Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMACNÝCH TECHNOLÓGIÍ
Študijný program: Informatika

Radoslav Menkyna

Aspektovo-orientované návrhové vzory

Klasifikácia a kombinácia

Bakalársky projekt

Vedúci bakalárskeho projektu: Ing. Valentino Vranić, PhD.
Máj 2007

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Study program: Informatics

Radoslav Menkyna

Aspect-Oriented Design Patterns

Classification and Combination

Bachalor‘s Thesis

Supervisor: Ing. Valentino Vranić, PhD.
May 2007

PREHLÁSENIE

Vedúci bakalárskeho projektu: Ing. Valentino Vranić, PhD.
Máj 2007

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne.

V Bratislave, 10 mája 2007

Radoslav Menkyna

ANOTÁCIA

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMACNÝCH TECHNOLÓGIÍ
Študijný program: Informatika

Autor: Radoslav Menkyna

Bakalársky projekt: Aspektovo-orientované návrhové vzory: Klasifikácia a kombinácia

Vedenie bakalárskeho projektu: Ing. Valentino Vranić, PhD.

Máj 2007

Táto práca predstavuje klasifikáciu aspektovo-orientovaných návrhových vzorov podľa štruktúry aspektu, ktorý reprezentuje daný návrhový vzor. Návrhové vzory je možné zatriediť do troch kategórií: návrhové vzory vnútro-typových deklarácií, návrhové vzory bodových prierezov a návrhové vzory videní. Práca identifikuje určitú závislosť medzi štrukturálnou kategóriou návrhového vzoru a jeho schopnosťou kombinácie. Ak poznáme štrukturálne typy návrhových vzorov, môžeme povedať, či kombinácia vzoru s už aplikovaným vzorom, vyžaduje alebo nevyžaduje zmenu už aplikovaného vzoru. Kombinácia je študovaná podrobne na štyroch aspektovo-orientovaných návrhových vzoroch, ktoré zastupujú všetky štrukturálne kategórie, v kontexte problému zavrhnutých tried. Zistená pravidelnosť v kombinácii aspektovo-orientovaných návrhových vzorov bola analyzovaná a zistené poznatky boli prezentované.

ANNOTATION

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Study program: Informatics

Author: Radoslav Menkyna

Bachelor's Thesis: Aspect-oriented design patterns: Classification and Combination

Supervisor: Ing. Valentino Vranić, PhD.

May 2007

This report proposes a classification of aspect-oriented design patterns according to the structure of an aspect that represents a particular design pattern and explores its role in pattern combination. Patterns can be classified into three categories: pointcut design patterns, advice design patterns, and inter-type declaration design patterns. Certain dependence between the design pattern structural class and its combination ability has been identified. Knowing a structural category of the patterns, it is possible to say whether a sequential combination of the pattern with a already applied pattern can be made with or without having to change the already applied design pattern. The combination is studied in detail on four aspect-oriented design patterns, which cover all the structural categories, in the context of class deprecation problem. Identified regularity in the combination of aspect oriented design patterns was discussed and summarized.

ACKNOWLEDGMENTS

I would like to thank Valentino Vranić for numerous discussions that have been very valuable and for his comments which helped me a lot to put the thesis into its final form.

Contents

Chapter 1

Introduction

In the field of software engineering progress is very fast.[1] New methods and techniques appear every day with common goal: to make software better. Better as quicker, more efficient, smaller, easier to write, reusable, etc. Using object-oriented design patterns has proved all this can be accomplished. Now aspect-oriented paradigm is here and new design pattern are emerging in it already. This work will try to take closer look at these patterns, classify them and study their combination. This chapter will provide a definition of the main terms and an introduction to aspect-oriented paradigm. It also summarizes the organization of following chapters.

1.1 Design Patterns

Patterns first emerged as a concept in architecture. An architect Christopher Alexander once noticed that two or more different projects share the same problems. The core of these problems is always the same. The solution to these problems will be similar. The idea is to find universal solution that can be then altered according our needs in other worlds to find a pattern.[Ale79a] This is the definition of a pattern by Christopher Alexander:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Ale79b].

Patterns were adopted by software engineering and found great use in object-oriented paradigm [Cop04]. Design pattern is a solution to some

[1] This report builds on the adapted text of my two papers that have preceded it. I am the only author of the paper in Appendix A [Men07], and my contribution to the paper in Appendix B [MV] is approximately 70 %.

kind of problem repeated through a code or through different projects. It tries to be a general solution to the set of similar design problems. Design patterns have to be customized to be used in a particular application. This is summarized in a definition by GoF:[2]

> "Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [GHJV95]."

Design patterns should not depend on details of a particular programming language, so different languages of the same paradigm can be used to implement them. By modularization the design pattern helps to increase reusability of code and makes it simpler to understand.

A design pattern follows a certain template which was created by Christopher Alexander. Template was then adapted to be used by software design patterns by GoF. This template describes the problem design pattern refers to, its solution, and possible consequences of using the pattern.

1.2 Idioms

Sometimes structures smaller than design patterns called idioms can be recognized. These structures also provide a solution to some kind of problem, but at a lower level. They are often language dependant and their nature usually comes from the syntax of a particular language. However, they are crucial to design patterns, because the combination of these idioms coupled with their generalization can lead to a design pattern.

This can be best seen on one pattern of GoF design patterns called the Visitor. A key to this pattern is the double dispatch technique. This technique is directly supported by, for example, CLOS. Thus, to CLOS, double dispatching was not a design pattern but an idiom. An idiom can then be generalized to become design pattern which can be used also in other languages [GHJV95].

Some fragments of code which are treated like idioms by part of the community are seen as antiidioms by others. An antiidiom is an idiom which should solve particular problem, but it does not solve the problem in full scope. For example, according to Arno Schmidmeier [Sch04] a common idiom Not Within is an antiidiom. This idiom should solve the problem with the infinite recursion, but it solves only a subset of this problem and examples where the infinite recursion occurs can be found.

[2] Gang of Four or GoF which stands for Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

1.3 Aspect-Oriented Paradigm

Aspect-oriented paradigm builds a lot on object-oriented paradigm which is a current mainstream. This brings many advantages. For example, our knowledge of object-oriented paradigm can be used in it or the same development environments can be used. Aspect-oriented paradigm tries to find elegant solutions to problems which solution was too complicated in older paradigms.

The main focus is on crosscutting concerns. These concerns cut across through base concerns. They are creating undesirable dependencies among the classes and cause code tangling and scattering. This can complicate future development of software. Typical crosscutting concerns are, for example, logging or authentication. The main idea of aspect-oriented programming is to define these crosscutting concerns in a modular way at one place [KLM+97].

There are many implementations of the aspect-oriented paradigm. Today the most popular language is AspectJ. This language was developed by Gregor Kiczales and his team at Palo Alto Research Center (PARC). There is also implementation of this paradigm by IBM called HyperJ. This language is slightly more powerful, but didn't find so wide use. There are also many aspect-oriented extensions of other languages like C#, C or C++, Python, PHP and other. AspectJ syntax is generally known and is used as default in aspect-oriented community.

The basic terms defined in the PARC approach to aspect-oriented programing are:

Join points are points in execution of code where crosscutting may occur. Different languages support different join points. In general, join points can be all method calls or executions, constructor calls or execution, object initialization, field operations, etc.

A *pointcut* declares join points where crosscutting should occur. It simply describes a set of join points. Various operators can be used to make selections of join points. When the code execution reaches a join point declared in a pointcut advice bound with this pointcut is executed.

An *advice* is a piece of code to be executed at desired join points declared in the pointcut. Advice can be executed before, after, or around certain pointcut.

A *inter-type declaration* is a static crosscutting technique that allows to add fields, methods and interfaces into existing classes.

An *aspect* is a base construct of AspectJ. It is very similar to a class from in object-oriented paradigm. An Aspect encapsulates the whole logic of a crosscutting concern. From the point of structure it consists of three main components: inter-type declarations, pointcuts and advices. It can also define fields, methods or nested classes like ordinary Java class.

Crosscutting concerns defined by these constructs are then weaved into

the code of base concerns during compilation. AspectJ compiler produces standard Java byte code.

1.4 Report Organization

The rest of the article is organized as follows. Chapter 2 presents an overview of common aspect oriented design patterns. In Chapter 3 the classification according to structure of the design pattern is presented. Chapter 4 presents the case study of class deprecation problem which is solved by aspect-oriented design pattern combination. Chapter 5 summarizes the regularity between design pattern structure its combination ability. Chapter 6 is the conclusion of my work.

Chapter 2

Selected Aspect-Oriented Design Patterns

There are two major groups of design patterns implemented in AspectJ. First there are whole new aspect-oriented design patterns but there is also quite group of object-oriented design patterns reimplemented in this language. Several studies show benefits of this reimplementation [HK02, HB03]. In most cases, a better modularity of GoF patterns was achieved. Modularity simplifies the use of these patterns and increases their reusability. The best results were achieved with patterns that have crosscutting structure (e.g., Observer).

It is questionable whether these reimplementations are true aspect-oriented design patterns, because they are solutions to problems which came from the object-oriented paradigm. In this work, my main focus will be at a intrinsic aspect-oriented design patterns. These patterns try to solve inherent aspect-oriented problems.

In general, programmers have less real project experience with aspect-oriented paradigm, because fewer projects are implemented in aspect-oriented languages. Experience has the key role in discovering idioms and design patterns. Hanenberg [HC02] presents strategies of aspect-oriented programming and points out that these strategies could become design patterns. While strategies are very similar to design patterns, they can't be regarded as design patterns because some of them rely too much on AspectJ. Despite this, patterns are emerging in the aspect-oriented paradigm already. This means that some of strategies and idioms were generalized, do not relay on one particular language any more and became patterns. Following sections will present an overview of common aspect-oriented design patterns identified so far.

2.1 Wormhole

The Wormhole pattern [Lad03] connects the callee with caller in such a way, that they share their context information. It creates a direct connection between two levels in the call stack. This is very helpful when additional context information has to be added [Lad03]. Instead of adding extra parameters in each method in the control flow or using a global storage, this pattern can be used.

The pattern can be implemented with an aspect in which three pointcuts are defined: one for the caller, another for the callee, and the third one as the so-called wormhole pointcut. Pointcuts for the callee and the caller must collect the context which is transferred through the wormhole. The Wormhole defines the places of execution of the callee's join points in the control flow of the a caller's join points [Lad03].

This is a template for the Wormhole pattern adopted from [Lad03]:

```
public aspect WormholeAspect {
        pointcut callerSpace(<caller context>)
                : <caller pointcut>;

        pointcut calleeSpace(<callee context>)
                : <callee pointcut>;

        pointcut wormhole(<caller context>, <callee context>)
                : cflow(callerSpace(<caller context>))
                        && calleeSpace(<callee context>);

        // advices to wormhole
        around(<caller context>, <callee context>)
                : wormhole(<caller context>, <callee context>) {
                ... advice body
        }
}
```

2.2 Exception Introduction

In some cases when the aspect is used to implement some crosscutting concern checked exceptions have to be caught in its advice. This usually happens when methods of Java libraries, which declare to throw such exceptions, are used in its advice. In AspectJ,[1] advice cannot declare to throw a checked exception unless the advised joint point declared this exception.The Base concern logic cannot declare those exceptions because it simply does not know anything about the logic used in the crosscutting concern [Lad03].

The Exception Introduction pattern [Lad03] suggest that the checked exceptions should be caught and simply wrapped into a new concern-specific

[1] This problem is common to many languages of aspect-oriented paradigm.

runtime exceptions. Such exceptions can be then thrown to higher level, where they can be unwrapped and the real cause of exception revealed [Lad03]. This is a template for this pattern adopted from [Lad03]:

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

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

void concernLogic() throws ConcernCheckedException {
        throw new ConcernCheckedException(); // simulating failure
        }
}
```

This simple form of Exception Introduction pattern uses a runtime exception. But the base concern is not prepared to catch such exception. This problem solves the extended version of this pattern, which tries to preserve the exception specified by the base concern (a specific exception). This is achieved by adding another aspect.The advice of this aspect will be invoked after the ConcernRuntimeException is thrown by any method that declares that it may throw a specific exception. In the advice, a check whether the cause for runtime exception is the same as cause for specific exception has to be performed [Lad03].

Here is the template for the extended version adopted from [Lad03]:

```
public aspect PreserveSpecificException {
        declare precedence: PreserveSpecificException, SpecificConcernAspect;

        after() throwing(ConcernRuntimeException ex) throws SpecificException
                : call(* *.*(..) throws SpecificException) {
                        Throwable cause = ex.getCause();

                        if (cause instanceof SpecificException) {
                                throw (SpecificException)cause;
                        }
                        throw ex;
        }
}
```

2.3 Participant

Usually aspects try to introduce some behavior to the base concern in such a way that the base concern is not aware of the aspect. In this case the roles change. Aspect makes classes to participate.

This is needed for the purpose an aspect is trying to achieve. In some cases, defining pointcuts only by the means of language syntax is not sufficient. For example, if the advice should affect only methods with certain characteristics, one cannot decide only according to their names whether to include them in the pointcut or not. Only the creator of these methods knows their characteristics. This is where comes from the idea that classes themselves should express if they want to be advised. If they want to participate they simply define an appropriate pointcut in them [Lad03].

Here is the base schema of this pattern. First an abstract aspect is defined. In this aspect, an abstract pointcut is declared that represents the characteristics of methods to be advised. This pointcut is then defined in the each base concern class which wants to be advised. The desired crosscutting concern is defined in the advice. This aspect is called the invitation aspect.

Each class which wants to participate defines a concrete aspect extending the abstract invitation aspect. This means class provides a definition of the abstract pointcut for join points of its methods which have desired characteristics [Lad03].

A template for an abstract invitation aspect adopted from [Lad03]:

```
abstract aspect AbstractDesiredCharacteristicAspect {
        public abstract pointcut desiredCharacteristicJoinPoints();

        // Example uses around(), but before() and after() work as well
        Object around() : desiredCharacteristicJoinPoints() {
        // advice code
        }
}
```

Template for the participating class:

```
public class MyClass1 {
// MyClass1's implementation

public static aspect DesiredCharacteristicParticipant
        extends AbstractDesiredCharacteristicAspect {
        public pointcut desiredCharacteristicJoinPoints() :
                call(* MyClass1.desiredCharacteristicMethod1())
                || call(* MyClass1.desiredCharacteristicMethod2())
        }
}
```

2.4 Cuckoo's Egg

The Cuckoo's Egg Design pattern [Mil04] is quite simple but powerful. It expresses how powerful aspect-oriented programming can be. It is used to control or change the objects created by the constructor call. This means that with this pattern it is possible to change the type of the object being instantiated [Mil04].

The pattern uses an aspect in which pointcut specifies join points, usually those are constructor calls of objects, and an advice which changes the object being created or performs some control logic over it. [Mil04].

This pattern could be also used to implement the Singleton pattern. In this case, the advice would only check whether an object of original class has already been created or not.

Here is simple template adopted from [Mil04]:

```
public aspect ControlClassSelectionAspect{ // Cuckoo's Egg Aspect
    public pointcut originalClassConstructorCall( ) : <call pointcut>

    Object around( ) : originalClassConstructorCall( ){
        return new DesiredClass( );
    }
}
```

Note that the swapping object must be a subtype of the original object class; otherwise, we will get a class cast exception on the first attempt to instantiate (now swapped) the original class.

2.5 Director

The Director pattern [Mil04] can be used to define some roles or behavior to an unknown number of classes. A role can be defined without knowing the particular class it will be applied to. A pattern can be used to define some logic in an abstract aspect without knowing the classes this logic will be applied to. The Director can also implement some relationships within abstract entities [Mil04].

Two aspects are used to implement director design pattern. The first aspect is abstract and it should specify the roles. This can be done using Java interfaces. The second aspect introduces the roles to specific classes. It can also provide any required implementations of methods which are needed but not implemented in target classes [Mil04].

2.6 Border Control

The Border Control design pattern [Mil04] is used to define some reasonable regions in the application. These regions are later reused by other aspects

to ensure they are used only in correct scope. Use of this pattern is also convenient when the changes in structure of the application are expected. After such changes only declarations of regions in border control aspect are changed and other aspects which are reusing this declarations will be also affected [Mil04].

The pattern can be implemented as single aspect that defines the regions. Regions are represented as pointcuts. These pointcuts can be reused later in other aspects in the system.

Folowing code is a template of Border Control design pattern adapted from [Mil04]:

```
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(..));
...
}
```

2.7 Policy

The main idea of this pattern is to define some policy or rules within the application. The rules can wary from suggestions and warnings to overriding methods, classes or libraries. This is very useful in some project where many developers are involved [Mil04].

The policy design pattern [Mil04] can be implemented as a single aspect. In this case aspect defines a project-wide rules or policies.A Second approach is used if not only top level policies are required, but also local rules or exceptions must be considered. Then the top level policies are defined in an abstract aspect in which an abstract pointcut is declared. The Abstract aspect is then overridden by a concrete aspect which applies the local policies [Mil04].

Here is template adapted from [Mil04]:

```
public abstract aspect ProjectPolicyAspect {
protected abstract pointcut allowedSystemOuts();
        declare warning: call(* *.println(..)) && !allowedSystemOuts():
        "System.out usage detected. Suggest using logging?";
}

public aspect MyAppPolicyAspect extends ProjectPolicyAspect {
// Specifies regions where messages to System.out are allowed.
protected pointcut allowedSystemOuts():
        BorderControllerAspect.withinMyAppMainMethod() ||
        BorderControllerAspect.withinThirdParty() ||
        BorderControllerAspect.withinTestingRegion();
}
```

2.8 Worker Object Creation

The Worker Object Creation pattern [Lad03] and Proceed Object pattern [Sch04]. are very similar. Despite different terminology, it is clear they share the same idea. In the following description, the worker object creation pattern terminology will be used.

This pattern has a wide use. It may be used when the use of the proceed call in an object-oriented context is needed or when the proceed call should be executed in a different thread. This can be used with Java Swing Framework, where all calls which update the GUI must be performed inside the switching event dispatch thread (e.g., implementing thread safety in the Swing applications) [Sch04]. Another example of the situation when the proceed call should be executed in a different thread is improving responsiveness of GUI applications which perform complex computations (e.g., authorization and transaction management) [Lad03].

This pattern can be also used to advise the proceed call. This is desired when the aspect uses an around advice and the algorithm in the advice itself should be e.g., traced or logged [Sch04].

The Worker Object Creation pattern uses aspect to automatically create an object of anonymous class implementing the runable interface. Pointcut of this aspect captures all the join points which are needed. Advice simply executes the join point by calling proceed() inside the run() method in the body of the anonymous worker class. Calling the run() method at a later time or even in another thread will execute the captured join point [Lad03].

Here is a template advice of Worker Object Creation pattern adopted from [Lad03]:

```
void around() : <pointcut> {
        Runnable worker = new Runnable () {
                public void run() {
                        proceed();
                }
        }
}
```

Chapter 3

Classification of Aspect-Oriented Design Patterns

To study how are aspect-oriented design patterns related to each other it is needed to classify them. GoF classification of object-oriented design patterns is a natural first choice for this (Section 3.1), but it will shown that the classification based on the internal structure of an aspect fits this task much better (Section 3.2).

3.1 Gang of Four Classification

According to Gamma et al. object-oriented design patterns are divided into three major categories which are reflecting the purpose of design patterns. Here are the definitions of these categories [GHJV95]

Creational patterns are concerned in the object creation. They are controlling this creation and try to solve problems that are associated with this process.

Structural patterns deal with the problem of composition of objects or classes. They try to find relationships between entities and produce simple design which gives an answer to the composition problem.

Behavioral patterns characterize the way how objects interact or communicate.

The question is whether this classification can be applied also to intrinsic aspect-oriented design patterns. The Answer to this question is not so straightforward.

Some patterns were strong candidates for a certain category. For example, the Cuckoo's Egg pattern (Section 2.4) can be considered as a strong candidate for creational pattern group because it affects the process of creating the objects. The Director pattern (Section 2.5) is a reasonably denoted

as a structural pattern because it affects the structure of the base classes by adding the new roles to them. The Wormhole pattern (Section 2.1) can be considered to be behavioral pattern because it creates new way in which two objects can communicate.

Other patterns were harder to classify by this criteria. One of such patterns was Worker Object Creation pattern (Section 2.8). Despite the fact that this pattern creates a worker object and it could be classified as creational, the creation of worker object is only a means of achieving main objective of the pattern: proceeding a call and its use in another context such as a new thread.

At last some patterns are impossible to classify by this criteria. These include Participant pattern (Section 2.3) or Border Control pattern (Section 2.6). These patterns are not creational because their point of interest is not creation of some objects. They can't be considered structural either because they are not concerned in the composition of objects into new entities, nor they are behavioral because they are not concerned how objects communicate and interact.

To conclude, it is possible to apply GoF classification to aspect-oriented design patterns, but in some cases it is questionable which category certain design pattern belongs to.

3.2 Classification According to Aspect Structure

Hanenberg et al. also presented a set of AspectJ idioms and a scheme for their interrelated application [HSU03]. Similarly to the well-scheme of GoF patterns [GHJV95], it is represented by a graph in which patterns that can be combined are connected by directed edges. Each edge is annotated with the role of the pattern in which it originates plays in the pattern in which the edge terminates. However, no attempt is made to categorize the idioms.

The GoF design pattern classification used as the main criterion the purpose of a design pattern. However, interesting criteria could be the structure of an aspect-oriented design pattern.

Each design pattern is usually represented by one or more aspects. As has been mentioned in the (Section 1.3), an aspect contains three main components: inter-type declarations, pointcuts, and advices. By studying available aspect-oriented design patterns, one may notice that 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. In other words, one component is crucial to understand or achieve the logic of the design pattern.

If main purpose is laid on the pointcut, advice is not presented or is very simple. For example, the advice only presents how pointcuts would be used

in it, but the advice logic is not presented.[1] In the category where advices have the main role, pointcuts are usually defined as abstract or they are only represented as <pointcut>.

According to the structure of the aspects that are representing aspect-oriented design patterns, these can be divided into three categories: pointcut design patterns, advice design patterns and inter-type declaration design patterns.

- **Pointcut design patterns**: Wormhole pattern, Participant pattern, Border Control pattern
- **Advice design patterns**: Worker Object Creation pattern, Exception Introduction design pattern, Cuckoo's Egg design pattern, Director design pattern.
- **Inter-type declaration design patterns**: Policy design pattern.

In most cases patterns were straightforward to classify by this criteria. One can notice the Participant pattern (Section 2.3) where the pointcuts crucial to logic of the design pattern are not defined the in the aspect but in plain Java classes. The Director pattern (Section 2.5), which is considered as advice pattern, has a logic that can be sometimes expressed in a more complex way than just by an advice (e.g., using interfaces or additional method definitions). At last, it is possible to mention wormhole pattern (Section 2.1) where the advice is present only to show how to use pointcuts in it.

[1] This can be seen in the wormhole pattern (Section 2.1).

Chapter 4

Combination of Aspect-Oriented Design Patterns

Aspect-oriented design patterns are defined in a modular way. They can be usually implemented by one or more aspects, but in most cases their logic is concentrated in one place. This makes the possible combination easier. A combination of two patterns means a subsequent interrelated application of two patterns to a problem at hand. Along with application of design patterns to already applied patterns observations how these patterns were affected has been made. In general, a design pattern can be combined with many other design patterns. The question is whether this combination would be useful and meaningful.

This chapter will provide a sample problem and its solution using combination of aspect-oriented design patterns.

4.1 Class Deprecation Problem

Due to its complexity, software is usually developed in teams. Team development requires developers to obey some common rules and policies. Thus, new versions of classes in frameworks used by application programmers occur quite often. The old version of a class cannot be simply replaced with a new one at once. A new class has to be tested (and corrected if necessary) for some time during which it is common to have and use both versions of the class.

All developers should be kept informed of new class versions and warned (Section 4.2) or sometimes even forced—to use them (Section 4.3). Just instructing developers to do so simply does not work. Developers often forget to obey policies or they simply overlook the information about a new class version. A better way is to incorporate this information into the build

process (Section 4.2). Compiler messages—warnings and errors—that warn of broken policies and rules will hardly be overlooked by the developers.

In some cases, when the policy must be strictly fulfilled, more radical steps may have to be taken (Section 4.3). By introducing a new version of a class into the framework, its former version becomes deprecated. It would be useful not just to inform programmers they are not allowed to use the old version any more, but also to automatically change any old class instantiation for the new one.

The following sections will present a solution of this problem that can be achieved by combining four specific aspect-oriented design patterns. As an example, a program was created that finds a shortest path between two points in a specific graph. The graph with the path is then displayed, but there are two classes suitable for this task present in the system. Class Display, which is deprecated, should not be used no more. Class Display2 should be used instead. This solution has been also implemented a is provided in appendix C.

4.2 Warn of Deprecated Class Use

We first assume it is sufficient to warn the developer using a deprecated class named, for example, Display, and suggest him to use Display2 instead. For this purpose, it is convenient to use the Policy pattern (Section 2.7). The following code snippet shows how the Policy pattern can be applied to achieve this goal. Aspect will detect every call to the Display constructor and show the provided warning during compilation upon these calls.

```
public aspect Warning {
        declare warning : call(Display.new())
                : "Class Display is deprecated. Use Dysplay2 instead.";

}
```

Subsequently, we realize that we have to allow the use of Display within the the testing package and third party code. In this situation, the Border Control pattern (Section 2.6) can be applied. This pattern defines regions in an application that can be used by other design patterns or aspects. Following code presents an application of the Border Control pattern to our problem. The aspect defines four public pointcuts which represent regions in our application.

```
public aspect Regions {
        public pointcut WithinMyApplication():within(pathfinder.*);
        public pointcut WithinDisplay():within(pathfinder.Display);
        public pointcut WithinMain()
                :withincode(public static void pathfinder.Main.main(..));
```

```
        public pointcut WithinTesting():within(pathfinder.Testing);
        public pointcut WithinClassSwitcher():within(pathfinder.ClassSwitcher);
}
```

Afterwards, PolicyAspect will have to be adapted as shown in the following snippet. This represents the combination of the Policy pattern with the Border Control pattern.

```
public aspect Warning {
        declare warning :
                call(Display.new()) && !Regions.WithinTesting()
                : "Class Display is deprecated. Use Dysplay2 instead.";

}
```

As you may recall, Border Control is a pointcut pattern (Section 3.2). On the other hand, Policy is an inter-type declaration pattern (Section 3.2). As can be seen in the example, combining a pointcut pattern with an already applied inter-type declaration pattern requires changes in the already applied pattern.

If one could predict that there will be exceptions from policy of the use of Display, it would be possible to apply the Border Control pattern first and the Policy pattern could be then added without having to change the existing code. This suggests that combining an inter-type declaration pattern with an already applied pointcut pattern can be performed without having to change the already applied pattern.

4.3 Deprecated Class Swapping

Assume now one would like to make a change from Display to Display2 automatic while still informing developers of attempts to use Display. For this, we may use the Cuckoo's Egg pattern (Section 3.2). This pattern captures calls to a constructor of a particular class and employs an around advice to replace it with a call to a constructor of another class.

The Following code shows how this pattern may be applied to replace Display constructions with Display2 construction:

```
public aspect ClassSwitcher {
        public pointcut oldClassConstructor( ) :
                call(Display.new( )) &&
                Regions.WithinMyApplication()&&
                !Regions.WithinTesting();

        Object around( ) : oldClassConstructor( ){
                return new pathfinder.Display2( );
        }
}
```

Recall from Section 3.2 that Display2 must be a subtype of Display; otherwise, we will get a class cast exception on the first attempt to instantiate Display. Moreover, we need Display2 to be a subtype of Display to make it compatible with the existing references to Display. This can be achieved either by stating inheritance directly in Display2 or by using a declare parents inter-type declaration (presumably, though not necessarily, in the Cuckoo's Egg aspect itself).

The Cuckoo's Egg pattern uses the pointcuts defined in the Border Control pattern, too. Cuckoo's Egg is an advice pattern and it was combined with Border Control without having to change it. This suggests that combining an advice pattern with an existing pointcut pattern can be made without changes in the already applied pattern.

4.4 Logging of Swapping

Assume there is a need to log the swapping of the deprecated class with the new one. This would seem a simple task. A logging code could be simply added to the CuckooEgg's advice. When there is a need to switch from deprecated class to new version this advice would be executed. But there is a problem. When the logging piece of code is added to the advice, assuming the logging is performed into a text file, it is needed to deal with an IOException that could occur during the execution of this code.

As mentioned in the Section 2.2, an aspect cannot declare throwing of an exception that was not declared by the advised join point. Another possibility is to throw a runtime exception. In this case the Exception Introduction pattern can be used. This pattern suggests to use a concern-specific runtime exception, which extends a runtime exception. By this, it becomes easier to distinguish between exceptions thrown by various aspects. Also, if a runtime exception has been used there would be no difference between exceptions thrown by various concerns [Lad03].

The following code snippet presents an example implementation of a concern-specific exception and a use of the Exception Introduction pattern adapted to our problem:

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

public aspect SwitchLogging {
        before(): adviceexecution() && Regions.WithinClassSwitcher() {
                try {
                        logSwapEwent()
                }
                catch(IOException e){
```

```
                throw new SwitchLoggingException(e);
            }
        }
```

Exception Introduction is considered to be an Advice pattern and it was added to the Cockoo's Egg pattern without having to make any change in it. As can be seen from the code snippet, it can also reuse definitions from the already applied Border Control pattern.

Chapter 5

Regularity in Aspect-Oriented Design Pattern Combination

As we will see in this chapter, the combination of aspect-oriented design patterns is substantially affected by their structural category (defined in Section 3.2). As mentioned before, by combination of two patterns is considered as a subsequent interrelated application of two patterns to a problem. Thus, one of the patterns is applied to the problem, and afterwards another one is applied in connection to the artifacts of the former pattern.

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. An example of a pattern that can be combined with almost any other pattern is Exception Introduction, which represents an advice pattern. Combination of this pattern with another advice pattern was presented in Section 4.4 where Exception Introduction was added to Cuckoo's Egg.

Another pattern that can be used with other, already applied patterns without having to make any changes to them is Policy, which is an inter-type declaration pattern. This pattern defines a pointcut that captures the join points in a base concern or another pattern whose occurrence represents breaking of some policy. If such a joint point occurs, a compile error or warning is issued.

It is also possible to combine a pointcut pattern with another pattern of the same type without having to change that pattern. In such a combination, the new pattern will actually use the pointcuts of the already applied pointcut pattern. A simple example of this would be combining a Wormhole pattern with an already applied Border Control pattern. This way, the Wormhole pattern would be able to use regions defined by the Border Control pattern in its own pointcuts.

However, combining a pointcut pattern with an already applied advice or inter-type declaration pattern usually requires a change of this pattern. An example of combining a pointcut pattern with an already applied inter-type declaration pattern has been presented in Section 4.2 where we had the Policy pattern applied and combined the Border Control pattern with it. Recall also from the same section that 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.

Assume the pointcuts of a particular non-pointcut pattern in a developing application can no longer be defined in a simple way because it is not certain whether the pattern should be applied to new classes. Such a pattern can be combined with a Participant pattern, which is a pointcut pattern, which would enable individual classes to declare participation in this pattern application. However, the implementation of a Participant pattern requires the already applied pattern code to be altered.

Figure 5.1 presents schematically the combinations of the aspect-oriented design patterns we discussed. Pattern category is indicated graphically: oval nodes represent advice patterns, rectangular nodes are pointcut patterns, and rhomboid nodes stand for inter-type declaration patterns. Where any pattern of the given category is applicable, its name is shown as asterisk. The edge direction corresponds to the direction of pattern application: an edge originates in the pattern being applied and ends in the pattern to which this pattern is applied to achieve pattern combination. Dashed edges mean no change of the pattern at the edge end is required, while solid lines mean the change is necessary.

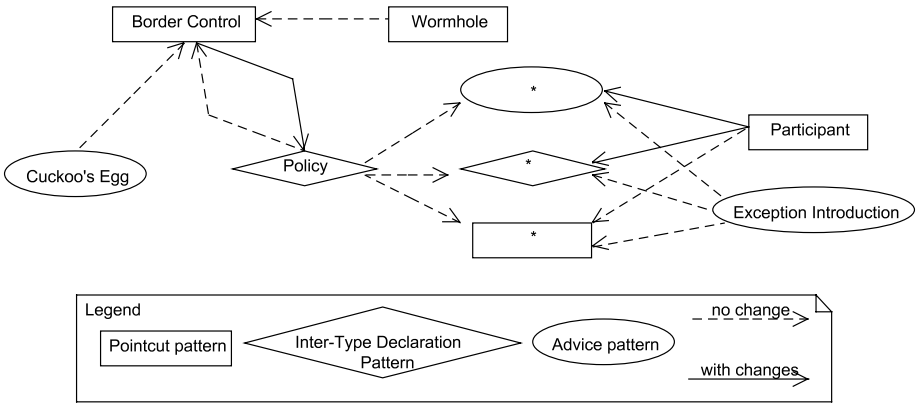

Figure 5.1: Combinations of aspect-oriented design patterns and required changes.

In Figure 5.1 it can be seen that combination of a pointcut pattern with another pointcut pattern does not require changes in the already applied

pattern. This represents simply defining further pointcuts.

Combining a pointcut pattern with an already applied advice or inter-type declaration pattern requires changes in the advice or inter-type declaration pattern since the combination assumes the use of pointcuts defined in the pointcut pattern by the patterns of the latter two categories. This is caused by the nature of pointcut patterns: they define pointcuts to be used by other aspects in the application.

On the other hand, a combination of an advice pattern or inter-type declaration pattern with other patterns of any category can be in most cases achieved without changes to the already applied pattern.

Table 5.1 summarizes the identified dependence of the aspect-oriented pattern combination on the aspect-oriented pattern category in the table. The columns represent the already applied patterns. The rows are the patterns to be applied. The values in the cells indicate whether the already applied patterns have to be modified in case of their combination with the pattern in the corresponding row.

Table 5.1: Aspect-oriented categories and their combinations.

combined with →	Pointcut Pattern	Advice Pattern	Inter-Type Declaration Pattern
Pointcut Pattern	no change	with changes	with changes
Advice Pattern	no change		
Inter-Type Declaration Pattern	no change		

Chapter 6

Conclusion and Future Work

This report was concerned with classification and combination of aspect-oriented design patterns. An overview of common aspect-oriented design patterns was presented. The GoF classification was applied to aspect-oriented patterns, but sometimes it was questionable which category certain design pattern belongs to. Therefore, a new classification according to the aspect structure has been introduced. This classification suggests to classify intrinsic design patterns according to structure of the aspect which represents them into three structural categories: pointcut design patterns, advice design patterns and inter-type declaration design patterns (Chapter 3.2).

Identified categories are significant in combination of aspect-oriented design patterns. According to structural category of design patterns it is possible to predict whether another design pattern can be added to an already applied pattern with or without having to modify the applied pattern (Chapter 5). Regularity in combining aspect-oriented design patterns has been studied on the problem of class deprecation. Incremental solution involving four aspect-oriented design patterns, of this problem, was presented (Chapter 4) and fully implemented (see the attached CD, Appendix C).

It is expected that new patterns will be discovered, which will bring new possibilities into the combinations. As a future work these combinations could be studied. Combination of reimplemented object-oriented patterns with intrinsic aspect-oriented design patterns could also be interesting. It seems some of object-oriented design patterns can be implemented using adapted aspect-oriented design patterns (e.g, Observer by Director or Singleton by Policy). This indicates that some aspect-oriented design patterns can be generalized forms of object-oriented design patterns.

Bibliography

[Ale79a] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979. Cited in [ST02].

[Ale79b] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979. Cited in [GHJV95].

[Cop04] James O. Coplien. The culture of patterns. *Computer Science and Information Systems (ComSIS)*, 1(2), November 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[HB03] Ouafa Hachani and Daniel Bardou. On aspect-oriented technology and object-oriented design patterns. In Jan Hannemann, Ruzanna Chitchyan, and Awais Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

[HC02] Stefan Hanenberg and Pascal Costanza. Connecting aspects in AspectJ: Strategies vs. patterns. In Yvonne Coady, Eric Eide, David H. Lorenz, Mira Mezini, Klaus Ostermann, and Roman Pichler, editors, *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, March 2002.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA*, pages 161–173, 2002.

[HSU03] Stefan Hanenberg, Arno Schmidmeier, and Rainer Unland. Aspectj idioms for aspect-oriented software construction. In *Proceedings of 8th European Conference on Pattern Languages of Programs, EuroPLoP 2003*, Irsee, Germany, June 2003.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi

Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[Men07] Radoslav Menkyna. Towards combining aspect-oriented design patterns. In Mária Bieliková, editor, *IIT.SRC: Student Research Conference*, pages 1–8. Slovak University of Technology, 2007.

[Mil04] Russell Miles. *AspectJ Cookbook*. O'Reilly, 2004.

[MV] Radoslav Menkyna and Valentino Vranić. A bordered cuckoo's egg policy: Combining aspect-oriented design patterns. Submitted to *2nd International Conference on Software and Data Technologies (ICSOFT)*, April 2007.

[Sch04] Arno Schmidmeier. Patterns and an antiidiom for aspect oriented programming. In *Proceedings of EuroPLoP 2004*, 2004.

[ST02] Alan Shalloway and James R. Trott. *Design Patterns Explained*. Software Patterns Series. Addison-Wesley, 2002.

Appendix A

Towards Combining Aspect-Oriented Design Patterns

This appendix contains:

> Radoslav Menkyna. Towards combining aspect-oriented design patterns. In Mária Bieliková, editor, *IIT.SRC: Student Research Conference*, pages 1–8. Slovak University of Technology, 2007.

This article was accepted to the Informatics and Information Technologies Student Research Conference (iit.src). It was awarded as best paper in Bachelor category and selected to be sent at the ACM Student Research Competition.

Towards Combining Aspect-Oriented Design Patterns

Radoslav MENKYNA*

Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
xmenkyna@is.stuba.sk

Abstract. Although aspect-oriented paradigm is quite new, patterns are emerging in it already. This article provides an overview of common aspect-oriented design patterns. The article discusses how these patterns can be combined. Certain dependence between the design pattern structure and its combination ability has been identified. Knowing a structure of patterns, it is possible to say whether adding a new pattern to existing one can be made with or without change of existing pattern. Classification according to structure into pointcut design patterns and advice design patterns is proposed.

1 Introduction

With increasing size of the project grows also the complexity of problems that have to be solved. It is natural to use simple smaller building blocks to produce bigger structures. That's why it is natural to combine design patterns to solve complex problems.

This article identifies some regularities in combining aspect-oriented design patterns according to their classification. Design patterns are classified according to structure into pointcut and advice design patterns. Certain dependence between the combination ability of specific pattern and its structure has been identified. According to structure of the pattern it is possible to say whether adding some pattern to existing pattern can be made with or without changes of existing pattern. The combination of design patterns allows using their features together, but also it can lead to new features.

The rest of the article is organized as follows. In Section 2 an overview of aspect-oriented design patterns can be found Section 3 presents a classification of aspect-oriented

* Supervisor: Dr. Valentino Vranić, Faculty of Informatics and Information Technologies STU in Bratislava.

IIT.SRC 2007, Bratislava, April 18, 2007, pp. 1–8.

design patterns according to Structure. Section 4 discusses how design patterns can be combined and presents the dependence between structure and combination ability of a pattern. Section 5 provides an overview of related work Section 6 represents the conclusion and future work.

2 Overview of Aspect-Oriented Design Patterns

Despite aspect-oriented paradigm is just spreading, some strategies and idioms have already been substantially generalized, do not rely on a particular language any more, and as such can be accepted as design patterns. This section will present an overview of common aspect-oriented design patterns.

2.1 Wormhole

The Wormhole pattern [4] connects the callee with caller in such a way that they share their context information. It creates a direct connection between two levels in the call stack. This is very helpful when additional context information has to be added [4]. Instead of adding extra parameters in each method in the control flow or using a global storage, this pattern can be used.

2.2 Exception Introduction

In some cases when the aspect is used to implement some crosscutting concerns checked exceptions have to be caught in its advice. This usually happens when methods of Java libraries, which declare to throw such exceptions, are used in its advice. In AspectJ,[1] advice cannot declare to throw a checked exception unless the advised joint point declared this exception. The base concern logic cannot declare those exceptions because it simply does not know anything about the logic used in the crosscutting concern [4].

The Exception Introduction pattern [4] suggest that the checked exceptions should be caught and simply wrapped into new concern-specific runtime exceptions. Such exceptions can be then thrown to higher level, where they can be unwrapped and the real cause of exception revealed.

2.3 Participant

Usually, aspects try to introduce some behavior to a base concern in such a way that the base concern is not aware of the aspect. In this pattern, the roles swap: an aspect makes classes to participate. This is needed for the purpose the Participant [4] is trying to achieve. In some cases, defining pointcuts only by the means of language syntax is not sufficient.

[1] This problem is common to many aspect-oriented languages.

For example, if the advice should affect only methods with certain characteristics, one cannot decide only according to their names whether to include them in the pointcut or not. Only the creator of these methods knows their characteristics. This is where comes from the idea that classes themselves should express if they want to be advised. If they want to participate they simply define an appropriate pointcut in them [4].

2.4 Cuckoo's Egg

Cuckoo's Egg design pattern [5] is quite simple but powerful. It expresses how powerful aspect-oriented programming can be. It is used to control or change the objects created by the constructor call. This means that with this pattern it is possible to change the type of the object being instantiated [5].

2.5 Director

The Director pattern [5] can be used to define some roles or behavior to an unknown number of classes. A role can be defined without knowing the particular class it will be applied to. A pattern can be used to define some logic in an abstract aspect without knowing the classes this logic will be applied to. The Director can also implement some relationships within abstract entities [5].

2.6 Border Control

The Border Control design pattern [5] is used to define some reasonable regions in the application. These regions are later reused by other aspects to ensure they are used only in correct scope. Use of this pattern is also convenient when the changes in structure of the application are expected. After such changes only declarations of regions in border control aspect are changed and other aspects which are reusing these declarations will be also affected [5].

2.7 Policy

The main idea of Policy pattern is to define some policy or rules within the application. The rules can wary from suggestions and warnings to overriding methods, classes or libraries. This is very useful in some project where many developers are involved [5].

2.8 Worker Object Creation

Worker Object Creation pattern [4] has a widespread use. For example, it may be used when the use of the proceed call in an object-oriented context is needed or when the proceed call should be executed in a different thread. This can be used with Java Swing Framework, where all calls which update the GUI must be performed inside the event dispatch thread [6]. Another example of the situation when the proceed call should be

executed in a different thread is improving responsiveness of GUI applications which perform complex computations (e.g., authorization and transaction management) [4].

This pattern can also be used to advise the proceed call. This is desired when the aspect contains an around advice and the algorithm in the advice itself should be, for example, traced or logged [6].

3 Aspect-Oriented Design Pattern Classification According to Structure

Interesting criterion for classification of aspect-oriented design patterns could be the structure of an aspect-oriented design pattern.

Each design pattern is usually represented by one or more aspects. Structurally, an aspect consists of three main components: inter-type declarations, pointcuts and advices. Generally, the structure of an aspect that represents a design pattern can be divided into two categories. These categories differ according to which component is more significant for the main meaning for the purpose of the design pattern or, in other words, which component is crucial to understand or achieve the logic of the design pattern.

If main purpose is realized by pointcuts, advice is usually not presented or is very simple. For example, an advice only presents how pointcuts would be used in it, but the advice logic is not represented. In the category where advices have the main role, pointcuts are usually declared as abstract or they are only symbolic.

Thus, according to their structure, aspect-oriented design patterns can be divided into two categories: pointcut design patterns and advice design patterns. Patterns from listed in the overview were divided according to structure into these categories:

- **Pointcut design patterns**: Wormhole pattern, Participant pattern, Border Control pattern

- **Advice design patterns**: Worker Object Creation pattern, Exception Introduction pattern, Cuckoo's Egg pattern, Director pattern, Policy pattern.

In most cases patterns were easy to classify by this criteria. It is possible to point out the Participant pattern (Section 2.3) where the pointcuts crucial to logic of the design pattern are not defined the in aspect but in plain Java classes. The Director design pattern (Section 2.5) has a logic that can be sometimes expressed in a more complex way than just by an advice (e.g., using interfaces or additional method definitions). In wormhole pattern (Section 2.1) the advice is present only to show how to use pointcuts in it. At last, the Policy design pattern (Section 2.7) is in this paper treated like the advice pattern, although it could be a representant of new inter-type declaration design patterns category. From the point of combination these two categories have the same behaviour.

4 Combination of Aspect-Oriented Design Patterns

Aspect-oriented design patterns are defined in a modular way. They can be usually implemented by one or more aspects, but in most cases their logic is concentrated in one place. This comes from the nature of aspect-oriented programming. Aspects usually alter behavior of base concerns without requiring awareness on their side. This makes the combination of aspect-oriented design patterns easier than the combination of object-oriented design patterns is. In general, a design pattern can be combined with many other design patterns. The question is whether this combination would be useful and meaningful. Due to this, we may expect that aspect-oriented design patterns more easily form pattern languages.

In the following sections a dependence between the structure and combination ability of patterns together with examples of combinations will be presented.

4.1 Dependence Between Structure and Combination Ability of Patterns

There is a connection between the structure of aspect-oriented design patterns and the way how they can be combined with other aspect-oriented design patterns. The combination of aspect-oriented design patterns is substantially affected by their structural type, this means is possible to make statements about the way how this pattern could be combined with other patterns.

It seems that a combining of a pointcut design pattern (Section 3) with another pointcut design pattern does not require changes of existing design pattern. In this kind of combination, the new pattern will reuse the pointcuts of the existing pattern. Adding pointcut a design pattern to an advice design pattern usually requires changes in the advice design pattern.

On the other hand, a combination of an advice design pattern with another design pattern of any structural type can be done without changes to existing design pattern. Examples of such combinations can be seen in sections: 4.2, 4.3. Dependence between the structure of design patterns and the way how they can be combined is summarized in Table 4.1.

Tab. 1. Dependence between the structure of design patterns and the way how they can be combined.

	pointcut design pattern	advice design pattern
pointcut design pattern	without change	change required
advice design pattern	without change	without change

4.2 Adding a Feature

Sometimes, a design pattern only adds some feature to the system. If this feature is needed by another design pattern, the patterns can be used together. Aspect-oriented design patterns are usually represented by one or more aspects. From the nature of aspects some design pattern can be added to another without the modification of the existing pattern.

An example of such a pattern that can be combined with almost any other pattern is the Exception Introduction design pattern (Section 2.2). This pattern adds the ability to use exceptions in advices in a proper way. When this feature is needed in another design pattern, the Exception Introduction pattern can simply be added to the program without having to make any change in existing patterns. Also Policy design pattern (Section 2.7) can be used together with another design patterns without any changes to those patterns.

Example of exception introduction pattern adapted from [4]:

```
public abstract aspect ExceptionIntroductionAspect {
        abstract pointcut operations();
        // pointcut operations defines where should exception occur.
        // When this is defined as another design pattern exception introduction is used together with an existing pattern
        // and any change of its code is required.
        Object around() : operations() {
                try {
                return proceed();
        } catch (CheckedException ex) {
                throw new RuntimeException(ex);
                // CheckedException will be caught and new runtime exception will be thrown.
                }
        }
}
```

Adding pointcut design pattern to an advice design pattern requires usually a change in existing advice pattern. Example of such a pattern is the Border Control design pattern (Section 2.6). This pattern defines regions that are later used by other aspects or design patterns. This suggests that after adding a feature represented by this pattern existing aspects and patterns have to be altered.

Assume the pointcuts of a particular pattern in a growing application can no longer be defined in a simple way because it is not certain whether the pattern should be applied to new classes. Such a pattern can be combined with a Participant pattern (Section 2.3) and the class can participate in the application of this pattern. Due to the implementation of the Participant pattern, the existing pattern code must be altered.

All the combinations in this section are summarized at Figure. 1. There are three groups of design patterns on the figure. Any pattern from the groups on a sides can be combined with any pattern in the middle. Combining pattern from the left group, advice design pattern, can be usually done without change. When using pattern from the right group, pointcut design pattern, changes of existing design pattern will be required. All patterns in the middle group are advice design patterns.

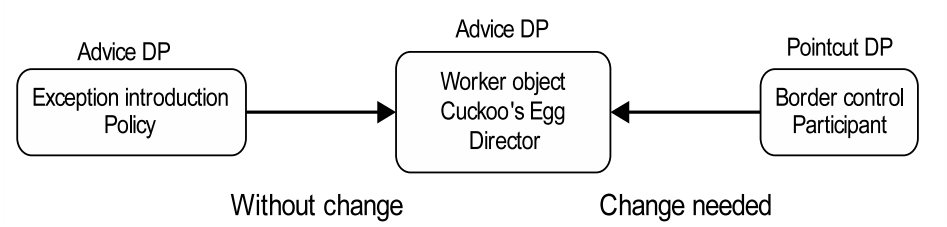

Fig. 1. Illustration of possible combinations of aspect-oriented design patterns.

4.3 Achieving New Functionality

Aspect-oriented design patterns can also be combined to achieve new functionality. A simple example of such a use is the combination of the Policy design pattern (Section 2.7) with Cuckoo's Egg design pattern (Section 2.4). The Policy pattern in most cases only defines some warnings or suggestions upon breaking the specific policy. By combining it with the Cuckoo's Egg design pattern it is possible to go further by not only showing warning, but also override the use of specific classes and their methods.

5 Related Work

In GoF description of object-oriented design patterns [1] a section called related patterns can be found. This section lists the names of the patterns often used with the pattern. This in other words suggests which patterns can be combined with the pattern to solve some problem. GoF also presented graphical representation of these relations. From this representation, groups of patterns that are often combined can bee seen.

Object-orientated design patterns were reimplemented in AspectJ [3]. In many cases better modularity was achieved, so the design pattern structure became clearer and simpler to understand [3]. These patterns could be also combined with the intrinsic aspect-oriented design patterns and better modularity achieved by reimplementation would ease this combination.

Combination of idioms is similar to combination of design patterns. Idioms can be generalized to become design patterns. Also idioms can be combined in various ways to achieve a solution to some problem [2].

6 Conclusion and Future Work

The Article proposed ways how the aspect oriented design patterns can be combined. Certain dependence between the design pattern structure and its combination ability was discussed. The combination of the pointcut design pattern with another pointcut pattern is usually done without need to modify the existing pattern. The combination of the pointcut design pattern with an advice design pattern requires a change of existing

structures during the implementation. Adding an advice design pattern to another design pattern leads usually to the implementation where no change of existing pattern is needed.

This paper provided an overview of common aspect-oriented design patterns. An alternative way of classification according to the structure of an aspect that represents the design pattern was presented. Aspect-oriented patterns were divided into two categories: pointcut design patterns and advice design patterns.

As a future work I would like to examine some inter-type declaration design patterns which can form another structural category. More combinations of the aspect-oriented design patterns should be examined. It is expected that new patterns will arise, which will bring new possibilities into the combinations. There are also many object-oriented design patterns reimplemented in AspectJ [3]. Combination of these patterns with intrinsic aspect-oriented design patterns could be also interesting.

Acknowledgement: This work was partially supported by the Scientific Grant Agency of Slovak Republic, grant No. VG1/3102/06.

References

[1] Gamma, E., et al.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] Hanenberg, S., Schmidmeier, A., Unland, R.: AspectJ Idioms for Aspect-Oriented Software Construction. In: *Proceedings of EuroPLoP 2003*, 2003.

[3] Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: *Proceedings of the 17th conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 2002, pp. 161–173.

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

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

[6] Schmidmeier, A.: Patterns and an Antiidiom for Aspect Oriented Programming. In: *Proceedings of 9th European Conference on Pattern Languages of Programs, EuroPLoP 2004*, Irsee, Germany, 2004.

Appendix B

A Bordered Cuckoo's Egg Policy: Combining Aspect-Oriented Design Patterns

This appendix contains:

> Radoslav Menkyna and Valentino Vranić. A bordered cuckoo's egg policy: Combining aspect-oriented design patterns. Submitted to *2nd International Conference on Software and Data Technologies (ICSOFT)*, April 2007.

Paper was submitted to 2nd International Conference on Software and Data Technologies (ICSOFT). Currently, a review process is ongoing. Results are going to be announced on May 21, 2007[1]. My contribution to this paper is approximately 70 %.

[1] http://www.icsoft.org/

A BORDERED CUCKOO'S EGG POLICY: COMBINING ASPECT-ORIENTED DESIGN PATTERNS

Radoslav Menkyna, Valentino Vranić
Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies
Slovak University of Technology, Ilkovičová 3, 84216 Bratislava 4, Slovakia
xmenkyna@is.stuba.sk, vranic@fiit.stuba.sk

Keywords: aspect-oriented design patterns, aspect structure, Policy, Border Control, Cuckoo's Egg, AspectJ

Abstract: This paper presents a combination of three particular aspect-oriented design patterns: Policy, Cuckoo's Egg, and Border Control. The combination is studied in the context of the class deprecation problem in team development. Each of these three patterns is a representative of one of the three structural categories of aspect-oriented design patterns: pointcut, advice, and inter-type declaration pattern category. Although aspect-oriented patterns mostly can be combined with one another without having to modify the code of the pattern that has been applied first, this is not always so. Based on the structural categorization of aspect-oriented design patterns, a regularity in their sequential combination is uncovered and discussed in general and within a detailed example of Policy, Cuckoo's Egg, and Border Control combination and further examples of aspect-oriented design pattern combinations.

1 INTRODUCTION

Although the notion of pattern in its original sense proposed by Alexander was indivisible of the notion of pattern language (Alexander, 1979), software patterns are often perceived as more or less independently applied sublimated pieces of development experience (Coplien, 2004). Having it this way, we tend first to discover new patterns and then think of the opportunities of their combination rather then to aim at discovery of integral pattern languages that inherently comprise the ties between the patterns. This is so with object-oriented design patterns, and we may see this also applies to aspect-oriented design patterns that are just being discovered both on individual basis (Laddad, 2003; Miles, 2004; Schmidmeier, 2004) and as pattern languages (Hanenberg et al., 2003).

There are already a significant number of aspect-oriented design patterns discovered. Here we will go through a combination of three particular aspect-oriented design patterns towards some general assumptions on aspect-oriented design pattern combination based on their structure.

Aspect-oriented design patterns discussed here are related to the mainstream aspect-oriented approach established by PARC (Kiczales et al., 1997) whose main programming language representative is AspectJ. Numerous other aspect-oriented languages, such as AspectC++, AspectS, or Weave.NET, follow this paradigm. Most of existing frameworks that provide aspect-oriented programming support, such as Spring, JBoss, or Seasor, also follow the PARC approach.

The rest of the article is organized as follows. First, Section 2 states the problem of class deprecation in team development which we will use to illustrate the pattern combination. Section 3 describes the structure of Policy, Border Control, and Cuckoo's Egg, the three aspect-oriented design patterns that can help in solving the class deprecation problem, and introduces a structural categorization of aspect-oriented design patterns. Section 4 shows how Policy, Border Control, and Cuckoo's Egg can be actually combined to solve the class deprecation problem. Based on the structural categorization of aspect-oriented design, Section 5 devises a regularity in the sequential combination of aspect-oriented design patterns and discusses it in general and within a detailed example of the Policy, Cuckoo's Egg, and Border Control combination along with further examples

of aspect-oriented design pattern combinations. Section 6 presents an overview of related work. Finally, in Section 7, we make some conclusions and indicate directions for further work.

2 Overcoming the Class Deprecation Problem in Team Development

This section will define the class deprecation problem in team development which we will use as an example for pattern application throughout the rest of the article.

Due to its complexity, software is usually developed in teams, and team development requires developers to obey some common rules and policies. A frequent example is the introduction of a new version of a class is to the framework used by application programmers. The old version of a class cannot be simply replaced with the new one at once. A new class has to be tested (and corrected if necessary) for some time during which it is common to have and use both versions of the class.

All developers should be kept informed of the new class version and warned—or sometimes even forced—to use it. Just instructing developers to do so simply doesn't work. Developers often forget to obey policies or they overlook the information about a new class version. A better way is to incorporate this information into the build process. Compiler messages—warnings and errors—that notify developers of broken policies and rules have a better chance not to be overlooked.

In cases when the policy must be strictly fulfilled, the more radical steps may have to be taken. By introducing a new version of a class into the framework, its former version becomes deprecated. It would be useful not just to inform developers they are not allowed to use the old version any more, but also to automatically detect all attempts to instantiate the old class and swap it with the new class instantiation. In the next two sections we will see how the realization of this policy can be achieved by combining three specific aspect-oriented design patterns, namely Policy, Border Control, and Cuckoo's Egg.

3 Structure of Aspect-Oriented Design Patterns

The main construct in PARC aspect-oriented programming is an aspect. It consists of pointcuts, which specify the join points the aspect affects, advices that implement the affecting functionality, and inter-type declarations that statically affect types by introducing new fields and methods into them, inheritance relationship, warnings, compile errors, softened exceptions, and annotations.

We will take a closer look at three aspect-oriented design patterns (Section 3.1–3.3) originally described by Miles (Miles, 2004) that we will combine (in Section 4) to solve the problem of class deprecation in team development described in the previous section. Subsequently, we will devise structural categorization of aspect-oriented design patterns (Section 3.4).

3.1 Border Control

The Border Control pattern (Miles, 2004) is used to define regions in the application. These 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. In case of system changes, only declarations of regions in the Border Control aspect should be changed and other aspects which are using these declarations will be automatically redirected. As shown in Figure 1, the Border Control pattern can be implemented by a single aspect containing only a set of pointcuts that define the regions. Regions may represent types or methods. For this, **within**() and **withincode**() primitive pointcuts are used, respectively.

3.2 Cuckoo's Egg

The Cuckoo's Egg pattern enables to put another object instead of the one that the creator expected to receive, much similar to what a cuckoo does with its eggs. The pattern is implemented by an aspect that consists of a pointcut that captures constructor calls of the object to be swapped and an advice that actually does the swapping by simply creating and returning another object.

Figure 2 shows an example code of the Cuckoo's Egg pattern. Several types can be covered by swapping (the code in the figure shows two classes, MyClass1 and MyClass2) with constructor calls restricted with respect to where they occur. Note that the swapping object must be a subtype of the original object class; otherwise, we will get a class cast exception on the first attempt to instantiate the original class.

3.3 Policy

The main idea of the Policy pattern is to define some policy or rules within the application. A breaking of

```
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(..));
    . . .
}
```

Figure 1: The Border Control pattern.

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

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

Figure 2: The Cuckoo's Egg pattern.

such a rule or policy involves issuing a compiler warning or error. This is very useful in projects that involve many developers. The Policy pattern can be implemented by a single or several aspects. A single aspect approach is used to define project-wide rules or policies. If local rules or exceptions have to be addressed as well, project-wide rules and policies are defined in an abstract aspect with an abstract pointcut. This pointcut is overridden in concrete aspects that inherits from the abstract aspect in order to implement local policies and rules (Miles, 2004).

3.4 Aspect-Oriented Design Pattern Categories

Each aspect-oriented design pattern comprises at least one aspect. By studying available aspect-oriented design patterns, one may notice that 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 the structure of the aspects that implement them, aspect-oriented design patterns can be divided into three categories: pointcut patterns, advice patterns, and inter-type declaration patterns. Each of the patterns introduced so far is a representative of one of these categories. In the following text, we present further examples of patterns and categorize them.

3.4.1 Pointcut Patterns

The Border Control pattern described in Section 3.1 is an example of a pointcut pattern. It actually contains no other elements than pointcuts. Other examples of pointcut patterns include Wormhole and Participant. The Wormhole pattern (Laddad, 2003) employs pointcuts to connect a method callee with a caller in such a way that they can share their context information. It creates a direct connection between two levels in the call stack. This is very helpful when additional context information has to be added (Laddad, 2003). Without this pattern we would have to add extra parameters to each method in the control flow or to use a global storage.

Usually, aspects introduce some behavior to base concerns in such a way that the base concern is not aware of the aspect. In the Participant pattern (Laddad, 2003), the roles swap: a class decides whether it will allow an aspect to affect it by declaring an appropriate pointcut. This may be useful when it is not possible to capture classes and methods that have to be affected by an aspect with the pointcut language in a reasonable way. For example, if an advice should affect only methods with some properties not reflected in their names, it is not possible to capture them by a pointcut other by literally listing them.

3.4.2 Advice Patterns

The Cuckoo's Egg pattern described in Section 3.2 is an example of an advice pattern. Other examples of advice patterns include Worker Object Cre-

```
public abstract aspect ProjectPolicy {
    protected abstract pointcut allowedSystemOuts();

    declare warning: call(* *.println(..)) && !allowedSystemOuts():
        "System.out usage detected. Suggest using logging?";
}

public aspect MyAppPolicy extends ProjectPolicy {
    protected pointcut allowedSystemOuts(): BorderControllerAspect.withinMyAppMainMethod() ||
        BorderControllerAspect.withinThirdParty() ||
        BorderControllerAspect.withinTestingRegion();
}
```

Figure 3: The Policy pattern (adapted from (Miles, 2004)).

ation and Exception Introduction.The Worker Object Creation pattern (Laddad, 2003)—also known as Proceed Object (Schmidmeier, 2004)—captures the original method execution into a runnable object. This way it may be manipulated further. A typical use is to post its execution to another thread. This is very useful with Java Swing framework where all calls that update the GUI must be performed inside the event dispatch thread. Another example is improving responsiveness of GUI applications (Laddad, 2003). This pattern can also be used to advise the call to **proceed**(). This is desired when an aspect contains an around advice and the algorithm in the advice itself should be, for example, traced or logged (Schmidmeier, 2004).

In some cases, checked exceptions have to be caught by an advice. This usually the case when methods of Java libraries which declare to throw such exceptions are used in the advice. In AspectJ, an advice cannot declare throwing of a checked exception unless the advised joint point declared this exception. The base concern logic cannot declare those exceptions because it simply does not know anything about the logic used in the crosscutting concern (Laddad, 2003). The Exception Introduction pattern (Laddad, 2003) suggests that the checked exceptions should be caught and simply wrapped into new, concern-specific runtime exceptions. Such exceptions can be then thrown to a higher level where they can be unwrapped and the real cause of exception revealed.

3.4.3 Inter-Type Declaration Patterns

The Policy pattern described in Section 3.3 is an example of an inter-type declaration pattern. Another example of an inter-type declaration pattern is Default Interface Implementation (Laddad, 2003) which employs inter-type declarations to introduce fully implemented methods into interfaces.[1] The classes that implement these interfaces inherit the method implementations and do not have to provide their implementation if the default one is satisfactory.

4 Combining Aspect-Oriented Design Patterns

This section will show how Policy, Border Control, and Cuckoo's Egg can be combined to solve the class deprecation problem in team development presented in Section 2.

Suppose the instantiation of sOldClass is deprecated. In our first approach to this problem we assume it is sufficient to issue a warning in case of deprecated class named instantiation. Developers are supposed to manually change to NewClass instead. Figure 4 shows how the Policy pattern can be applied to achieve this. The aspect in the figure will detect every call to the OldClass constructor and show the provided warning text during compilation. AspectJ 5 supports declaring annotations, so a standard @deprecated annotation can be introduced instead of a general warning with a custom message as can be seen in Figure 5.

Subsequently, we realize that we have to allow the use of OldClass within the testing package and third party code. In this situation, the Border Control pattern (Section 3.1) can be applied. This pattern defines regions in the application that can be used by other design patterns or aspects. Figure 6 presents an application of the Border Control design pattern to our problem. The aspect defines three public pointcuts which represent regions in our application. Afterwards, we

[1] Laddad actually introduces Default Interface Implementation as an AspectJ idiom (Laddad, 2003).

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

Figure 4: Capturing instantiations of a deprecated class with the Policy pattern.

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

Figure 5: A Policy pattern that introduces @deprecated annotation.

will have to adapt the OldClassDeprecation aspect as shown in Figure 7. By this, we actually combined a Policy with an existing Border Control.

As you may recall, Border Control is a pointcut pattern (Section 3.1). On the other hand, Policy is an inter-type declaration design pattern (Section 3.3). As we saw in the example, combining a pointcut pattern with an existing inter-type declaration pattern requires changes in the existing inter-type declaration pattern.

If we knew from the beginning there will be exemptions from banning the use of OldClass, it would be possible to apply the Border Control pattern first and the Policy pattern could be then added without having to change the existing code. This suggests that combining an inter-type declaration pattern with an existing pointcut pattern can be performed without having to change the existing pattern.

Assume now we would like to make a change from OldClass to NewClass automatic while still keeping developers informed of attempts to instantiate OldClass outside of the testing package and third party code. This may be achieved with the Cuckoo's Egg pattern (Section 3.2). This pattern captures calls to a constructor of a particular class and employs an around advice to replace each such call with a call to a constructor of another class.

Figure 8 shows how Cuckoo's Egg may be applied to replace OldClass constructions with NewClass construction. A Cuckoo's Egg pattern uses the pointcuts defined in an existing Border Control pattern. Cuckoo's Egg is an advice design pattern and it was combined with Border Control without having to change it. This suggests that combining an advice pattern with an existing pointcut pattern can be made without changes in the existing pointcut pattern.

Recall from Section 3.2 that NewClass must be a subtype of OldClass; otherwise, we will get a class cast exception on the first attempt to instantiate OldClass. Moreover, we need NewClass to be a subtype of OldClass to make it compatible with the existing references to OldClass to which it would be assigned. This can be achieved either by defining inheritance directly in NewClass or by using a declare parents inter-type declaration (presumably, though not necessarily, in the OldClassDeprecation aspect itself).

5 Regularity in Aspect-Oriented Design Pattern Combination

As we will see in this section, the combination of aspect-oriented design patterns is substantially affected by their structural category (defined in Section 3.4). Under a combination of two patterns we understand a subsequent interrelated application of two patterns to a problem at hand. In other words, one of the patterns is applied to the problem, and afterwards another one is applied in connection to the artifacts of the former pattern.

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. An example of a pattern that can be combined with almost any other pattern is Exception Introduction, which represents an advice pattern. Exception Introduction can simply be added to the program without having to make any change to already applied patterns.

Another pattern that can be used with other, already applied patterns without having to make any changes to them is Policy, which is an inter-type declaration pattern. This pattern defines a pointcut that captures the join points in a base concern or another pattern whose occurrence represents breaking of some policy. If such a joint point occurs, a compile error or warning is issued.

It is also possible to combine a pointcut pattern with another pattern of the same type without having to change that pattern. In such a combination, the new pattern will actually use the pointcuts of the al-

```
public aspect MyApplicationRegions {
    public pointcut TestingRegion(): within(com.myapplication.testing.+);
    public pointcut MyApplication(): within(com.myapplication.+);
    public pointcut ThirdParty(): within(com.myapplication.thirdpartylibrary.+);
}
```

Figure 6: The Border Control pattern used to partition code into regions.

```
public aspect OldClassDeprecation {
    protected pointcut allowedUse(): BorderControlAspect.ThirdParty() ||
        BorderControlAspect.TestingRegion();

    declare @constructor: OldClass.new(): @deprecated;
}
```

Figure 7: Combining Policy with Border Control.

ready applied pointcut pattern. A simple example of this would be combining a Wormhole pattern with an already applied Border Control pattern. This way, the Wormhole pattern would be able to use regions defined by the Border Control pattern in its own pointcuts.

However, combining a pointcut pattern with an already applied advice or inter-type declaration pattern usually requires a change of this pattern. An example of combining a pointcut pattern with an already applied inter-type declaration pattern has been presented in Section 4 where we had the Policy pattern applied and combined the Border Control pattern with it. Recall also from the same section that 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.

Assume the pointcuts of a particular non-pointcut pattern in a developing application can no longer be defined in a simple way because it is not certain whether the pattern should be applied to new classes. Such a pattern can be combined with a Participant pattern, which is a pointcut pattern, which would enable individual classes to declare participation in this pattern application. However, the implementation of a Participant pattern requires the already applied pattern code to be altered.

Figure 9 presents schematically the combinations of the aspect-oriented design patterns we discussed. Pattern category is indicated graphically: oval nodes represent advice patterns, rectangular nodes are pointcut patterns, and rhomboid nodes stand for inter-type declaration patterns. Where any pattern of the given category is applicable, its name is shown as asterisk. The edge direction corresponds to the direction of pattern application: an edge originates in the pattern being applied and ends in the pattern to which this pattern is applied to achieve pattern combination. Dashed edges mean no change of the pattern at the edge end is required, while solid lines mean the change is necessary.

In Figure 9 we see that combining an advice or inter-type declaration pattern can be done without having to change the already applied advice pattern. Combining a pointcut patterns with an already applied advice pattern requires changes of the advice pattern is required.

We consider obvious that a combination of a pointcut pattern with an existing pointcut pattern requires no change to the existing pattern: we simply define further pointcuts. Combining a pointcut pattern with an already applied advice or inter-type declaration pattern requires changes in the advice or inter-type declaration pattern since the combination assumes the use of pointcuts defined in the pointcut pattern by the patterns of the latter two categories. This is caused by the nature of pointcut patterns: they define pointcuts to be used by other aspects in the application. On the other hand, a combination of an advice pattern or inter-type declaration pattern with other patterns of any category can be in most cases achieved without changes to the already applied design pattern.

Table 1 summarizes the identified dependence of the aspect-oriented pattern combination on the aspect-oriented pattern category. The columns represent the already applied patterns. The rows are the patterns to be applied. The values in cells indicate whether the already applied patterns have to be modified in case of their combination with the pattern in the correspond-

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

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

Figure 8: Combining Cuckoo's Egg with Border Control.

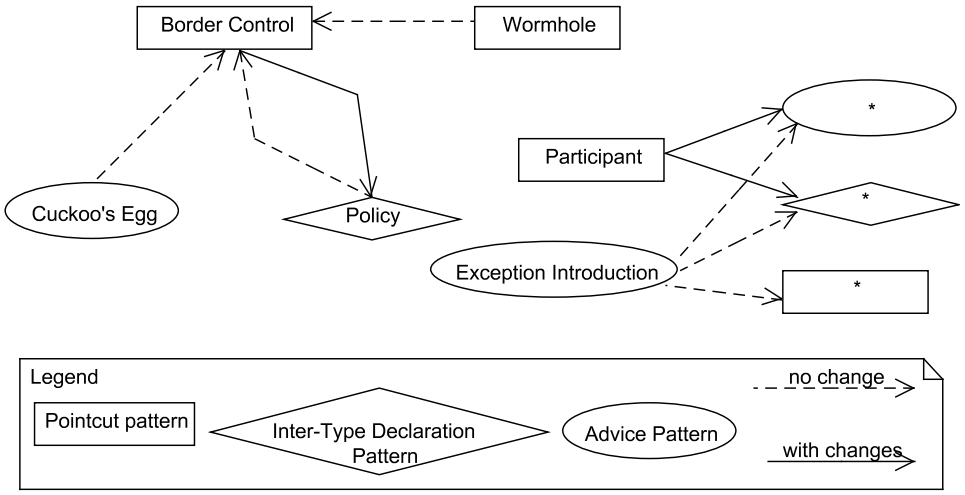

Figure 9: Combinations of aspect-oriented design patterns and required changes.

ing row.

6 Related Work

Hanenberg et al. present a set of AspectJ idioms[2] and a scheme for their interrelated application (Hanenberg et al., 2003). Similarly to the well-known scheme of GoF patterns (Gamma et al., 1995), it is represented by a graph in which patterns that can be combined are connected by directed edges. Each edge is annotated with the role of the pattern in which it originates plays in the pattern in which the edge terminates. However, no attempt is made to categorize the idioms.

Although intrinsic aspect-oriented design patterns are different than object-oriented patterns, we may encounter a certain analogy between our categories and those proposed by Gamma et al. (Gamma et al., 1995). Thus, 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. It has to be admitted that inter-type declarations correspond to creational patterns to a lesser extent, but we may see them as patterns of creating new elements and relationships.

Class deprecation in team development can be seen as a change and as such it is related to a broader area of change control. Dolog et al. describe how aspects can be used to capture change (Dolog et al., 2001) and apply their approach to improve product customization. Captured in an aspect, a change becomes pluggable and reapplicable. The reapplication of a change implemented as an aspect to a new product version in its simplest form takes only including the aspect in a build. Conventional approach would require performing a manual change extraction from the old version and its manual integration into the new version.

[2]Although denoted as idioms, they are applicable to PARC style aspect-oriented languages as much as the patterns presented in this paper.

Table 1: Aspect-oriented categories and their combinations.

combined with →	Pointcut Pattern	Advice Pattern	Inter-Type Declaration Pattern
Pointcut Pattern	no change	with changes	with changes
Advice Pattern	no change		
Inter-Type Declaration Pattern	no change		

7 Conclusions and Further Work

In this paper, we proposed a categorization of aspect-oriented design patterns according to their structure into three categories: pointcut, advice, and inter-type declaration patterns. This categorization is particularly useful in determining whether a combination of an aspect-oriented design pattern with another, already applied pattern requires a change in this pattern.

We studied the combination of aspect-oriented design patterns of different categories with respect to the stability of the already applied patterns in detail on the combination of Policy, Border Control, and Cuckoo's Egg applied to the class deprecation problem in team development.

Our further work involves exploring the possibilities of employing aspect-oriented design patterns and their combinations in capturing changes in a pluggable and reapplicable way in general and in application customization in particular. Class deprecation treated in this paper is actually one such change. It would also be interesting to explore the possibilities of the guided design pattern instantiation (Marko, 2004) with respect to aspect-oriented patterns and making use of the combination constraints based on aspect-oriented pattern categories in such a process.

We will also seek further parallels between categorization of GoF object-oriented design patterns and our categorization of aspect-oriented design patterns by exploring aspect-oriented implementations of GoF object-oriented design patterns (Hannemann and Kiczales, 2002).

ACKNOWLEDGEMENTS

This work was supported by the Science and Technology Assistance Agency under the contract No. APVT-51-024604 and by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/3102/06.

REFERENCES

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press. Cited in (Coplien, 2004).

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

Dolog, P., Vranić, V., and Bieliková, M. (2001). Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hanenberg, S., Schmidmeier, A., and Unland, R. (2003). Aspectj idioms for aspect-oriented software construction. In *Proceedings of 8th European Conference on Pattern Languages of Programs, EuroPLoP 2003*, Irsee, Germany.

Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. In *Proc. of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Aksit, M. and Matsuoka, S., editors, *Proc. of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, Jyväskylä, Finland. Springer.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning.

Marko, V. (2004). Template based, designer driven design pattern instantiation support. In *Proceedings of 8th East European Conference on Advances in Databases and Information Systems, ADBIS 2004*, Budapest, Hungary.

Miles, R. (2004). *AspectJ Cookbook*. O'Reilly.

Schmidmeier, A. (2004). Patterns and an antiidiom for aspect oriented programming. In *Proceedings of 9th European Conference on Pattern Languages of Programs, EuroPLoP 2004*, Irsee, Germany.

Appendix C

Attached CD Contents

Attached CD contains electronic version of this document and a full implementation of all examples from Chapter 4 in a sample application. CD also contains Java Runtime Environment which is needed to run the application. Eclipse, an open development platform, is also provided for browsing trough the application. Further instructions can be found in the readme file stored on CD.