Combination of Aspect-Oriented Design Patterns

Erasmus Mobility at Lancaster University

Lecture 2

Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology
Bratislava, Slovakia
vranic@fiit.stuba.sk
http://fiit.stuba.sk/~vranic/

September 16–19, 2008

Overview

1. Introduction
2. Generally Applicable Aspect-Oriented Change Types
3. Structural Categorization of Aspect-Oriented Design Patterns
4. Combining Aspect-Oriented Design Patterns
5. Regularity in Aspect-Oriented Design Pattern Combination
6. Related Work
7. Summary

Introduction

- The notion of pattern in its original sense proposed by Alexander was indivisible of the notion of pattern language[1]
- Software patterns are often perceived as more or less independently applied sublimated pieces of development experience[2]
- We tend first to discover new patterns and then think of the opportunities of their combination
- This also applies to aspect-oriented design patterns being discovered both on individual basis[3] [4] [5] and as pattern languages [6]

[1] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] J. O. Coplien. The culture of patterns. *Computer Science and Information Systems (ComSIS)*, 1(2), November 2004.

[3] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[4] R. Miles. *AspectJ Cookbook*. O'Reilly, 2004.

[5] A. Schmidmeier. Patterns and an antiidiom for aspect oriented programming. In *Proc. of 9th European Conf. on Pattern Languages of Programs, EuroPLoP 2004*, Irsee, Germany, July 2004.

[6] S. Hanenberg, A. Schmidmeier, and R. Unland. Aspectj idioms for aspect-oriented software construction. In *Proc. of 8th European Conf. on Pattern Languages of Programs, EuroPLoP 2003*, Irsee, Germany, June 2003.

The Catalog of Changes in Web Application Domain (1)

- Integration Changes
  - One Way Integration: Performing Action After Event
  - Two Way Integration: Performing Action After Event
- Grid Display Changes
  - Adding Column to Grid: Performing Action After Event
  - Removing Column from Grid: Method Substitution
  - Altering Column Presentation in Grid: Method Substitution

The Catalog of Changes in Web Application Domain (2)

- Input Form Changes
  - Adding Fields to Form: Enumeration Modification with Additional Return Value Checking/Modification
  - Removing Fields from Form: Additional Return Value Checking/Modification
  - Introducing Additional Constraint on Fields: Additional Parameter Checking or Performing Action After Event
- Introducing User Rights Management: Border Control with Method Substitution
- User Interface Restriction: Additional Return Value Checking/Modifications
- Introducing Resource Backup: Class Exchange

Generally Applicable Aspect-Oriented Change Types

- Performing Action After Event
- Method Substitution
- Enumeration Modification
- Additional Return Value Checking/Modification
- Additional Parameter Checking
- Performing Action After Event
- *Border Control*
- Class Exchange = *Cuckoo's Egg*

Code Schemes or Patterns?

- What is a pattern?
- Some proclaimed aspect-oriented design patterns are quite simple (consider Border Control)
- Couldn't the generally applicable change types be considered as patterns?
- Patterns or idioms?

Aspect-Oriented Programming and AspectJ

- Crosscutting concerns are implemented as aspects
- Variety of aspect-oriented approaches and languages
- AspectJ is the most widely used and influential aspect-oriented language
- The key issue is to identify and specify places where the crosscutting code affects the rest of the code
- Such places are called *join points* and they are specified by *pointcuts*
- Additional behavior to be performed before, after, or instead of join points is specified in *advices*
- *Inter-type declarations* enable introduction of new members into existing types, as well as introduction of compile warnings and errors

Aspect-Oriented Design Patterns

- A closer look at the structure of four particular aspect-oriented design patterns
- Structural categorization of aspect-oriented design patterns
- Applying the combination of aspect-oriented design patterns to the problem of class deprecation in team development
- Generalize the dependencies between aspect-oriented design patterns in their subsequent application

Border Control

- Used to define regions in the application
- The regions are intended for use by other aspects to ensure they are applied only to appropriate places
- This is convenient also when the system changes are expected
  - Only declarations of regions in the Border Control aspect should be changed
  - Other aspects which are using these declarations will be automatically redirected

```
public aspect MyRegionSeparator {
    public pointcut myTypes1(): within(mypackage1.+);
    public pointcut myTypes2(): within(mypackage2.+);
    public pointcut myTypes(): myTypes1() || myTypes2();
    public pointcut myMainMethod():
        withincode(public void mypackage2.MyClass.main(..));
    . . .
}
```

Cuckoo's Egg

- Enables to put another object instead of the one that the creator expected to receive (similar to what a cuckoo does with its eggs)

```
public aspect MyClassSwapper {
    public pointcut myConstructors():
        call(MyClass1.new()) || call(MyClass2.new());

    Object around(): myConstructors() {
        return new AnotherClass();
    }
}
```

- Several types can be covered by swapping
- Swapping can be restricted with respect to the place of construction
- Note: the swapping object must be a subtype of the original object class

Policy

- Defines a policy or rules within the application
- Breaking of such a rule or policy should result in issuing a compiler warning or error
- Useful in projects that involve many developers

```
public abstract aspect GeneralPolicy {
    protected abstract pointcut warnAbout();

    declare warning: warnAbout(): "Warning...";
}

public aspect MyAppPolicy extends ProjectPolicy {
    protected pointcut warnAbout():
        call(* *.myMethod(..)) || call(* *.myMethod2());
}
```

Exception Introduction (1)

- If an advice calls a method that throws a checked exception, it is forced to cope with it
- Sometimes, it is not possible to handle the exception in the advice, so it has to be thrown to a higher context
- But in AspectJ an advice cannot declare throwing a checked exception unless the advised joint point declared this exception, which is unlikely since base concerns are mostly not expected to be adapted to their aspects

Exception Introduction (2)

- Exception Introduction[7] catches a checked exception and wrappes it into a new concern-specific runtime exceptions

```
public abstract aspect ConcernAspect {
    abstract pointcut operations();

    before(): operations() {
        try {
            concernLogic();
        } catch (ConcernCheckedException ex) {
            throw new ConcernRuntimeException(ex);
        }
    }
    void concernLogic() throws ConcernCheckedException {
        . . .
    }
}
```

[7] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*

Aspect-Oriented Design Pattern Categories

- Each aspect-oriented design pattern comprises at least one aspect
- In the aspects of each pattern one of the three main parts of an aspect, i.e. a pointcut, advice, or inter-type declarations, prevails in achieving the purpose of the pattern
- According to the element that dominates, aspect-oriented design patterns can be divided into:
  - Pointcut patterns
  - Advice patterns
  - Inter-type declaration patterns

Pointcut Patterns

- Border Control is a pointcut pattern
- It actually contains no other elements than pointcuts
- Other examples:[8]
- Wormhole employs pointcuts to connect a method callee with a caller so they can share their context information
- Participant enables a class decide whether it will allow an aspect to affect it by declaring an appropriate pointcut

[8] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*

Advice Patterns

- Cuckoo's Egg is an advice pattern
- Other examples:[9]
  - Worker Object Creation—aka Proceed Object[10]—captures the original method execution into a runnable object to be manipulated further
  - Exception Introduction resolves inability of advices in AspectJ to throw checked exceptions

[9] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*

[10] A. Schmidmeier. Patterns and an antiidiom for aspect oriented programming. In *Proc. of 9th European Conf. on Pattern Languages of Programs, EuroPLoP 2004*, Irsee, Germany, July 2004.

Inter-Type Declaration Patterns

- Policy is an inter-type declaration pattern
- Another example is Default Interface Implementation[11] which employs inter-type declarations to introduce fully implemented methods into interfaces.
- Denoted as idiom by Laddad

[11] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*

Invoking an Example: Class Deprecation

- Team development requires developers to obey some common rules and policies
- Frequently, a new version of a class is introduced into the framework
- The old version of the class cannot be simply replaced with the new one at once
- A new class has to be tested for some time during which it is common to have and use both versions
- All developers should be kept informed of the new class version and warned—or sometimes even forced—to use it
- The best way is to incorporate this information into the build process (compilation warnings and errors)
- Sometimes a new version should swap the old one

Class Deprecation Warning

- Suppose the instantiation of OldClass is deprecated
- Assume it is sufficient to issue a warning in case of deprecated class instantiation
- Policy can be applied to achieve this

```
public aspect OldClassDeprecation {
    declare warning: call(*.OldClass.new()): "Class OldClass deprecated.";
}
```

- AspectJ 5 supports declaring annotations, so a standard @deprecated annotation can be introduced instead of a general warning with a custom message

```
public aspect OldClassDeprecation {
    declare @constructor: OldClass.new(): @deprecated;
}
```

- This approach was not used because annotations declarations are made on type patterns, not pointcuts, which poses significant limitations

Partial Class Deprecation (1)

- Assume OldClass can be used within the testing package and third party code
- Border Control can be applied to partion the code into regions

```
public aspect Regions {
    public pointcut Testing(): within(com.myapplication.testing.+);
    public pointcut MyApplication(): within(com.myapplication.+);
    public pointcut ThirdParty(): within(com.myapplication.thirdpartylibrary.+);
    public pointcut ClassSwitcher(): within(com.myapplication.ClassSwitcher);
}
```

- Afterwards, OldClassDeprecation aspect has to be adapted:

```
public aspect OldClassDeprecation {
    protected pointcut allowedUse():
        Regions.ThirdParty() || Regions.Testing();

    declare warning:
        call(Display.new()) && !allowedUse(): "Class OldClass deprecated.";
}
```

Partial Class Deprecation (2)

- By this, we actually combined a Policy with an existing Border Control
- Thus, combining a pointcut pattern with an existing inter-type declaration pattern requires changes in the existing inter-type declaration pattern
- If we knew there will be exemptions from banning the use of OldClass, it would be possible to apply Border Control first with Policy pattern added without having to change the existing code
- Thus, combining an inter-type declaration pattern with an existing pointcut pattern can be performed without having to change the existing pattern

Class Swapping

- Now we want to make a change from OldClass to NewClass automatic while keeping developers informed of attempts to instantiate OldClass in prohibited regions
- This may be achieved with Cuckoo's Egg

```
public aspect OldClassDeprecation {
    public pointcut oldClassConstructor():
        call(*.OldClass.new()) &&
        !Regions.ThirdParty() && !Regions.Testing();

    Object around(): oldClassConstructor() {
        return new MyApplication.NewClass();
    }
}
```

- Cuckoo's Egg uses the pointcuts defined in Border Control
- Thus, combining an advice pattern with an existing pointcut pattern can be made without changes in the existing pointcut pattern

Adding Logging (1)

- Assume there is a need to log the swapping of the deprecated class with the new one
- It looks simple: the logging code could be simply added to the Cuckoo's Egg advice
- But we have to deal with IOException in the advice, and an it can't declare throwing of an exception that was not declared by the advised join point

Adding Logging (2)

- Exception Introduction can resolve this

```
public class SwitchLoggingException extends RuntimeException {
    public SwitchLoggingException(Throwable cause) { super(cause); }
}

public aspect SwitchLogging {
    before(): adviceexecution() && Regions.ClassSwitcher() {
        try {
            logSwapEvent()
        } catch(IOException e) {
            throw new SwitchLoggingException(e);
        }
}
```

- Exception Introduction was added to the existing Cockoo's Egg without having to modify it
- It also uses pointcuts from the already applied Border Control

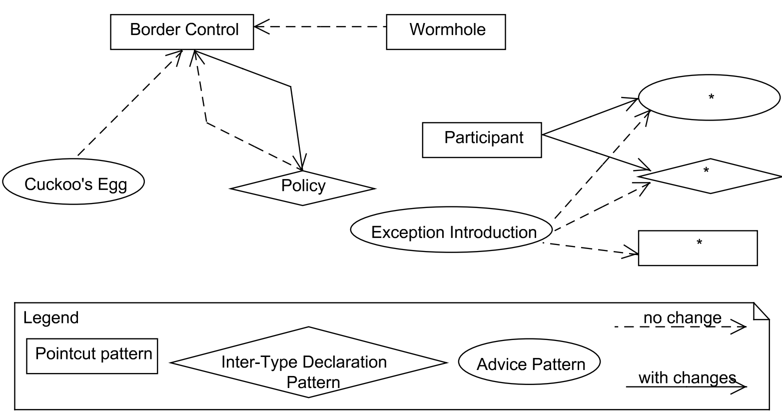
What Combinations Require Changes? (1)

- Combination of aspect-oriented design patterns is substantially affected by their structural category
- Under a combination of two patterns we understand a subsequent interrelated application of two patterns to a problem at hand
- Thanks to the crosscutting nature of aspects, most aspect-oriented design patterns can be combined with other patterns without the need to modify the already applied patterns
- Exception Introduction can simply be added to the program without having to make any change to already applied patterns

What Combinations Require Changes? (2)

- Policy can be used with other, already applied patterns without having to make any changes to them
- It is also possible to combine a pointcut pattern with another pattern of the same type without having to change that pattern (Wormhole and Border Control)
- However, combining a pointcut pattern with an already applied advice or inter-type declaration pattern usually requires a change of this pattern (Policy and Border Control)
- If we go the other way around, i.e. if we combine an inter-type declaration pattern (e.g., Policy) or advice pattern (e.g., Cuckoo's Egg) with an already applied pointcut pattern (e.g., Border Control), this can be done without having to change them
- Also, if we combine a non-pointcut pattern with Participant, it has to be altered

What Combinations Require Changes? (2)



Related Work (1)

- Hanenberg et al. present a set of AspectJ idioms and a scheme for their interrelated application[12] (similarly to the well-known scheme of GoF patterns)
- However, no attempt is made to categorize the idioms
- Analogy between categories proposed here and those proposed by Gamma et al. (behavioral, structural, and creational)[13]
  - Advice patterns recall behavioral patterns since they affect behavior
  - Pointcut patterns deal with how aspects are composed with classes, objects, and other aspects, which is a paraphrase of the description of structural patterns
  - Inter-type declarations correspond to creational patterns to a lesser extent, but we may see them as patterns of creating new elements and relationships

[12] S. Hanenberg, A. Schmidmeier, and R. Unland. Aspectj idioms for aspect-oriented software construction. In *Proc. of 8th European Conf. on Pattern Languages of Programs, EuroPLoP 2003*, Irsee, Germany, June 2003.

[13] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Related Work (2)

- Class deprecation in team development can be seen as a change and as such it is related to a broader area of change control
- Aspects can be used to capture change[14] [15]
- Captured in an aspect, a change becomes pluggable and reapplicable

[14] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog. Developing Applications with Aspect-Oriented Change Realization. Accepted to 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008, October 2008, Brno, Czech Republic.

[15] M. Bebjak, V. Vranić, and P. Dolog. Evolution of web applications with aspect-oriented design patterns. In M. Brambilla and E. Mendes, editors, *Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007*, Como, Italy, July 2007.

Summary

- A categorization of aspect-oriented design patterns has been presented
- Three categories: pointcut, advice, and inter-type declaration patterns
- This categorization can be used in determining whether a combination of an aspect-oriented design pattern with another, already applied pattern requires a change in this pattern
- Combination of aspect-oriented design patterns of different categories presented on the class deprecation problem in team development
- Combination of Policy, Border Control, Cuckoo's Egg, and Participant
- Further work involves exploring the possibilities of employing aspect-oriented design patterns and their combinations in capturing changes in a pluggable and reapplicable way