible for two or more hyperslices to share common classes, the class that was going to be replaced by Cuckoo's Egg had to be physically copied into the corresponding hyperslice.

Since the Hyper/J manual [12] mentions no such restriction, we assume this is not an inherent feature of Hyper/J. The copying was used just as a workaround to achieve a working implementation in Hyper/J. Speaking in the context of the example from Sect. 3.2, the appropriate solution was to map the SlPanel class from the gui package to the additional hyperslice denoted as blue as follows:

```
package gui: Feature.gui
package blue: Feature.blue
class gui.SlPanel: Feature.blue
```

Also, since the last available version of Hyper/J still didn't support a fully functional explicit class composition as declaerd in the manual [12], the Slpnel class had to be renamed (to BlueSlPanel) so it would not have been replaced by

mergeByName compositions. We found that an explicit override such as the one in Sect. 2.3 actually works with a simple code.

Some aspect-oriented patterns could not be implemented in Hyper/J. Both Exception Introduction and Worker Object Creation capture dynamic join points which are not supported by Hyper/J. The Policy pattern captures join points that occur during compile time, while Hyper/J composes previously compiled classes. The Wormhole pattern is based on capturing a control flow, which can't be done in Hyper/J.

5 Related Work

The Observer pattern implementation in CaesarJ [4] is actually realized by the Director pattern. Instead of intertype declarations, CaesarJ provides the component composition, which is symmetric. Components are wrapped into wrapping classes which encapsulate different concerns. This kind of encapsulation is similar to encapsulation of subjective classes in Hyper/J by hyperslices. However, this kind of implementation is still not purely symmetric because an asymmetric feature has to be used: an advice.

Kiczales and Haneman discussed the sustaining composability of aspect-oriented implementations of GoF design patterns [6].

Universality of aspect-oriented patterns is also examined by Hannenberg and Constanza [5]. They also discussed the difference between (general) design patterns and strategies specific to AspectJ and suggested the idea that, contrary to strategies, design patterns can be expressed in Alexander's form. In this paper, this idea has been examined further by expressing aspect-oriented design patterns in Coplien's form, a derivative of Alexander's form.

6 Conclusions and Further Work

Currently, asymmetric aspect-oriented approach is undoubtedly the mainstream in programming, but this doesn't mean that symmetric aspect-oriented approach is unimportant. It has its place in aspect-oriented modeling and from there it can affect the future (of) aspect-oriented languages. From this point of view, it may be worthwhile exploring whether aspect-oriented design patterns known only in their asymmetric implementation are also valid in this other branch of aspect-orientation.

This paper presents three aspect-oriented patterns—Director, Cuckoo's Egg, and Border Control—described in Coplien's form and implemented symmetrically in Hyper/J. Sustaining composability of the symmetric implementations of these patterns has been demonstrated by a small study that required the composition of these patterns.

Although it has been possible to develop a fully functional composition of symmetric pattern implementations, there have been some differences with respect to the asymmetric composition worth mentioning related mainly to the

way Cuckoo's Egg and Border Control are composed and limitations of the last available Hyper/J version.

Beyond the patterns presented here, some other patterns could not be implemented in Hyper/J: Exception Introduction, Worker Object Creation, Policy, nor Wormhole.

Difficulties with Hyper/J and lack of another symmetric aspect-oriented language brings to mind to base further experiments with symmetric aspect-oriented implementation of design patterns and in general on an emulated symmetric aspect-oriented programming in AspectJ. This is inspired by Jacobson's and Ng's implementation of peer use cases in which aspects act as partial classes with each one bringing its elements into the base implementation expressed as an empty class [8]. Advices could be then used to merge or override methods in the sense of Hyper/J while refraining from their asymmetric use.

Acknowledgements This work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09.

This contribution/publication is also a partial result of the Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] Alexander, C.: The Timeless Way of Building. Oxford University Press (1979)
[2] Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley (2005)
[3] Coplien, J.O.: Design pattern definition. `http://www.hillside.net/component/content/article/50-patterns/222-design-pattern-definition`
[4] Darmstadt, T.U.: CaesarJ documentation. `http://caesarj.org/index.php/ProgrammingGuide/`
[5] Hanenberg, S., Constanza, P.: Connecting aspects in aspectj: Strategies vs. patterns (2002)
[6] Hanneman, J., Kiczales, G.: Design pattern implementation in java and aspectj. In: 17th conference on object-oriented programming, systems languages anf applications(OOPSLA). pp. 1491–1497 (Oct 2002)
[7] Harrison, W., Ossher, H., Tarr, P.: Assymetrically vs. symmetrically organized paradigms for software composition (Dec 2002)
[8] Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley (2004)
[9] Kiczales, G., et al.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) Proc. of 11th European Conference on Object-Oriented Programming (ECOOP'97). LNCS 1241, Springer, Jyväskylä, Finland (Jun 1997)
[10] Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., Greenwich, CT, USA (2003)
[11] Miles, R.: AspectJ Cookbook. O'Reilly (2004)
[12] Tarr, P., Ossher, H.: Hyper/J User and Instalation manual. IBM Research (2000)
[13] Vranić, V.: Towards multi-paradigm software development. Journal of Computing and Information Technology (CIT) 10(2), 133–147 (2002)