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

Jaroslav Bálik **Valentino Vranić**

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava

vranic@fiit.stuba.sk

ESCOT 2011 – July 25, 2011, Lancaster, UK

Overview

1. Symmetry of Aspect-Oriented Approaches
2. Aspect-Oriented Design Patterns
3. Pattern Composition

Asymmetric and Symmetric AOP

- Asymmetric AOP: *aspects* (on one side) as something that affects the *base code* (on the other side)
  - Aspects are said to be woven into the base code
  - AspectJ and like—PARC[1] AOP
  - Mainstream approach in AOP
- Symmetric AOP: aspects as partial *views* of classes
  - Functional classes are constructed by the compositions of selected *views*, i.e. aspects
  - Hyper/J—IBM Watson Research Center
  - No industry-strength languages

[1] Palo Alto Research Center

A More Comprehensive View of Symmetry

- Symmetry here is percieved mostly as element symmetry
- A more comprehensive view of symmetry includes *join point symmetry* and *relationship symmetry*[2]

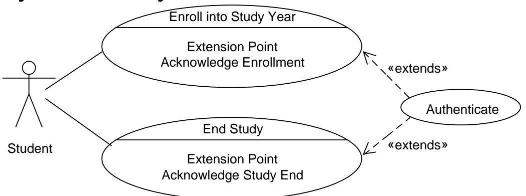
[2] W. Harrison, H. Ossher, P. Tarr. Assymetrically vs. symmetrically organized paradigms for software composition. Research Report RC22685, IBM Watson Research Center, 2002.

Significance of Symmetric Approaches

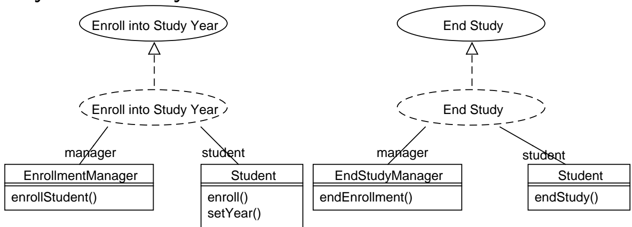
- Asymmetric approaches may lead to avoidance of aspect-orientation for the base design
- Symmetric approaches seem to be important in aspect-oriented analysis and design
- Theme—an academic approach that supports symmetry in analysis and design

Use cases—Intrinsically Aspect-Oriented

- Symmetrically



- Asymmetrically



Aspect-Oriented Design Patterns

- Aspect-oriented reimplementations of object-oriented design patterns are sometimes denoted as aspect-oriented design patterns
- But there are intrinsic aspect-oriented patterns
- Aspect-oriented patterns are virtually defined by their AspectJ implementations—an asymmetric approach
- Are they intrinsically asymmetric?

Symmetric Pattern Form

- Symmetric aspect-oriented approaches are important
- A general aspect-oriented pattern should be possible to implement both asymmetrically and symmetrically
- Aspect-oriented patterns should be examined for the existence of their symmetric form
- Individual symmetric implementations of patterns are only the first step
- Their ability to be composed with each other should sustain among their symmetric implementations
- We did this for three patterns:
    - Director
    - Border Control
    - Cuckoo's Egg

Coplien's Form

- Coplien's form was used to express patterns in a general form, departed from (asymmetric) implementation details
- E.g., in AspectJ terms, the Cuckoo's Egg pattern captures a constructor call by an around advice and creates and provides an object of another type

Cuckoo's Egg in Coplien's Form

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.

Hyper/J Pattern Implementations

- Hyperspace: a set of classes to be operated upon
- Hyperslices (concerns): views containing partial classes
- Hypermodules: compositions of views into complete classes

Cuckoo's Egg in Hyper/J (1)

```
public class Nest {
    Egg e;
    public Nest() {
        e = new Egg();
    }
    public void report() {
        e.exec();
    }
}

public class Egg {
    public void report() {
        System.out.println("original egg");
    }
}

public class CuckoosEgg {
    public void report() {
        System.out.println("cuckoo's egg");
    }
}
```

Cuckoo's Egg in Hyper/J (2)

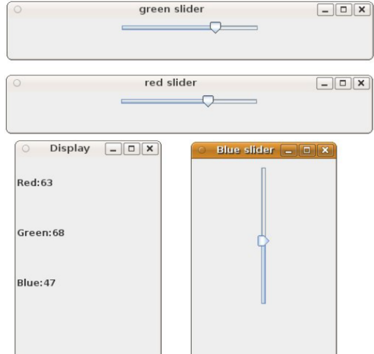
```
-hyperspace
   hyperspace cuckoosegg
   composable class Egg;
   composable class CuckoosEgg;
   composable class Nest;

-concerns
   class Egg: Feature.egg
   class CuckoosEgg: Feature.cuckoo
   class Nest: Feature.nest

-hypermodules
   hypermodule CuckooDemo
      hyperslices: Feature.egg, Feature.cuckoo, Feature.nest;
      relationships: overrideByName;
      override class Feature.egg.Egg
         with class Feature.cuckoo.CuckoosEgg;
   end hypermodule;
```

Cuckoo's Egg in AspectJ

```
public aspect PutCuckoosEgg {
    declare parents: CuckoosEgg extends Egg;
    Egg around(): : call(Egg.new(..)) {
        return new CuckoosEgg();
    }
}
```

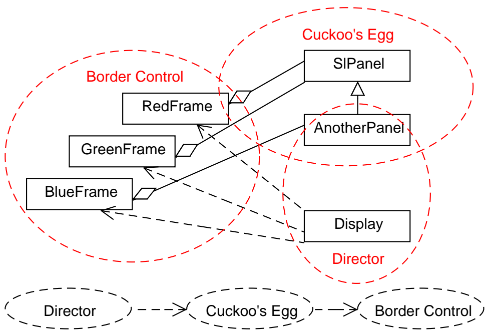
A Study (1)

- A small study that involved a composition of three aspect-oriented patterns has been developed
- The state of three horizontal RGB color sliders is observed by a display
- The blue slider is replaced by a vertical slider



A Study (2)

- The pattern composition has been implemented in AspectJ and in Hyper/J



Asymmetric Pattern Composition in Hyper/J

- Border Control defines a partitioning according to color packages
- Cuckoo's Egg swaps an SlPanel instance by an AnotherSlPanel instance in the BlueFrame class using the pointcut defined by Border Control
- Director enforces the Observer pattern onto the Display class and panel classes affecting AnotherPanel, too

Symmetric Pattern Composition in Hyper/J (1)

- Border Control's partitioning is realized by concern mappings

  package red: Feature.red
  package green: Feature.green
  ...

- Cuckoo's Egg swaps the Feature.blue hyperslice defined by Border Control using the override statement

  override hyperslice Feature.blue with hyperslice Feature.panelswap;

Symmetric Pattern Composition in Hyper/J (2)

- Director defines its additional roles in separate hyperslices

```
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 classes so that
    - BlueSlPanel[3] and SlPanel implement the Subject interface
    - Display implements the Observer interface

[3] playing the role of AnotherPanel

Hyper/J Limitations

- A class can't belong to different hyperslices
  - The class that was going to be replaced by Cuckoo's Egg had to be physically copied into the corresponding hyperslice
- No fully functional explicit class composition (known issue)
  - Explicit override actually works, but only in simple cases; the problem arises when combined with mergeByName
  - SlPanel in the blue hyperslice (its copy) had to be renamed (to BlueSlPanel) so it would not have been replaced by mergeByName compositions
- These limitations are rather technical, not intrinsic to the symmetric approach as such

Patterns That Couldn't Be Implemented in Hyper/J

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

Summary

- Symmetric aspect-oriented approaches are important
- Aspect-oriented design patterns should be examined for the existence of their symmetric form
- Validity of the symmetric pattern forms can be checked empirically by their sustaining ability to be composed