VILNIUS UNIVERSITY

Žilvinas Vaira

INVESTIGATION, IMPROVEMENT AND DEVELOPMENT OF ASPECT-ORIENTED DESIGN PATTERNS

Doctoral dissertation

Technological sciences, informatics engineering (07 T)

Vilnius, 2012

Dissertation has been prepared during the period 2007 – 2011 at the Vilnius University.

Scientific supervisor:
Prof Dr Albertas Čaplinskas (Vilnius University, Technological Sciences, Informatics Engineering – 07 T).

VILNIAUS UNIVERSITETAS

Žilvinas Vaira

ASPEKTINIO PROJEKTAVIMO ŠABLONŲ TYRIMAS, TOBULINIMAS IR KŪRIMAS

Daktaro disertacija

Technologijos mokslai, informatikos inžinerija (07 T)

Vilnius, 2012

Disertacija rengta 2007 – 2011 metais Vilniaus universitete.

Mokslinis vadovas:
prof. dr. Albertas Čaplinskas (Vilniaus universitetas, technologijos mokslai, informatikos inžinerija – 07 T).

Acknowledgments

I would like to express my thanks to all the people who have been in one way or another involved in the preparation of this thesis.

First of all, I would like to thank my scientific supervisor Prof Dr Albertas Čaplinskas for support and guidance throughout the process of this dissertation research. He managed to find the time and energy to read probably every draft of this thesis and untangle "crosscutting concerns" of my scientific writing. I am very grateful to him for pushing further my ideas and significantly increasing the quality of my work. I am also grateful for the patience and constructive feedback from other members of staff in the Software Engineering Department of Institute of Mathematics and Informatics.

I also wish to thank Software Engineering Research Group headed by Prof Jacques Pasquier for providing SimJ framework for experimental application. Personal thanks to Prof Jacques Pasquier, Dr Patrik Fuhrer and Minh Tuan Nguyen for inspiration and guidance of the initial research that finally evolved into this dissertation.

I greatly appreciate the time and effort of committee members and especially the reviewers Prof Dr Rimantas Butleris and Prof Dr Saulius Gudas who read and commented on an initial version of this thesis.

I would also like to thank Prof Dr Vitalij Denisov for the support and assistance, Dr Dalia Baziukė for setting an example I can follow and other members of staff in the Department of Informatics in Klaipėda University.

Special thanks to Kristina Pociuvienė, who helped me with the proofread of my papers and this thesis.

Finally, I wish to thank my parents for all of their support and tolerance during this challenging period of my life. I also want to thank my brothers and all friends for helping me to relax. I would like to particularly thank my girlfriend Laura for her friendship and love during the writing of my thesis and especially when I had little time for her.

Abstract

Software systems are permanently changed in order to meet new requirements and to adapt them to permanently changing technology. Design modularity decouples design concerns that probably can be changed and in this way facilitates further system changes. Unfortunately, some design concerns, called crosscutting concerns, cannot be modularized using traditional modularization methods and techniques. Modularization of crosscutting concerns is the research subject of the new emerging software engineering paradigm, aspect-oriented analysis and design. However, this paradigm is not mature enough yet. In particular, it is still unknown which design patterns developed in the object-oriented paradigm can be adapted for aspect-oriented paradigm and how to transform them from one paradigm to another in a systematic way. Despite the fact that some attempts have been made to solve this problem, the proposed solutions just only eliminate crosscutting concerns in the investigated object-oriented design patterns, but do not generate pure aspect-oriented patterns. In addition, these solutions are ad hoc ones. No systematic procedure has been proposed for this aim so far. One more problem is the application of pure aspect-oriented patterns in the design of aspect-oriented domain frameworks. Although such patterns allow to use abstract aspects in the design of hot spots as well as to eliminate additional crosscutting concerns in the frameworks, the properties of final result – complexity of program code, its performance, etc. – have not been investigated properly yet. The thesis defines the class of object-oriented design patterns which can be transformed into pure aspect-oriented ones, proposes a systematic procedure for such transformation and investigates properties of resulting patterns from the viewpoint of their applicability in the design of aspect-oriented domain frameworks. This is the main contribution of the research work. The case study methodology has been used for the experimental research of the properties of aspect-oriented domain frameworks designed or redesigned using the proposed approach. Two aspect-oriented domain frameworks – simulation framework SimJ and web application

framework SimpleW – have been investigated. The first one has been redesigned from object-oriented framework developed by Software Engineering Group at Fribourg University and the second one has been developed from the scratch. The experimental research has demonstrated that the proposed approach can be successfully applied to real-word applications, facilitates the design of aspect-oriented frameworks and improves their quality.

Contents

List of Figures

List of Tables

List of Examples

Glossary

Adaptive programming – adaptive programs likewise object-oriented programs consist of a structural definition and behavioural definition but are different in a way that class structures are described only partially, giving a number of constraints that must be satisfied and that behaviour is not implemented exhaustively (Lieberherr et al., 1994).

Advice – the construct that is responsible for taking actions in the places defined as joint points.

Application framework – a framework covering a functionality that can be applied to different domains. According to (Johnson, 1988), an application framework is a reusable „semi-complete" application.

Architectural pattern – a high-level structure that contains a set of predefined sub-systems defines the responsibilities of each sub-system and details the relationships between sub-systems (Buschmann, 1996).

Aspect – represents crosscutting concern in the form of one or several aspects of a concrete concern.

Aspect weaving – the process of aspect compilation, named due to similarity to the real-life weaving process.

Aspectization – the redefinition of OO design patterns in terms of AO paradigm.

Aspectized AO design pattern – implementation of the OO design pattern in some OO language, for example in Java, is directly replaced by the analogous code written in some AO language, for example, in AspectJ (Hannemann, Kiczales 2002).

Base program – a program developed by the programming language of the paradigm on top of which aspect-oriented paradigm is used and which complements base program paradigm by providing new type of modularity.

Black-box framework – in such frameworks composition is the predominant technique used to design hot spots. A black-box framework does not require a deep understanding of the framework's implementation because the behaviour is extended by composing objects together and delegating behaviour between objects.

Case study – the case study is an empirical research method that aims at investigating some phenomena in his context (Runeson, Höst, 2009).

Class – in object-oriented programming is a construct that describes a type of object.

Class library – set of predefined dynamically loadable classes used to develop applications.

Code skeleton – describes the overall architecture of an application, that is, its basic components and the relationships between them. Typically, the skeleton is constructed from a collection of interfaces and abstract classes, which together specify the structural and behavioural relationships that the framework supports.

Composition (of objects) – defines a way of composing objects together, and delegating behaviour between objects. Delegation is the idea that instead of an object doing something itself, it gives another object the task.

Composition (of patterns) – defines a way of composing design patterns together. Compositions of patterns can by divided into 4 categories: invocation-based composition, class-level interlacing when the implementations of two patterns have one or more classes in common, method-level interlacing when the implementations of two patterns have one or more methods in common, overlapping when the implementations of two patterns share one or more statements, attributes, methods, and classes.

Composition filters – software development approach that similarly as AOP aims to solve a number of obstacles not properly addressed by the current object-oriented languages.

Conceptual analysis – the analysis of concepts, terms, variables, constructs, definitions, assertions, hypotheses, and theories that involves examining these for clarity and coherence, critically scrutinizing their logical relations, and identifying assumptions and implications (Machado, Silva, 2007).

Concern – some distinct part of a system, its cohesive functionality or properties.

Constructive research – a research procedure for producing innovative constructions, intended to solve the problems encountered in the real world and to make some contribution to the theory of the discipline in which it is applied (Lukka, 2003; Crnkovic, 2010).

Control flow – namely, a flow of control that refers to the execution order of statements, instructions, or function calls in a program of an imperative or a declarative programming language.

Critical case – an extreme case that is suitable to test hypotheses in critical situations.

Crosscutting concerns – in programming are considered as an unwanted result of code tangling and scattering.

Design pattern – a way of reusing abstract knowledge about a design problem and its solution. To be more exact, the design pattern in an abstract way describes a set of solutions to a family of similar design problems (MacDonald et al., 2002).

Domain framework – a framework capturing knowledge and expertise in a particular problem domain. Frameworks are built for various purposes and usually they are specific to one or several domains. Sometimes domain frameworks are referred to as enterprise application frameworks.

Dynamic crosscutting – crosscutting behaviour of a system defined directly by pointcuts and advice.

Encapsulation – surrounding objects with a common interface in a way that makes them interchangeable and hides their states from direct access.

Event-based system – "*a system in which the integrated components communicate by generating and receiving event notifications"* (Fiege, 2002).

Framework – a software framework is a reusable „semi-complete" software construction that can be finished, specialized and selectively changed by users in order to develop applications, software products and solutions.

Frozen spot – the unchangeable parts of the skeleton or class libraries in frameworks.

Functional programming – a software engineering paradigm that treats programming as the evaluation of mathematical functions and which data is immutable or treated as such.

Generative programming – defines approaches of automation of software development (Czarnecki, 2000).

Granularity – measurement of how system is broken down into smaller parts, smaller but greater number of entities means increase of granularity.

Hook – a mechanism allowing users to customize a framework by tapping into and modifying its inner workings. Customization can be done by composing and subclassing existing classes and/or by defining implementations of abstract operations.

Hot spot – a part of a framework where new application-oriented functionality can be added or in some other way customized. Hot spots provide one or several hooks (usually abstract operations) allowing to customize hot spots.

Idiom – the lowest level patterns that are language specific reoccurring solutions to common programming problems.

Inheritance – is a way to reuse code by sub typing from more abstract classes.

Instance – an exemplar of a concrete object.

Intentional programming – "*an extendible programming environment under development at Microsoft Research since early nineties … that supports the development of domain-specific languages and generators of any architecture … in a unique way*" (Czarnecki, Eisenecker, 2000).

Inter-type declarations – declarations that are made by aspects for defining a type: interface, class or aspect. It consists of member or method introductions, type-hierarchy modifications and is used to implement so called static crosscutting.

Interface – a set of predefined operations used to communicate for objects with each other. Java programming language provides direct construct for defining interfaces in other OOP languages it can be performed by abstract classes.

Join point – is a one of many points in a system where concerns crosscut.

Native AO design pattern – a native AO solution that is introduced to the same problems that are addressed by the OO design pattern (Hachani, Bardou, 2002; Hirschfeld et al., 2003).

Logic programming – software engineering paradigm that uses mathematical logic for computer programming.

Meta-programming – can be described by a phrase "*a program that manipulates another program is clearly an instance of meta-programming*" (Czarnecki, Eisenecker, 2000).

Object – refers to a particular instance of a class.

Paradigm – "*a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated; broadly: a philosophical or theoretical framework of any kind*" (Merriam-Webster, 2011)

Paradigm-independent design problem – an abstract design problem that may occur in several software engineering paradigms and solution of such a problem defined for particular paradigm can always by described by the constructs of that particular paradigm.

Paradigm-specific design problem – design problem that is based on specific paradigm related design constraint and may occur in one particular software engineering paradigm.

Pointcut – it is a part of aspect construct that represents affected join points. It also can be described as some sort of a query for selecting required join points.

Programming paradigm (software engineering paradigm) – that is related to some "general rule for attacking similar problems", has "their user communities" and becomes "embodied in the programming languages". (Floyd, 1979)

Pure AO design pattern – a design pattern that solves paradigm-independent design problem, which solution consists only of aspects.

Rule-based system – an approach used to design a system that stores and manipulates the knowledge in order to interpret information in a useful way.

Separation of concerns – the process of modularization of the crosscutting behaviour of concerns.

Static crosscutting – corresponds to the crosscutting of the static structure of the types that is implemented by inter-type declarations of aspects. Static crosscutting is not directly affected by pointcuts and advices.

Subject-oriented programming – a software engineering paradigm in which the behaviour and state of objects is considered as an extrinsic features of objects, as some kind of subjective views.

Supporting framework – frameworks that address specific, computer-related domains such as memory management or file systems. Support for these kinds of domains is necessary to simplify program development. Typically, such frameworks are used together with domain and/or application frameworks and support some internal mechanisms of the later.

Typical case – a representative case that is suitable to test hypotheses in usually occurring situations.

White-box framework – a framework where inheritance is the predominant technique used to design hot spots. The customization of a white-box

framework requires understanding the internals of the framework because its behaviour is extended by creating subclasses, taking advantage of inheritance. *Wildcard* – represents a characters that substitutes for other characters in regular expressions and can be used for the naming conventions to optimize pointcuts in AOP programs.

Abbreviations and Acronyms

AO	aspect-oriented.
AOP	aspect-oriented programming.
AspectJ	first aspect-oriented programming language proposed by (Kiczales et al., 2001).
C#	C Sharp, object-oriented programming language proposed by (Hejlsberg, 2003).
CERN	European Organization for Nuclear Research.
CoR	Chain of Responsibility design pattern.
Eclipse	an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle (http://www.eclipse.org).
GoF	Gang of Four, four authors of the (Gamma, et al. 1994) book.
GoF design patterns	23 object-oriented design patterns of the Gang of Four (Gamma, et al. 1994) book.
GoF_{AO} design patterns	object-oriented design patterns of the Gang of Four (Gamma, et al. 1994) book that were transformed from to design aspects.
GoF_{*AO} design patterns	GoF_{AO} design patterns that solve object creation problems (creational design patterns).
Java	object-oriented programming language proposed by (Gosling, 2005).
JBoss AOP	Java aspect-oriented supporting framework (Fleury, Reverbel, 2003).
JHotDraw	Java GUI framework for technical and structured Graphics (www.jhotdraw.org).
LISP	is the one of the oldest high-level programming languages specified in 1958, the name derives from the phrase “list processing”.
NetBeans	an open-source integrated development environment for

	software development (www.netbeans.org).
OO	object-oriented
OOP	object-oriented programming.
Python	general purpose programming language proposed by (Rossum, 1993).
SimJ	OO domain framework purported to design discrete events based simulation applications.
SimpleW	OO domain framework purported to simplify the design of personal web portals.
SoC	Separation of Concerns metric.
Spring AOP	one of the key components of Spring framework enabling technology to implement custom aspects (Laddad, 2010).
UML	Unified Modelling Language (Fowler, 2003).

Introduction

Research Context and Challenges

Mainly, software systems are permanently changed in order to meet new requirements and to adapt them to permanently changing technology. Design modularity decouples design concerns that probably can be changed and in this way facilitates further system changes (Bertrand Meyer, 1997). Object-oriented (OO) software engineering paradigm proposes a number of powerful modularization methods and techniques. Unfortunately, some design concerns, called crosscutting concerns, cannot be modularized using these methods and techniques. The solution of this problem has been proposed by the new emerging software engineering paradigm, aspect-oriented (AO) paradigm (Kiczales, et al., 1997). This paradigm proposes also the solutions of some other software engineering problems that have poor or even no solution in the OO paradigm. For example, one of such problems is the encapsulation of the multiplicity of subjective views in objects. It is very troubling to model several views by one object because different views require that, depending on the view, different properties of the same object would be accessible (Harrison, Ossher, 1993). AO paradigm proposes an elegant solution of this problem. However, this paradigm is still not enough mature. In particular, it is still unknown which design patterns developed in the object-oriented paradigm, for example, design patterns investigated by Gamma et al. (Gamma et al., 1994), can be adapted for aspect-oriented paradigm and how to transform them from one paradigm to another in a systematical way. Gamma et al. are often referred to as the Gang of Four, or GoF, and the patterns investigated by them as GoF design patterns.

Object-oriented design patterns have been developed as a result of in-depth analysis and generalization of best object-oriented design practices. The concept of design pattern has been highly influential to the field of software engineering, first of all, to object-oriented design theory and practice.

However, 23 GoF and other OO design patterns have been investigated only in the narrow context of OO paradigm and the extent of their applicability in other paradigms is still an open research question. Although some researches (Hannemann, Kiczales, 2002; Hachani, Bardou, 2002; Hirschfeld et al., 2003) investigate how the 23 GoF design patterns can be rewritten in an AO manner, no one investigated systematically the problem of transformation of OO design patterns into analogous design patterns in other software engineering paradigms. It means that it is still unknown which design patterns can be applied to solve paradigm independent design problems and which are paradigm-specific, therefore meaningless in other paradigms. In particular, it is very important to answer this question at least for OO and AO paradigms. It is important from both theoretical and practical points of view. OO design patterns have been extracted analyzing a huge amount of successful designs. Although there are ways proposed how to perform some automatic inference of new design patterns (Tonella, Antoniol, 1999), their acceptance is still directly related to successful application of common design ideas many times in many projects. Such pattern gathering process is very slow and expensive. It would not be reasonable to repeat this process for AO paradigm from scratch. It is obvious that it is preferable to rely on the experience gained in other software engineering paradigms, first of all, in OO paradigm and to adapt for AO paradigm the design patterns developed and well-investigated there.

Problem Statement

The subject of the thesis research is pure AO design patterns and their application in the design of AO frameworks. By pure AO design patterns the patterns implemented using aspects only are considered. Mixed AO design patterns, in contrast, are such patterns which are implemented using both, aspects and objects. Usually in mixed design patterns aspects play supporting role and mainly are used only to eliminate concern crosscutting in the pattern implementation code.

The research aims to identify these GoF design patterns that solve OO paradigm independent design problems, to develop techniques for transformation of such patterns to pure AO design patterns, and to investigate the properties of AO domain frameworks developed using the resulted design patterns.

Motivation

Aspect-oriented programming (AOP) emerged as a stand alone paradigm already almost 15 years ago (Kiczales, et al., 1997). However, still very few widely accepted and well documented pure AO design patterns have been proposed. Up to time, even basic OO design patterns – 23 GoF patterns – have not been transformed into pure AO form. Moreover, the question is still open which GoF design patterns can be transformed into pure AO design patterns and why. Although some researches (Bynens, Joosen, 2009; Hanenberg, Schmidmeier, 2003; Laddad, 2003; Miles, 2004) proposed a number of paradigm dependent AO design patterns and idioms, and others (Hannemann, Kiczales, 2002; Hachani, Bardou, 2002; Hirschfeld et al., 2003) investigated how some of GoF design patterns can be redesigned as mixed AO design patterns, all these researches had a sporadic, ad hoc character and they still do not answer the above presented question.

On the other hand, the design patterns play essential role in the development of many applications, especially in the development of various frameworks. There exists a large and well documented experience of application of GoF design patterns in the design of OO frameworks (Adair, 1995; Appleton, 1997; Fayad, Schmidt, 1997; Johnson 1997; Kaisler 2005). It is evident that these and other design patterns facilitate and improve the design of frameworks, make their design documentation more transparent. This is true for OO as well as for AO frameworks. It means that there exist a strong need in pure AO analogues of GoF design patterns and the investigation of the impact of application of these patterns on the run-time properties of framework implementations.

Aims and Objectives of the Research

The research aims to define the class of object-oriented design patterns which can be transformed into pure aspect-oriented ones, proposes a systematic procedure for such transformation and investigates properties of resulting patterns from the viewpoint of their applicability in the design of aspect-oriented domain frameworks. In order to achieve these aims, the following research objectives have been stated:

- evaluate the state of affairs, compare existing approaches to the development of AO design patterns, and highlight their advantages and shortcomings;
- investigate which GoF design patterns solve such design problems that arise in AO paradigm and how these patterns can be transformed into pure AO design patterns;
- investigate applicability of such design patterns in the design of AO domain frameworks and the impact of their application on the complexity of the resulting code, its performance and other run-time characteristics.

Research Questions and Hypotheses

Main questions that need to be answered in this research are:

- How mature is the AO software engineering paradigm currently?
- What techniques can be used to develop AO design patterns and what advantages and shortcomings has each of these techniques got? Does the aspect-oriented paradigm generate some new patterns that are specific only to this paradigm?
- In which different ways design patterns can be implemented, when they solve paradigm independent design problems and design problems that are specific to object-oriented or aspect-oriented software engineering paradigms?
- Is it possible to implement at least some of GoF design patterns using aspect-oriented constructs only? Which and how, if it is possible? Is

such implementation in some way better than the object-oriented one? How to measure this?

- In which way are the aspects different as classes from the viewpoint of design patterns and what is the impact of such differences on the structure and other properties of pure AO design patterns?
- What are the advantages of application of pure AO design patterns in real-life applications in general and, particularly, in AO domain frameworks?
- What is the impact of application of pure AO design patterns in AO domain frameworks on the crosscutting, complexity of code implementation and framework run-time performance?

To answer these questions, the following hypotheses have been stated:

- there exist paradigm-independent design problems, at least in the context of OO and AO software engineering paradigms;
- aspect-oriented constructs are sufficient to implement those GoF design patterns that solve paradigm-independent design problems, despite the fact that aspects cannot be directly instantiated;
- efficiency of designs is improved by the usage of pure AO design patterns combined with GoF design patterns;
- the usage of pure AO design patterns allows designing of new kind of hot spots in white-box AO domain frameworks (i.e. hot spots represented by abstract aspects);
- the usage of pure AO designs patterns reduces crosscutting in AO domain frameworks;
- the development of AO domain frameworks using pure AO design patterns has no particular impact on the overall run-time performance of the applications developed using such frameworks.

Research Design and Research Methods

The research design of present thesis is of an exploratory nature. Aspect-oriented software engineering paradigm is relatively young and the research in

this area is still in its infancy. It means that relatively large amount of library research is required in order to define exact structure of a problem, to gain a better understanding of the environment within which the problem arises.

The exploratory nature of the research and the engineering nature of the research subject require that engineering methods would be used to solve the problem under consideration. In this context, the best candidate is constructive research.

Finally, according to (Cooper, Schindler, 1998), any exploratory research is mainly qualitative in his nature. For this reason, it is impossible to validate all obtained results quantitatively, by measurements, because qualitative factors cannot be measured in principle. Additionally, any dissertation research is a small-scale research from both financial and time points of view. It means that in such research it is too expensive and practically impossible ensure high statistical reliability and high level statistical significance of quantitative measurements in cases, when such measurements can be done. Thus, despite its possible biases, the case study methodology is the only practically acceptable methodology to validate the research results.

Taking into account all the discussed above, the research design provides three distinctive research phases, namely, conceptual analysis (Laurence, Margolis, 2003) of related work, constructive research that aims to develop the transformation techniques to transform GoF design patterns into pure AO design patterns, experimental investigation of the applicability of pure AO design patterns in the development of AO domain frameworks.

Conceptual analysis is the analysis of concepts, terms, variables, constructs, definitions, assertions, hypotheses, and theories. It involves examining these for clarity and coherence, critically scrutinizing their logical relations, and identifying assumptions and implications (Machado, Silva, 2007). The goal of conceptual analysis is to increase the conceptual clarity of the research subject. The primary utility of conceptual analysis is to determine the existing state of the research field so that further work may be strategically and appropriately planned (Penrod, Hupcey, 2004). The conceptual analysis of related works has

been carried out to generate important theoretical constructs and to provide the theoretical basis for further research as well as to prevent from performing a research that has already been done by others (Hart, 1998). The main field on which conceptual analysis has been performed encompasses both object-oriented and aspect-oriented software design patterns. Generally, conceptual analysis allows answering the questions how mature the AO software engineering paradigm currently is, in which way the aspects are different as classes from the viewpoint of design patterns and what is the impact of such differences on the structure and other properties of pure AO design patterns.

An essential part of conceptual analysis is the categorization of concepts. The categorization has been used as a base to define the class of object-oriented design patterns which can be transformed into pure aspect-oriented ones.

The constructive research approach is a research procedure for producing innovative constructions, intended to solve the problems encountered in the real world and to make some contribution to the theory of the discipline in which it is applied (Lukka, 2003; Crnkovic, 2010). The central notion of this approach, the novel construction, is an abstract notion with a great variety of potential realizations. Models, designs, methods, algorithms, and most other artefacts are considered as constructions. It means that they are invented and developed, not discovered. Mathematical algorithms and new mathematical entities are examples of theoretical constructions. The constructive research approach is based on the belief that by a profound analysis of what works (or does not work) in practice one can make a significant contribution to theory. In the present thesis this approach is used as a methodological basis to develop the transformation rules transforming GoF design patterns to their pure AO analogues, GoF_{AO} design patterns. It is probable that not all of 23 GoF design patterns have pure AO analogues and GoF_{AO} include less than 23 patterns.

As a result of profound analysis of the problem, it has been discovered that aspects are similar to singleton classes. This result suggests that classes in OO design patterns can be replaced by aspects. The details of such transformation should be investigated for each particular pattern and the findings should be

generalized in order to develop the rules applicable to all GoF design patterns. The similarities between classes and aspects suggest also that OO patterns, which cannot be implemented, using singleton classes only, cannot be transformed into GoF_{AO} patterns and, consequently, solve OO-specific design problems. Some design patterns out of 23 GoF patterns are dedicated to solve object creation problems. At first glance, the usefulness of such patterns in AO paradigm is highly questionable and should be investigated specifically, if even they can be transformed into GoF_{AO} patterns. Such class patterns are denoted by GoF_{*AO}.

The constructive research methodology is used also for testing of working hypotheses that has been provisionally accepted in the present thesis. One of the advantages of this methodology is that it allows not only to test and investigate the properties of the innovative construction but also to study its development process. On the other hand the constructive research, in parallel with some other methodologies of experimental research, can be viewed as a kind of case study methodology. However, according to the conventional view, case studies should be used for falsification of the hypothesis only. Case study itself cannot prove any hypothesis and should be linked to some hypothetico-deductive model of explanation. However, the correspondence of the case study to real-world situations and its multiple wealth of details state that this view is only partly correct (Flyvbjerg, 2004). Taking into account this argument and the fact that the dissertation research is a small-scale research from both, financial and time points of view, the case study methodology has been approved as the main hypothesis testing methodology. Mainly, the case study is an empirical research method that aims at investigating some phenomena in his context (Runeson, Höst, 2009). In present thesis the aim is to investigate the impact of application of pure AO design patterns on the design of AO domain (white-box) frameworks.

According to (Ragin, 1992) case studies can be enhanced by the strategic selection of cases: critical or typical. A critical case can be thought as an extreme case that is suitable to test hypotheses in critical situations. The case of

such AO domain framework, which is designed using at least one GoF_{*AO} design pattern, has been chosen as a critical case. In addition, two typical cases have been chosen: redesign of an existing OO domain framework into an AO domain framework using GoF_{AO} patterns and the design of a new AO domain framework using GoF_{AO} patterns.

The first typical case is constrained by the existing design of the OO framework and allows investigating the consequences of the redesign when a part of object-oriented framework design has been replaced by relevant pure AO design patterns. Only the parts of the framework that have been affected by some crosscutting of concerns have been reworked. The second typical case has no preliminary design constrains and allows choosing any design that is most suitable for designing aspects. As the result, three cases have been studied.

Generally, quantitative and qualitative data collection methods can be used for evaluation of the results of any case study. Quantitative data relies on numbers that are analyzed using statistics. Qualitative data relies on the text, diagrams and pictures that are analyzed using categorization and sorting. In case studies qualitative data analysis is used more often. The usage of both, qualitative and quantitative data, complimentary provides stronger evidence for the evaluation of the hypotheses (Runeson, Höst, 2009). Thus, both approaches have been used.

The main steps of applied case study methodology can be summarized shortly as follows:

1. identify the aspects that should be designed;
2. decide what design patterns should be applied in order to design identified aspects;
3. design and implement aspects, document observations and findings, and collect other qualitative data;
4. perform measurements, test the code and collect quantitative data;
5. evaluate the structure of the code according to criteria;
6. analyze, generalize the collected data and evaluate hypothesis.

Summary of Research Results

The results of the thesis research can be summarized as follows:

- The hypothesis has been proven that there exist paradigm-independent design problems, at least in the context of OO and AO software engineering paradigms.
- There has been identified the subset of 23 GoF object oriented design patterns (20 GoF patterns) which solve paradigm-independent design problems and can be transformed into pure AO design patterns (GoF_{AO} patterns).
- The hypothesis has been proven that aspect-oriented constructs are sufficient to implement 20 of GoF_{AO} design patterns, with regard that 5 of them are exposed to some reduced applicability.
- The rules have been proposed how to transform 20 GoF design patterns into GoF_{AO} design patterns.
- The hypothesis has been validated that the usage of GoF_{AO} design patterns (next to 23 GoF design patterns) improves the efficiency of domain frameworks designs.
- The hypothesis has been proven that the usage of GoF_{AO} design patterns allows designing a new class of hot spots in white-box AO domain frameworks, (namely, hot spots represented by abstract aspects).
- The hypothesis has been validated that the usage of GoF_{AO} designs patterns reduces crosscutting in AO domain frameworks.
- The hypothesis has been rejected that the development of AO domain frameworks using GoF_{AO} design patterns has no particular impact on the overall run-time performance of the applications developed using such frameworks.

Contributions of the Dissertation

The present thesis is one of the first researches that aims to investigate pure AO design patterns and the application of such patterns in the design of AO domain frameworks. Although several attempts (Arpaia, et al, 2008; Santos et

al., 2007; Kulesza et al., 2006) to design customizable aspects in frameworks have been made, none of them investigates the use of pure AO design patterns to design aspects as hot spots and none of them examines the design of AO frameworks in such detail. It is also the first work that states the question about the existence of design problems which are common to all or, at least, to several software engineering paradigms. Finally, the case study methodology applied in present thesis supports the empirical research approach in which constructive research and case study research methods can be used to validate hypotheses in software engineering.

The practical significance of the thesis is as follows:

- 20 pure aspect-oriented design patterns, that have been developed in the thesis research, can be applied developing any aspect-oriented domain frameworks as well as other aspect-oriented applications;
- the thesis demonstrates how abstract aspects should be designed so that to be applicable as hot spots in aspect-oriented domain frameworks.

Approbation

The main results of the thesis were presented and approved at the following conferences:

- 3rd International Conference on Pervasive Patterns and Applications, PATTERNS 2011, September 25-30, 2011 – Rome, Italy;
- 15th Conference of Lithuanian Computer Society "Computer Days – 2011", September 22–24, 2011, Klaipeda, Lithuania;
- 50th Conference of Lithuanian Mathematicians Society, June 18–19, 2009, Vilnius, Lithuania.
- 12th Student Scientific Society conference "Fundamental Research and Innovation in Science Integration". Klaipeda University Faculty of Natural Science and Mathematics, 2009, Klaipeda, Lithuania.

Outline of the Dissertation

The text of the thesis consists of introduction, 5 main chapters, conclusions, list of references, list of publications and appendixes. Main chapters are provided with summary and (except Chapter 1) with conclusions.

Introduction describes research context and challenges, presents the problem statement, discusses motivation, aims and objectives of the research, states research questions and hypotheses, describes research design and research methods, research results, contributions of the thesis, and approbation of obtained results.

Chapter 1 presents preliminaries on design patterns, aspect-oriented paradigm and frameworks.

Chapter 2 describes the results of critical analysis of related works.

Chapter 3 develops and discusses main theoretical results of the research. It proves the hypothesis that there exist, at least in the context of OO and AO software engineering paradigms, paradigm-independent design problems, identifies the subset of 23 GoF object oriented design patterns (20 GoF patterns) which solve paradigm-independent design problems and can be transformed into pure AO design patterns (GoF_{AO} patterns), proves the hypothesis that aspect-oriented constructs are sufficient to implement 20 of GoF design patterns, with regard that 5 of them exposes some reduced applicability, and presents the rules to transform 20 GoF design patterns into GoF_{AO} design patterns.

Chapter 4 describes in details case studies on application of the transformed design patterns to design frameworks and validation of research hypothesis. It validates the hypotheses that the usage of GoF_{AO} design patterns (next to 23 GoF design patterns) improves the efficiency of domain frameworks designs, that the usage of GoF_{AO} design patterns allows designing a new class of hot spots in white-box AO domain frameworks, namely, hot spots represented by abstract aspects, that the usage of GoF_{AO} designs patterns reduces crosscutting in AO domain frameworks, and that the development of AO domain frameworks using GoF_{AO} design patterns has no particular impact on the

overall run-time performance of the applications developed using such frameworks.

Chapter 5 discusses some open questions and limitations.

Conclusions present the main conclusions of the dissertation.

Appendixes presents preliminaries about AspectJ programming language, list of remaining transformed GoF design pattern descriptions and extended versions of several diagrams.

Chapter 1

Preliminaries

The chapter defines details about the terminology and the concepts used in the thesis. **Section 1** provides a definition and the scope of the design patterns used in this research. **Section 2** discusses general concept of programming paradigm, presents the main principles of aspect-oriented programming and the syntax of AspectJ programming language. **Section 3** analyzes the concept of the framework. It determines the kind of the framework that is investigated further in present thesis and discuses in short the approach proposed to design frameworks using hot spots and hooks.

1.1. Design Patterns

In software engineering, more exactly, in the object-oriented programming, the concept of design pattern has been introduced by (Gamma, et al., 1994). The term was borrowed from the architectural terminology where it was coined by Alexander (Alexander, et al, 1977). Alexander explained the concept of design patterns in the following way:

> *"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."* (Alexander, et al, 1977)

Gamma et al. have accepted this understanding of design pattern and proposed to define object-oriented design patterns using four essential elements: pattern name, problem, solution and consequences. There are many different approaches to reuse, including the code, design and concept reuse. The latest is

supported by design patterns (Gamma et al., 1994). According to Robert Martin,

> *"At the highest level, there are the architecture patterns that define the overall shape and structure of software applications. Down a level is the architecture that is specifically related to the purpose of the software application. Yet another level down resides the architecture of the modules and their interconnections. This is the domain of design patterns"* (Martin, 2000)

A design pattern is a way of reusing abstract knowledge about a design problem and its solution. To be more exact, the design pattern in an abstract way describes a set of solutions to a family of similar design problems (MacDonald et al., 2002). It describes the idea of a design decision in the form that is sufficiently abstract to be reused in different settings. It can be said that a design pattern is a guideline how to design some element of a system. Design patterns do not influence the overall system architecture. They define the architecture of lower level constituents of a system, namely, subsystems and components. It should be noted that design patterns are not the lowest level patterns. The lowest level patterns are called idioms. They are language specific reoccurring solutions to common programming problems. According to Ramnivas Laddad,

> *"The difference between design pattern and idioms involves the scope at which they solve problems and their language specificity. From the scope point of view, idioms are just smaller patterns. From the language point of view, idioms apply to specific language whereas the design patterns apply to multiple languages using the same methodology."* (Laddad, 2003)

An example is the Java idiom for ending a program when the window is closed (Example 1). Mainly, the patterns define relationships between the entities in the implementation domain (Shalloway, Trott, 2001) or, in other words, some parameterized collaborations. However, it is difficult to develop a single body

of code or even a framework that adequately solves each problem in the family.

```
1  addWindowListener(
2    new WindowAdapter() {
3       public void windowClosing(WindowEvent e) {
4         System.exit(0);
5       }
6    }
7  )
```

Example 1 Java idiom for ending a program

Most design patterns represent families of solutions the structures of which cannot be adequately represented by a static framework (MacDonald et al., 2002). According to Aleksandra Tešanović,

> *"…a pattern is not an implementation, although it may provide hints about potential implementation issues. The pattern only describes when, why, and how one could create an implementation."* (Tešanović, 2004)

In other words, in any particular case the pattern should be adapted to the particular context.

Gamma et al. described 23 object-oriented design patterns using their four essential elements format (Gamma, et al., 1994). The structure of the design pattern is a part of design pattern essential element – solution. In (Gamma, et al., 1994) the structure of design pattern is represented using the early graphical form of UML (Booch, et al., 2000; Fowler, 2003) as collaborations. Namely this notation is used in the present thesis.

Since the authors of (Gamma, et al., 1994) are often referred to as the Gang of Four (GoF), the abbreviation GoF is used also to refer to these patterns. By analogy, the abbreviation GoF_{AO} is used in the thesis to refer to pure aspect-oriented design patterns that are analogous to GoF design patterns. (Gamma, et al., 1994) have also proposed the following categorization of design patterns by their design purpose: creational, structural, and behavioural.

The concept of design pattern has also been introduced in aspect-oriented programming (Hanenberg, Costanza, 2002; Hanenberg, Schmidmeier, 2003; Laddad, 2003; Schmidmeier, 2004; Miles, 2004; Griswold et al., 2006;

Lagaisse, Joosen, 2006; Bynens et al., 2007; Bynens, Joosen, 2009; Menkyna et al., 2010). These works will be discussed in detail in Chapter 3.

In summary, in software engineering, a design pattern can be defined as a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into code, but only a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically are collaborations, they show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. The application of design patterns is possible in different software engineering paradigms, however, is very limited in functional programming languages, in which data is immutable or treated as such, because many of currently used patterns imply mutable state.

1.2. Aspect-Oriented Software Engineering Paradigm

AO paradigm is one of the several software engineering paradigms which differ in the notion of algorithm and other details. Merriam-Webster dictionary defines the term paradigm as

> *"a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated; broadly: a philosophical or theoretical framework of any kind"* (Merriam-Webster, 2011)

In the field of computer science the term has been introduced by Robert W. Floyd in his 1978 Turing Award Lecture. According Robert W. Floyd the paradigm of programming can be defined as correspondent to the following statement:

> *Programming paradigm is related to some "general rule for attacking similar problems", has "their user communities" and becomes "embodied in the programming languages"* (Floyd, 1979)

The “rule for attacking similar problems” in this context can be understood as some refined and abstract program design method that is directly or indirectly supported by programming language. Programming paradigms differ in the concepts and abstractions to represent the elements of program and the notion of algorithm (steps that compose a computation). Most popular programming paradigms are procedural programming, logic programming, functional programming, object-oriented programming, and aspect-oriented programming (Ambler,et al., 1992). Many current programming languages (Java, C#, Python, Common LISP, etc.) are based on several programming paradigms, sometimes even on so different paradigms as object-oriented and functional programming. Such programming languages are called multi-paradigm languages (Wampler, Clark, 2010).

The concept of paradigm can be extended to be applicable not only to programming but also to analysis, design, testing and other activities related to software development process. Such extended paradigms are called software engineering paradigms. Sometimes a software engineering paradigm is understood also as a software development strategy or a software lifecycle model. However, in the present thesis the term “software engineering paradigm” is used in the first sense. Thus it is always referred to object-oriented, aspect-oriented and other extended programming paradigms.

A number of software engineering paradigms, methodologies and approaches exist today. Although the object-oriented (OO) paradigm still remains one of most popular, it is gradually replaced by the aspect-oriented (AO) one (Kiczales et al, 1997; Lopes, 2005). Mainly, it is because of the concern crosscutting problem. Object-oriented paradigm suffers from inability to separate crosscutting concerns. OO system may have and often has such properties that must be implemented by more than one functional component. It means that the implementation of such a property crosscuts the static and dynamic structure of the program. The AO paradigm solves this problem by the separation of concerns. However, the separation of concerns itself is not enough to develop a new mature software engineering paradigm. It is also

necessary to provide some solutions that allow coping with other important software engineering issues, including software reuse.

Some software system design problems are paradigm-independent. For example, the problem how to decouple the resource and its consumer does not depend on any particular software engineering paradigm. The proposed solution is the Façade pattern that suggests inserting of an abstract interface between the consumer and the resource (Martin, 2000). This idea is very abstract and also paradigm-independent. Originally, the Gang of Four (GoF) defined the intent of the Façade pattern as more narrow, only for subsystems but not for any resource:

> *"Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use."* (Gamma et al., 1994)

This idea is still paradigm-independent. However, it should be implemented in a paradigm-dependent way. It means that, first of all, it should be expressed in terms of a particular paradigm and can be implemented only afterwards. In other words, for each paradigm the patterns solving paradigm-independent design problems should be expressed in terms of this paradigm and it should be done in a compact way. For example, in OO paradigm the idea of the Façade pattern can be described as follows: define a new class that hides the interfaces of several other classes under the new unified interface. Since the description of the idea of pattern should be as compact as possible, the question which concepts should be used to describe this idea must be investigated for any particular pattern. Though often the concepts describing a design pattern in one paradigm (e.g. OO paradigm) can be expressed directly by concepts of some other paradigm (e.g. AO paradigm), it is questionable whether such translation is the best way to transform the design patterns from one paradigm to another.

Patterns that solve the paradigm-dependent design problems are not idioms. They are language-independent and still very abstract. Such patterns describe a set of solutions to a family of similar design problems and should be effectively expressed in the vocabulary of any programming language that is

based on this paradigm. The OO patterns, AO patterns and patterns for other paradigms eventually must be described in a paradigm-dependent way. It is reasonable that, despite the fact that in software engineering the patterns are often identified only with the object-oriented paradigm, some of them can be considered at a more abstract paradigm-independent level and specialized for any particular paradigm. Consequently, the patterns solving paradigm-independent design problems can be defined for a new paradigm in two different ways: by rewriting the patterns already defined for some paradigm (e.g. OO paradigm) in terms of a new paradigm (e.g. AO paradigm) or by generalizing the patterns already defined for some paradigm, defining them in a paradigm-independent way and then specializing such paradigm-independent definitions for new paradigms. It seems that the latter way is more promising. However, currently it is still unknown even, which of the 23 GoF design patterns solve paradigm-independent design problems and can be adapted to other paradigms. The present thesis investigates this question in the context of two paradigms, namely, OO and AO paradigms. Of course, the fact that some OO design patterns can be adapted to solve aspect design problems does not mean that they really solve paradigm-independent design problems. However they can be considered as candidates to do this.

Let us discuss the most important concepts and terms of the AO paradigm. The term *concern* in the context of AOP addresses any piece of interest or focus in a program. Typically, concerns are synonymous with features or behaviours (Laddad, 2010). Most of the concerns can be encapsulated using procedures, classes and other abstractions of traditional programming languages. However, some concerns are spread all over the system. Such concerns are called *crosscutting concerns*. The simplest form of the crosscutting explanation is that concerns are stated as crosscutting "*if the methods related to those concerns intersect*" (Elrad, 2001). The crosscutting concerns are considered as a harmful phenomena because of code tangling and scattering (Miller, 2001). A typical example of crosscutting concern is logging that is usually spread across several modules (Laddad, 2010) (Fig. 1). Other examples are failure handling,

coordination, synchronization, memory management, persistence, security, caching, monitoring, etc.

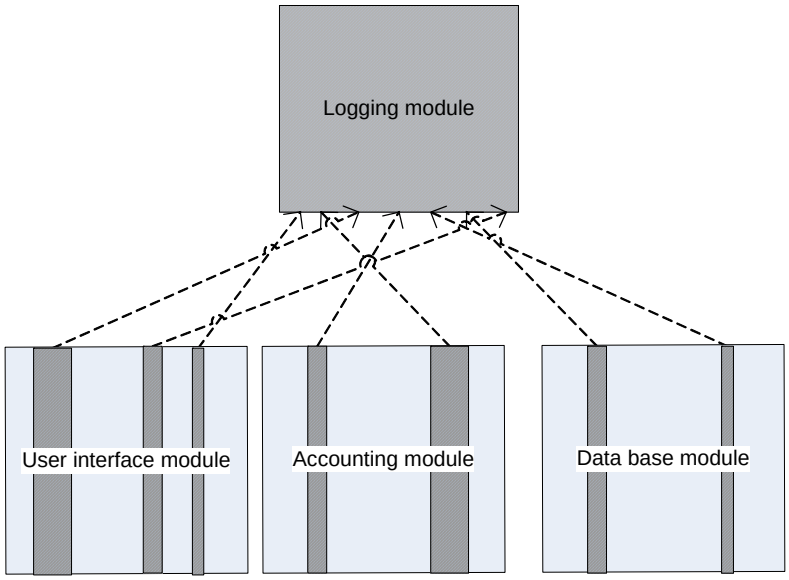


Fig. 1 Crosscutting concern

Aspect-oriented (AO) paradigm has been developed with the aim to deal with the problem of crosscutting concerns. This paradigm is built on top of the OO paradigm and complements this paradigm by providing new type of modularity that pulls together the widespread implementation of a crosscutting concern into a single unit termed *aspect*. In this way AO paradigm solves the problem of crosscutting concerns. Aspects of a system developed using AO paradigm can be changed, inserted or removed at compile time, and even reused. In order to affect regular class-based code referred to as base program, aspects must be woven into the code they modify. It is done by the special meta-programming utility regarded as *aspect weaver*. The weaver scatters aspect code across the classes affected by this aspect and interweaves this code with the code of corresponding classes. Fig. 2 illustrates the separation of concerns represented in Fig. 1. In Fig. 1 the change of a logger concern requires the change of method calls in other modules. In Fig. 2 other modules do not contain any calls to the logger module.

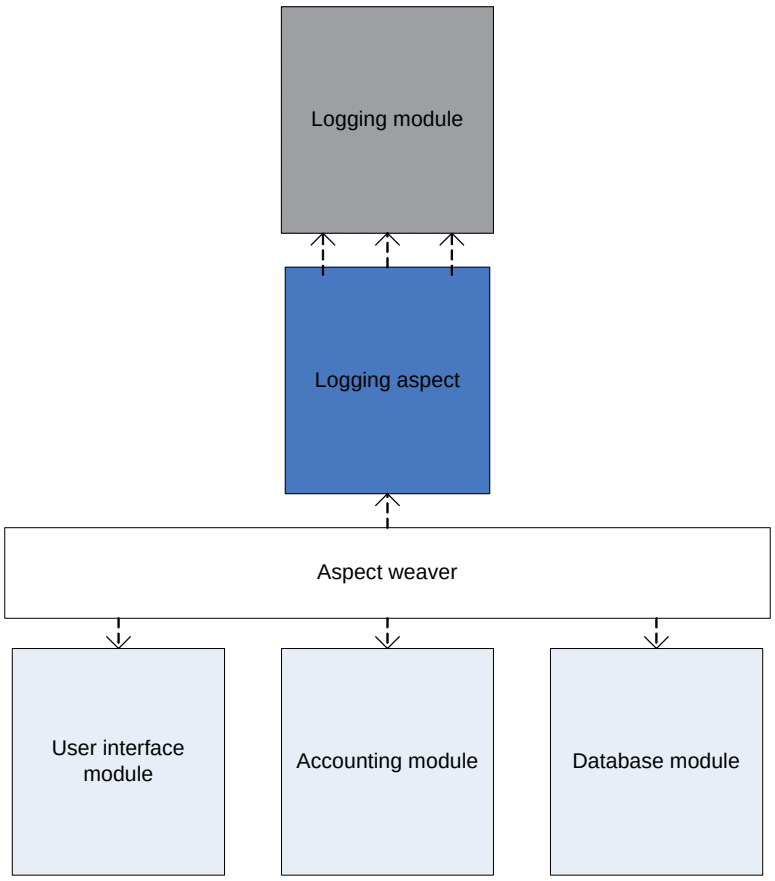


Fig. 2 Concern crosscutting handled by aspect

Aspects contain the information about the places where a necessary code should be weaved into the classes. The places or in other words points where the weaver should inject the code fragments are named *join points*. More precisely, join points are the points in the system where concerns crosscut. The information about the join points is held by a construct named *pointcut*. Pointcut takes a part of aspect structure and identifies references to affected join points. It is not necessary to write information about all join points separately, this can be shortened by using the so called wildcards or some logical similarities. The pointcut can be described as a query for selecting required join points.

An *advice* is the construct that is responsible for taking actions in the places defined as joint points. Advice contains the functionality that must be performed at the particular set of join points. This functionality mainly consists

of some calls to methods of crosscutting concern that must be weaved into other concerns.

Joint point, pointcut and advice are the basic concepts of AO paradigm. Their implementations may vary in different AO languages. In present thesis AO examples of implementation code are described using programming languages AspectJ (Laddad, 2010) and Java (Arnold, 2005). The preliminaries about AspectJ language are presented in APPENDIX A.

1.3. Frameworks

A software framework is a reusable „semi-complete" software construction that can be finished, specialized and selectively changed by users in order to develop applications, software products and solutions. Roughly, all software frameworks can be divided into three categories (Adair, 1995; Kaisler, 2005):

- *Application frameworks* – covering a functionality that can be applied to different domains. According to (Johnson, 1988), an application framework is a reusable „semi-complete" application.
- *Domain frameworks* – capturing knowledge and expertise in a particular problem domain. Frameworks are built for various purposes and usually they are specific to one or several domains. Sometimes domain frameworks are referred to as *enterprise application frameworks*.
- *Supporting frameworks* – frameworks that address specific, computer-related domains such as memory management or file systems. Support for these kinds of domains is necessary to simplify program development. Typically, such frameworks are used together with domain and/or application frameworks and support some internal mechanisms of the later.

A domain framework, which produces applications that are built from a collection of interacting objects, is referred to as an object-oriented domain framework. There are several definitions of an object-oriented domain framework. For example, Johnson and Foote define an object-oriented domain framework in the following way:

> *"A framework is a set of classes that embodies an abstract design for solutions to a family of related problems*." (Johnson, Foote, 1988).

Gamma et al. define it also in a similar way:

> *"A framework is a set of cooperating classes that make up a reusable design for a specific class of software."* (Gamma, et al., 1994)

In the present thesis the following definitions describing an object-oriented framework from two different perspectives have been accepted:

> *"… a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact."* (Johnson, 1997)
>
> *"… a framework is the skeleton of an application that can be customized by an application developer."* (Johnson, 1997)

The concept of an object-oriented framework is build around such fundamental object-oriented programming (OOP) concepts as class abstractions and class inheritance. According to (Froehlich et al, 1998), the general structure of an OO domain framework consists of fixed and variable parts. The fixed part includes class libraries and the code skeleton that defines the range of applications that a framework can support (Kirk, 2005). It describes the overall architecture of an application, that is, its basic components and the relationships between them. Typically, the skeleton is constructed from a collection of interfaces and abstract classes, which together specify the structural and behavioural relationships that the framework supports. The unchangeable parts of the skeleton or class libraries are called *frozen spots* (Froehlich et al, 1998). The parts of a framework where new application-oriented functionality can be added or customized in some other way are called *hot spots*. Hot spots provide one or several *hooks* (usually abstract operations) allowing to customize hot spots. A hook is a mechanism allowing users to customize a framework by tapping into and modifying its inner workings. Customization can be done by composing and subclassing existing classes

and/or by defining implementations of abstract operations. Object-oriented frameworks can be classified into *white-box frameworks* and *black-box frameworks*. The main difference between them depends on what techniques are used to design hot spots: inheritance or composition. In white-box frameworks, inheritance is the predominant technique used to implement the hooks. In black-box frameworks, vice versa, the predominant technique is composition. The customization of a white-box framework requires to understand the internals of the framework because its behaviour is extended by creating subclasses, taking advantage of inheritance. A black-box framework does not require a deep understanding of the framework's implementation because the behaviour is extended by composing objects together, and delegating behaviour between objects. Delegation is the idea that instead of an object doing something itself, it gives another object the task. A white- box framework can be converted into a black box framework by replacing over-ridden methods by message sends to components (Johnson, Foote, 1988). A framework can be both white-box and black-box at the same time.

Design patterns are often used to refine and optimize the frameworks (Fayad, Schmidt, 1997). Patterns can help to design hot spots, as well as frozen spots or other parts of the framework, where flexible and customizable solutions are required. However patterns help to build parts of frameworks, but do not tell how to build the whole framework, (Kaisler, 2005). In addition, a pattern gives only the design idea and an exact solution still has to be implemented to fit the context of the framework. On the other hand, frameworks do not embody patterns, just solutions implicated by patterns. The refinement process of the framework is necessary to apply a certain pattern correctly. It should also be noted that new design patterns are often discovered namely by designing the frameworks. In summary, there are two important distinctions between patterns and frameworks. Firstly, frameworks are complete reusable implementations whereas patterns are design abstractions. As for the second, patterns are considerably smaller structures than frameworks (Johnson, 1997).

The present thesis is dealing with the category of aspect-oriented domain frameworks, namely, white-box frameworks. Aspect-oriented domain framework is a framework that alongside with traditional object-oriented mechanisms provides abstract aspects as hot spots. Such hot spots are inherited by concrete aspects. The applications produced using aspect-oriented domain frameworks consist of a collection of interacting objects, which are weaved with aspects provided by the framework. Aspect-oriented domain frameworks should not be confused with aspect-oriented supporting frameworks (e.g. JBoss AOP (Fleury, Reverbel, 2003) or Spring AOP (Laddad, 2010)) which provide the means to implement crosscutting concerns and/or programming constructs used to specify the crosscutting behaviour of a program.

Chapter 2

State of the Art

The chapter presents the critical analysis of the related works on the aspectization of OO design patterns, paradigm-specific AO design patterns, compositional properties of design patterns, and the design of AO frameworks. **Section 1** analyzes separation of concerns in the context of AOP, **section 2** – proposed methods to transform OO design patterns into AO design patterns, **section 3** – methods to design the compositions of patterns and compositional properties of patterns, **section 4** – proposed paradigm-specific AO design patterns, and, finally, **section 5** analyzes the current experiences in the design of AO frameworks.

2.1. Separation of concerns and AOP

Specifications, design and implementations of software systems in the OO paradigm suffer from tangling and scattering of concerns. Deficiencies of OO design patterns and their actual implementations have been observed in (Cacho et al., 2005; Hannemann, Kiczales, 2002; Piveta, Zancanella, 2003) and others. AOP that originates from the work (Kiczales et al., 1997) attempts to solve the problem of tangling and scattering of concerns by concern separation. The first programming language, which was labelled as the "aspect-oriented" one, was AspectJ (Kiczales et al., 2001). The most important goal of aspect-oriented programming languages is to localize crosscutting of concerns. However, as suggested by Robert E. Filman and Daniel P. Friedman, AOP can be thought in a more general sense:

> *"AOP can be understood as the desires to make quantified statements about the behaviour of programs and to have these quantifications hold over programs written by oblivious programmers."* (Filman, Friedman, 2001)

Many techniques for separating individual concerns were developed long before the AspectJ (Lopes, 2005). Later, techniques not related to one particular concern were suggested: composition filters (Aksit, 1992), meta-level-programming (Kiczales et al., 1991), adaptive programming (Lieberherr et al., 1994), subject-oriented programming (Harrison, Ossher, 1993), etc. Although all these techniques have been proposed for separation of concerns, they are different in their nature and, according to Meslati (Meslati, 2009), at least the concepts of composition filters approach cannot be directly mapped to concepts of AOP. Design patterns have also been introduced as a way to achieve a better separation of concerns. AOP can be implemented in many different (not necessarily object-oriented) ways, including rule-based systems, event-based systems (Filman, Friedman, 2001), intentional programming, meta-programming, generative programming (Czarnecki, Eisenecker, 2000) and others. All these approaches provide some means to express and to implement quantified statements. However, they are still different by the implementation issues they address. For example, the rule-based systems allow a direct implementation of quantified statements while meta-programming lets programmer to manipulate the fragments of a program code in a base language using meta-level language elements. Nevertheless, only the AO languages introduce special concepts used to describe such quantifications. Already the AspectJ has unified a wide spectrum of concern separation ideas using relatively few and simple concepts as well as in a more attractive way than the previous approaches (Lopes, 2005). The new constructs introduced by AO languages (concerns, aspects of concerns, pointcuts and advices, intertype declarations, etc.) allow a programmer to capture the tangled and scattered concern parts and to keep them in separate localized aspects (Laddad, 2003; Czarnecki, Eisenecker, 2000). They extend traditional software engineering paradigms and allow implementing a new kind of architectural patterns. The main idea is to specify, analyze and implement a software system as a collection of separate concerns. To this end, many paradigm-independent as well as paradigm-dependent design problems must be solved. Appropriate

design patterns are required to solve these problems. Although aspects have grown up from OOP, they are also used today together with other paradigms. For example it is possible to speak about aspects in functional programming languages (Dantas et al., 2008) or even in logic programming languages (Filman, Friedman, 2001). It means that the AO paradigm is not an independent one like the OO paradigm, but a paradigm that is built by "aspectization" of some other paradigms that remains beyond it. However, in present thesis only the case where the aspect-oriented paradigm is built over the object-oriented paradigm is considered. In this case, the problem of aspectization of OO design patterns arises. The term "aspectization" addresses the redefinition of OO design patterns in terms of AO paradigm.

2.2. Aspectization of Object-Oriented Design Patterns

In the object-oriented programming, each design pattern defines a parameterized collaboration of objects or, more exactly, a parameterized "*relationships between classes and objects with defined responsibilities that act in concert to carry out the solution*" (Maioriello, 2002). The OO design patterns have already been used for some time and became even "*part of the cutting edge of object-oriented technology*" (Shalloway, Trott, 2001). Many such patterns, for example, the Visitor, Decorator, and Observer, are already well researched and the techniques of their application are elaborated in detail. AOP has grown out directly from OOP, but, together with objects, it provides a new kind of entities, namely, aspects. Due to this fact, a number of new pattern related research questions arise: Does an AO design pattern define a parameterized collaboration of aspects and what means the term "collaboration of aspects"? What techniques can be used to develop AO design patterns and what advantages and shortcomings has each of these techniques got? Does the aspect-oriented paradigm generate some new patterns that are specific only to this paradigm? How mature is the AO software engineering paradigm currently?

The problem of the aspect-oriented implementation of OO design patterns is one far from being simple. A detailed analysis should be made in order to understand the implementation of which OO design patterns are affected by the usage of aspects and how. Afterwards, each design pattern must be redefined from the perspective of AO paradigm. The compositional properties of patterns also should be investigated. When implementing several design patterns in a system, they "*crosscut each other in multiple heterogeneous ways so that their separation and composition are far from being trivial*" (Cacho et al., 2005).

The aspectization of OO design patterns have been investigated by (Hirschfeld et al., 2003; Hachani, Bardou, 2002; Noda, Kishi, 2001; Nordberg, 2001; Nordberg, 2001a; Arnout, Meyer, 2006; Bernardi, DiLucca, 2005; Piveta, Zancanella, 2003;Cunha et al., 2006).

Generally, the aspectization of OO design patterns has been performed in two different ways:

- Implementation of the OO design pattern in some OO language, for example in Java, is directly replaced by the analogous code written in some AO language, for example, in AspectJ (Hannemann, Kiczales 2002);
- A native AO solution is introduced to the same problems that are addressed by the OO design pattern (Hachani, Bardou, 2002; Hirschfeld et al., 2003).

The effectiveness of AOP usually is evaluated by comparing system implementations as well as the process of their development with and without AOP (Papapetrou, Papadopoulos, 2004). A number of metrics have been proposed to measure the effectiveness of the implementation of OO design patterns in AO languages. Hannemann and Kichales (Hannemann, Kiczales 2002) propose to use metrics suite <code locality, reusability, composability, (un)plug ability> and demonstrate that applying this suite even for 17 out of 23 GoF patterns the implementation was improved by rewriting from Java to AspectJ. The quantitative assessment of Java and AspectJ implementations for the 23 GoF patterns has also been done in (Garcia et al., 2005) and (N. Cacho,

et al., 2005). To this end, the authors use the metrics suite <separation of concerns (SoC), coupling, cohesion, code size > defined in (Sant'Anna, et al. 2003; Garcia 2004). Garcia et al. demonstrate that aspect-oriented implementations of most of the 23 GoF patterns improve these patterns regarding the SoC. However, taking into account the whole suite of metrics, the implementations of only 4 patterns exhibit significant improvements. Thus, despite the fact that many patterns like Observer, Visitor, Adapter, Composite and Decorator are confirmed to be better when implemented in AO languages, there are patterns that have less improvements or can become even more complicated.

A number of researchers (Lorenz, 1998; Noda, Kishi, 2001; Hachani, Bardou, 2002; Hachani, Bardou, 2003; Schmidmeier et al., 2003) investigated the benefits of implementing GoF patterns in AspectJ by direct rewriting their implementation from Java to AspectJ. The research in (Hachani, Bardou, 2002; Hachani, Bardou, 2003) focuses on the confusion, indirection, encapsulation breaching, and inheritance related problems raised by the use of OO design patterns. These problems are mainly induced by code scattering and code tangling. So, it is reasonable to expect that implementations in AO languages at least partly will solve these problems. The research has demonstrated that, for most of GoF patterns, such implementations indeed improve a separate reuse of both the pattern and the main application code and solve the confusion, indirection, and encapsulation breaching problems. Inheritance related problems have also been solved for some patterns and lowered for others. Similar results were also obtained in (Hirschfeld et al., 2003).

It is likely that the idea of direct rewriting from one language to another has arisen because some researchers assumed that any design pattern in both paradigms should be implemented in analogous ways. However, as stated in (Vaira, Čaplinskas, 2009), the rewriting of particular cases from one programming language to another can be considered only as samples, but not as the general solution how the design patterns should be redefined for AOP. In addition, the idea behind the pattern usually can be implemented in several

different ways, and it is not so simple to say that the solution obtained by rewriting is really the best one.

So, it seems that a better way to implement design patterns, which solve paradigm-independent (to respect of OO and AO paradigms) design problems, is not to emulate OO patterns but to express the idea behind the pattern directly in AOP terms. Despite the fact that such a way is more difficult than the replacing of OO implementations by the analogous code written in some AO language, it allows us to implement the patterns in more effective way. Particularly, (Noble et al., 2007) demonstrated that using the native approach a number of design patterns (Spectator, Regulator, Patch, Extension, Heterarchical design) can be implemented in the AOP in a simple and elegant way. Although these patterns do not belong to the GoF patterns, they describe a set of solutions to a family of similar design problems. In fact, most of them should be considered as degenerate collaborations because they, like the Singleton pattern, include only one role. Besides, it is questionable if the Heterarchical design pattern is really a design pattern at all. It is rather an architectural pattern. Nevertheless, the research carried out by Nobel et al. demonstrates that the native approach is really promising.

The native approach for the implementation some of the GoF patterns (Template method, Creational patterns, Factory method) has also been used in (Hanenberg, Schmidmeier, 2003). Inter alia, (Hanenberg et al., 2003) has demonstrated that Container Introduction pattern, which is difficult to implement in OOP, can be elegantly implemented in the AOP.

Both aspectization approaches – code rewriting and native – consider AO design patterns as patterns that describe the interactions among objects and aspects. In other words, rewriting of patterns as well as the native approach both aim to improve the implementation of mixed objects and aspects collaborations. The question of how to apply the patterns to design the collaborations of aspects still remains open. Open remain also following questions: Is it possible to implement at least some of GoF design patterns using aspect-oriented constructs only? Which and how, if it is possible? Is such

implementation in some way better than the object-oriented one? How to measure this? The current thesis attempts to answer these questions.

2.3. Compositional Properties of Aspect-Oriented Design Patterns

2.3.1. Analysis of the related works

Both aspectization approaches that were analyzed in the previous section deal only with single design patterns. They do not take into account how separation of concerns in one pattern will affect compositions of several design patterns. The compositional properties of aspect-oriented implementation of OO design patterns obtained by direct rewriting of OO code in some AO language have been investigated in (Cacho et al., 2005; Denier et al., 2005; Denier, Cointe, 2006).

In (Cacho, et al., 2005) the results of an empirical study that investigates whether aspect-oriented implementations improves composability of design patterns in the context of medium-size software systems are presented. Since in such context the design patterns are composed many times and in different manner crosscutting each other in multiple heterogeneous ways, it is natural to expect that aspectization of patterns can significantly improve the implementations of such compositions. However, the study has showed that the results depend on the patterns involved, composition intricacies, and other particular circumstances. In the research, 62 pair-wise compositions of OO design patterns were investigated. Additionally some compositions involving more than two patterns have also been investigated. All investigated compositions have been divided into 4 categories:

- Invocation-based composition when the implementations of the two composed patterns are disjoint and they have no class in common. The roles of patterns are only connected through one or more method calls. This is the simplest form of pattern composition.
- Class-level interlacing when the implementations of two patterns have one or more classes in common. The roles of patterns are implemented

by different sets of methods and attributes in these shared classes. It means that the involved patterns have coinciding participant classes, but there is no common method or attribute implementing roles of both patterns. Consequently, the pattern implementations are interlaced (or tangled) at the class level.

- Method-level interlacing when the implementations of two patterns have one or more methods in common. Different pieces of code in these methods are dedicated to implement roles of both patterns. It means that the pattern implementations are interlaced at the method level.
- Overlapping when the implementations of two patterns share one or more statements, attributes, methods, and classes. This combination style is different from method-level interlacing because here the shared elements are entirely part of roles in both patterns;

In call based compositions, the design patterns are related by some dependencies between the classes of different design patterns. In class overlapping compositions, design patterns overlap by using the same classes in different roles. Method overlapping compositions are similar to the class overlapping compositions. The only difference is that patterns overlap using the same methods. In the completely overlapping compositions, either design patterns overlap by using several common methods or several classes, or one design pattern is part of the other. The compositions of OO design patterns have been transformed into aspect-oriented ones using transformation proposed by (Hannemann, Kiczales, 2002). The evaluation of AO compositions implemented in AspectJ language has been performed using such metrics suite <tangling, cohesion, size, SoC> (Sant'Anna, et al. 2003; Garcia 2004). Both OO and AO compositions have been evaluated and the results were compared. The research demonstrated that "*the aspectization results depend on the patterns involved, the composition intricacies, and the application requirements. In some situations, the aspectization of the pattern composition is not straightforward and several design options need to be considered. Sometimes, it requires a global reasoning in order to understand that impact of*

each design option in the context of the whole system implementation" (Cacho, et al., 2005). The aspectization of some specific compositions with strong coupling between the patterns can bring modularity problems. In some cases, the aspectization of a given design pattern in complex compositions can be not a good design option taking into account the application requirements because it reduces the performance of the whole application program. In summary, this research investigates relatively huge number of pattern compositions and different composition categories but does not cover all the composition possibilities. For example, it does not investigate method overlapping.

Denier et al (Denier et al., 2005; Denier, Cointe, 2006) investigated some cases of composition in the context of JHotDraw framework. This research shows that there is a need for configuration of composition, which involves aspect ordering as well as pointcut transformation. For example, the presence of Composite or Decorator patterns in the base program can have an impact on the Observer pattern pointcuts. The research investigated various types of compositions, from aspectized compositions to compositions of aspects, however the results of the research are insufficient to discover general tendencies.

Thus, (Cacho et al., 2005; Denier et al., 2005; Denier, Cointe, 2006) analyzed a number of compositional properties of aspectized by rewriting GoF patterns including concern separation degree in particular design patterns as well as in their compositions. However, no one investigated the question, whether it is possible to separate all concerns in the whole system. No one investigate also how compositions of design patterns changes involved patterns. However, these properties are also very important compositional properties, especially, in the context of AO frameworks. For this reason, an experimental research described in next subsection has been performed.

2.3.2.Experimental investigation of Separation of Concerns in the Aspectized Design Pattern Application

The object-oriented framework SimJ has been used as a test-bed to investigate separation of concerns in the compositions of aspectized by rewriting GoF

design patterns described in (Hannemann, Kiczales, 2002). SimJ framework is intended to be used to build applications for simulations in different application domains. The supermarket simulator application implementation has been used in this research. The SimJ framework has been chosen because its design is strongly based on design patterns. It is also important that design patterns form compositions in this framework. The following GoF design patterns have been used in the framework: Singleton, Adapter, Façade, Factory Method, Flyweight, Iterator, State and Template (Gamma et al., 1994). Since design patterns tend to overlap and sometimes it is even hard to identify the particular design pattern correctly, not all design patterns that have been used in the framework were analyzed.

Only the separation of logging concern has been investigated in the research. The framework has been aspectized in two different ways – by replacing OO design patterns with rewritten design patterns in AspectJ and by reprogramming corresponding SimJ modules in AspectJ without application of any design patterns – which results have been compared afterwards. Not all design patterns was rewritten from Java to AspectJ directly. Some implementations were modified in order to adapt them to the compositions. In one case the aspectization of Singleton design pattern failed because, in this case, the singleton was parameterised, whereas the implementation of the Singleton proposed by (Hannemann, Kiczales, 2002) does not allow using parameters in the constructor.

The results of the experiment are summarized in the Table 1. Full separation of the logging concern succeeded only by rewriting the modules Table 1. However, such aspectization affects all application programs developed using this framework. This means that although rewriting of design patterns should be performed in the frameworks in order to avoid changes in application programs, the aspectization of design patterns by rewriting may not be sufficient for full separation of concerns. It also follows that the design pattern aspectization approach proposed by (Hannemann, Kiczales, 2002) in some cases may reduce the universality of GoF design patterns. The proposed

approach is local and does not take into account the interaction of the patterns in the whole system.

Table 1 Results obtained using two different implementations

Method / Results	**Rewriting design patterns**	**Rewriting modules**
Separation of logging concern	Partially separated	Fully separated
Units required to implement logging concern	1 aspect	2 aspects, 2 classes
Patterns used to implement the concern	4 Adapter, 1 Singleton	
Affected patterns	4 Adapter, 4 Singleton, 1 Flyweight	
Impact on the application programs	Absent	All application programs must be changed

2.4. Paradigm-Specific Aspect-Oriented Design Patterns

Apart from the design patterns for the design of objects and classes, in the AO paradigm are also required pure AO design patterns, that is, design patterns for the design of aspects themselves. The problem of the development of such patterns is even more complicated than the problem of aspect-oriented implementation of OO patterns. This problem has been investigated by (Lorenz, 1998; Noble, 2007; Bynens, Joosen, 2009; Hanenberg, Costanza, 2002; Hanenberg, Schmidmeier, 2003; Laddad, 2003; Schmidmeier, 2004; Miles, 2004; Griswold et al., 2006; Lagaisse, Joosen, 2006; Bynens et al., 2007; Menkyna et al., 2010). However the research is still at its early phase, mostly, based on the occasional experience gained from developing of the industrial software systems and did not answer a number of important

questions: *In which different ways can design patterns be used to solve OO and AO design problems? In which way are the aspects different as classes from the viewpoint of design patterns and what is the impact of such differences on the structure and other properties of pure AO design patterns?*

The work (Hanenberg, Costanza, 2002) was one of the first on AO-specific design patterns. In this paper, a number of so called AO strategies have been proposed. However, the authors had different opinions on how these strategies should be treated. According to Hanenberg,

> *"… these strategies are no patterns. The main purpose of identifying these strategies was to find out what language features of AspectJ are usually used in what situations. …The strategies have directly arisen from the usage of AspectJ, so they are the result of observing AspectJ code."* (Hanenberg, Costanza, 2002)

Hanenberg suggested that at the time it was impossible to develop some AOP – specific patterns because the aspect-oriented community has still not developed any common understanding of aspect-oriented programming or had any commonly accepted design notation. According to Costanza, the proposed strategies are the first steps towards AO-specific design patterns and even should be regarded only as some proto-patterns but not the patterns themselves. Hanenberg and Schmidmeier (Hanenberg, Schmidmeier, 2003) were going one step ahead. They investigated not only the implementations of some GoF patterns using the native approach, but proposed also the so-called Pointcut Method pattern, which "*is used, whenever a certain advice is needed whose execution depends on runtime specific elements*" (Hanenberg, Costanza, 2002). The authors themselves considered Pointcut Method as an AspectJ idiom, but not as a design pattern, and did not present it in the pattern format. They wrote:

> *"…we still neglect to put the idioms in such a format because of two reasons. First, we feel that it is still more important to discuss typical design decisions in aspect-oriented languages than to claim that a number of good patterns are found. And second, it is still not yet clearly determined what language features an aspect-oriented*

> *language will provide in the future: the provided language features still evolve from version to version. Hence, a collection of good design decisions might be no longer valid in the future because of language changes in AspectJ."* (Hanenberg, Schmidmeier, 2003)

Nevertheless, the Pointcut Method is expressed in a language independent AOP vocabulary and should be considered rather as AO-specific design pattern than as an idiom of AspectJ. According to the classification proposed by (Menkyna et al., 2010), it belongs to the advice category. (Menkyna et al., 2010) suggested that the prevailing part of AO-specific design patterns can be divided into: pointcut patterns, advice patterns, and intertype declaration patterns.

Up to date, a number of AO-specific design patterns have been proposed also by other authors. Some of them are:

- **Wormhole**: "*transport context information throughout a method call chain without the need for parameters*" (Laddad, 2003; Bynens, Joosen, 2009; belongs to the category of the pointcut patterns);
- **Participant**: "*connect an abstract pointcut for each subsystem separately and within that subsystem*" (Laddad, 2003; Bynens, Joosen, 2009; belongs to the category of the pointcut patterns);
- **Director** (Default Interface Implementation): "*abstract aspect with multiple roles as nested interfaces*" (Miles, 2004; Bynens, Joosen, 2009; Menkyna et al., 2010; belongs to the category of the inter-type declaration patterns);
- **Border Control**: "*set of pointcuts that delimit certain regions in the base application*" (Miles, 2004; Bynens, Joosen, 2009; belongs to the category of the pointcut patterns);
- **Cuckoo's Egg**: "*put another object instead of the one that the creator expected to receive*" (Miles, 2004; Menkyna et al., 2010; belongs to the category of the advice patterns);

- **Worker Object Creation**: "*captures the original method execution into a runnable object*" (Laddad, 2003; Schmidmeier, 2004; Menkyna et al., 2010; belongs to the category of the advice patterns);
- **Exception introduction**: "*solves the problem of the exception handling in the advice, by catching a checked exception and wrapping it into a new concern-specific runtime exceptions*" (Laddad, 2003; Menkyna et al., 2010; belongs to the category of the advice patterns); and
- **Policy**: "*defines some policy or rules within the application. A breaking of such a rule or policy involves issuing a compiler warning or error*" (Miles, 2004; Menkyna et al., 2010; belongs to the category of the inter-type declaration patterns).

In summary, any of the above presented patterns are elaborated in detail. Mostly they define individual roles but not collaborations and, consequently, still should be regarded as fragments of patterns rather than real design patterns. Nevertheless they should be considered as a valuable contribution to the field because they are significant milestones towards the AO design pattern development and probably will stimulate the development of more complex aspect-oriented design structures. However, there is "*still a lot of work*" (Bynens, Joosen, 2009) to be done.

2.5. Aspect-Oriented Framework Design

The application of aspects in the design of various kinds of frameworks has been investigated by several authors (Rausch et al., 2003; Arpaia, et al, 2008; Santos et al., 2007; Kulesza et al., 2006). However, in most cases aspects have been used to design only frozen spots (i.e. unchangeable parts of framework) or, in the best-case, as supporting means to design OO hot spots but not as an implementation mechanism of AO hot spots. For example, Rausch et al. (Rausch et al., 2003) used aspects as a glue code for gluing framework core and the produced applications. They performed the case study, in which a small application and a persistence framework were glued by the special program in AspectJ. They proposed also how to model aspect-oriented gluing

in UML. Authors stated that they have modelled framework's hot spots as aspects, but indeed they used aspects to glue object-oriented hot spots with an AO application program. In order to develop an application using such framework, one still need to develop classes with the intention that they will implement operations of some abstract framework classes designed as inheritance-based hot spots and other classes that will be calling operations of framework interface as composition-based hot spots. Only the necessary inheritance declarations would be implemented in aspects using intertype declarations and compositions of classes would be defined by aspects using pointcuts and advice.

A more exhaustive and a more related to the research issues of present thesis are the works (Arpaia, et al, 2008; Santos et al., 2007). They investigated the design of AO frameworks that implement hot spots using customizable aspects. (Arpaia, et al, 2008) have developed AO framework for applications required to control the measurement station that tests superconducting magnets at the European Organization for Nuclear Research (CERN). They have designed an abstract synchronization aspect that provides reusable code and behaviour required to design necessary synchronization logic and policies. Concrete aspects are used for customizing synchronization behaviour and are particularly responsible for intercepting "*components and services interactions that need to be synchronized and enforce the right synchronization policy in the right context*" (Arpaia, et al, 2008). In (Santos et al., 2007) abstract aspects are used in a framework to encapsulate into single module hot spots supporting some framework feature. Authors refer to abstract aspects as specialization aspects. They propose to express hot spots by specialization aspects and to implement the applications by extending specialization aspects with concrete aspects. However, they do not discuss how the AO design patterns can be applied for this aim.

More complex design structures that involve some idioms of AspectJ were suggested in (Kulesza et al., 2006). Authors propose how to use extension join

points to design hot spots. However, they do also not discuss the application of AO design patterns.

As discussed above, a number of AO design patterns have been proposed by (Hanenberg et al., 2003; Laddad, 2003; Miles, 2004; Bynens, Joosen, 2009). Some of these patterns have been successfully applied in the case studies described in Chapter 5. It is likely that other patterns can also be successfully applied to design AO frameworks. However, no report has been published up to date about the application these patterns in the framework design.

2.6. Summary

In this chapter the approaches used to develop the aspect-oriented design patterns has been analyzed. The proposed design pattern transformation techniques were discussed and compositional properties of AO design patterns have been examined. The known AO-specific design patterns have been discussed. Some initial attempts to design AO framework have been considered and it was shown how abstract aspects and idioms of AspectJ have been used to implement the hot spots. The main conclusions of the chapter are as follows:

1. There are many techniques that deal with crosscutting of concerns, though only AOP provides specific programming constructs to deal with it.
2. The novelty of AO programming constructs and its dependence on the base software engineering paradigm requires new kind of architectural and design patterns.
3. Despite the fact that many aspectized design patterns are confirmed to be better when implemented in AO languages, there are patterns that have fewer improvements or can become even more complicated.
4. Rewriting of particular cases from one programming language to another can be considered only as samples, but not as the general solution how the design patterns should be redefined for AO paradigm.

5. Current design pattern aspectization approaches aim to improve the implementations of mixed objects and aspects collaborations, not to design the collaborations of aspects.
6. Current aspectized design patterns are not sufficient for complete separation of concerns. The implementations of these design patterns are not enough universal and can be applied to a very specific application context. It is necessary to analyze more general implementations of design patterns and possible application contexts of such design patterns.
7. The aspectization of design patterns by rewriting proposed by (Hannemann, Kiczales, 2002) may not be sufficient for full separation of concerns when rewriting compositions of patterns and in some cases it may reduce the universality of GoF design patterns. The proposed approach is local and does not take into account the interaction of the patterns in the whole system.
8. AO-specific design patterns still should be regarded as fragments of patterns rather than real design patterns. Nevertheless they provide a valuable contribution to the field because they are significant milestones towards the AO design pattern development and probably will stimulate the development of more complex aspect-oriented design structures.
9. Nobody has considered the relation between the transformation of design patterns from one paradigm to another and the character – paradigm-independent or paradigm-specific – of the problems which intent to solve these patterns.
10. It is still unknown how to transform OO design patterns into pure AO design patterns.
11. Although aspects have been used as an instrument to design unchangeable parts of the frameworks, as supporting means to design OO hot spots in the frameworks and in several cases even as a customizable aspects that implement hot spots in the frameworks,

nobody analyzes how the AO design patterns can be applied for this aim.

12. It is still unknown how to use pure AO design patterns to design hot spots for AO frameworks and, consequently, current AO frameworks still do not use the aspect-oriented paradigm in its full extent.

The results of this chapter have been published in (Vaira, Čaplinskas, 2011; Vaira, Čaplinskas, 2011a; Vaira, Čaplinskas, 2009; Vaira, 2009).

Chapter 3

Development of the methods and procedures for transformation of GoF design patterns into pure AO design patterns

This chapter presents main theoretical results of the doctoral research. **Section 1** proposes the classification of object-oriented and aspect-oriented design problem solutions. **Section 3** describes technique developed for rewriting paradigm-independent GoF design patterns in terms of aspects. **Section 4** demonstrates detailed analysis of applicability of the technique to four representative design patterns. **Section 5** provides short descriptions of the remaining sixteen transformed design patterns.

3.1. Classification of Object-Oriented and Aspect-Oriented Design Problem Solutions

The initial number of GoF design patterns has already required some categorization in order to organize them properly. The most widely known two-dimensional pattern categorization is provided in design patterns catalogue (Gamma, et al., 1994), where they are classified by the purpose and scope of the particular pattern. Further, the increase of design patterns in numbers and the necessity to analyze pattern collections from different perspectives stimulated the appearance of other design pattern classifications, such as Zimmer (Zimmer 1995) and Buschmann (Buschmann, et al., 1996) classifications. In order to simplify the understanding of the overall structure of the Gamma pattern catalogue and to ease the classification of other design patterns Zimmer proposes the classification of relationships between the pairs

of design patterns (Zimmer 1995). According to Zimmer, using this classification design patterns can be arranged into 3 different layers: basic design patterns and techniques, design patterns for typical software problems and design patterns specific to an application domain. Buschmann classifies design patterns by their granularity and purpose (i.e. two-dimensional classification). Although the purpose criterion remains the same as in Gamma catalogue, Buschmann provides wider range of design pattern purposes. Granularity allows classifying patterns by the level of abstraction of a particular design pattern. It brings additional architectural and coding patterns next to design patters. Design patterns are classified into 3 different groups by granularity: architectural patterns, design patterns and idioms.

All these classifications are well suited to classify design patterns of OO software design. However, in the present thesis AO paradigm design patterns are analyzed as well. There exist design patterns also in other paradigms, such as functional paradigm (Lämmel, Visser, 2002). Design patterns can be easily categorized by the particular paradigm which they belong to. AO design patterns according to (Menkyna et al., 2010) can be categorized by taking into account different AO design mechanisms to which design patterns are focused: pointcut patterns, advice patterns, and intertype declaration patterns. Nevertheless, there is no categorization or classification which allows classifying cross-paradigm design patterns, such as aspectized design patterns proposed by (Hannemann, Kiczales, 2002). It is also necessary to distinguish paradigm independency and paradigm specificity of the design problems being solved by the patterns. It must be considered that AO programs are built over the OO base program and may result in a variety of mixed design structures.

In order to satisfy the above named requirements necessary to arrange design patterns of both OO and AO paradigms the following classification of the ways of solving OO and AO design problems using design patterns has been considered (Table 2)[1].

[1] Such problems that are solved by the composition of several patterns are not considered.

Table 2 The classification of OO and AO design problem solutions

Solutions / Problems	**OO solution**	**AO solution**	**Mixed AO and OO solution**
Paradigm independent problems (e.g. communication of the entities with different interfaces; solved by the Adapter pattern)	Use a pattern composed of pattern-oriented objects only (Gamma et al., 1994)	Use a pattern composed of pattern-oriented aspects only	Use a pattern composed of pattern-oriented aspects and objects (Hannemann, Kiczales, 2002)
OO specific problems (e.g. making clones of an existing object; solved by the Prototype pattern)	Use a pattern composed of pattern-oriented objects only (Gamma et al., 1994)	Use a pattern composed of pattern-oriented aspects that are bonded with base OO program (Laddad, 2003, Miles, 2004)	Use a pattern composed of pattern-oriented aspects that are bonded with the base OO program, and pattern-oriented objects (Hannemann, Kiczales, 2002, Laddad, 2003, Miles, 2004; Hanenberg, Unland, 2003)
AO specific problems (e.g. invoking a chain of advices when a pointcut matches; solved by the Chained Advice pattern)	Use a pattern that is implemented by an aspect-aware base OO program (Griswold, et al., 2006; Bynens, Joosen, 2009)	Use a pattern composed of pattern-oriented aspects only (Miles, 2004, Hanenberg; Unland, 2003; Bynens et al., 2007)	Use a pattern composed of pattern-oriented aspects and an aspect-aware base OO program (Laddad, 2003; Hanenberg, Unland, 2003)

This classification can be illustrated by a graphical diagram. Fig. 3 shows two crosscutting concerns. The boundaries of concerns are represented by a straight line[2]. Applications of patterns in the program are represented by large ovals, aspects – by small stroked ovals. Dashed ovals represent the application of

[2] In the models of real-world programs, usually, it is impossible to separate concerns by the straight line.

patterns that solve the problems using mixed solutions. Rectangular shapes represent classes. Solid lines between classes and aspects represent associations (including inheritances), dashed lines connect join points in the classes and pointcuts in the aspects, respectively. The connected classes and aspects are filled with upward diagonal patterns. Design patterns that solve OO or AO paradigm-specific problems and are implemented using the constructs of an appropriate paradigm only are placed at the top of the diagram. Design patterns that solve the AO problem and are implemented using OO constructs or, vice versa, design patterns that solve the OO problem and are implemented using AO constructs are placed in the middle of the diagram. Design patterns solving paradigm-independent design problems are placed at the bottom of the diagram.

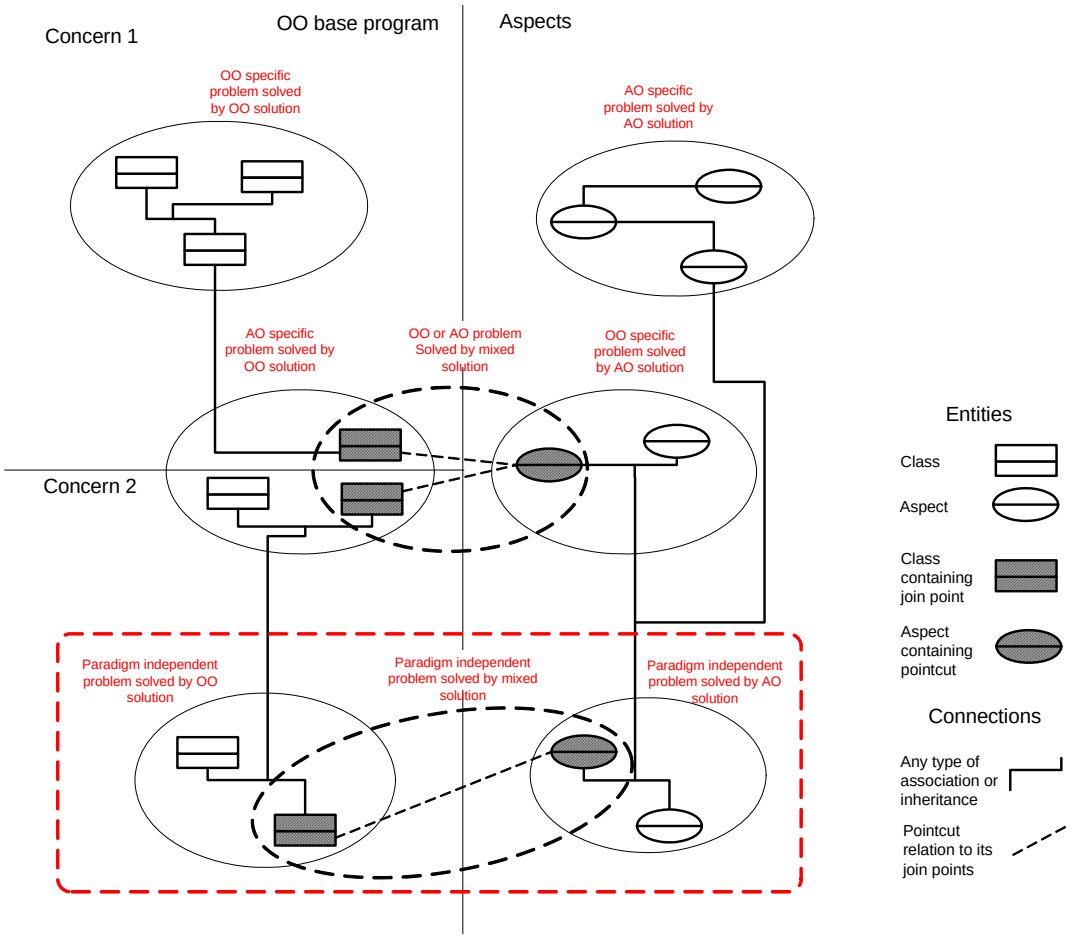


Fig. 3 A graphical diagram illustrating the classification presented in **Table 2** (bigger diagram can be found in APPENDIX C)

The structure of the solution used by such patterns is the same in both AO and OO paradigms, but the elements that constitute the patterns are different. Fig. 3 demonstrates all ways that can be used to solve design problems using different implementations of design patterns. Similarly as in (Bynens, Joosen, 2009), this classification is based on the nature of problems that patterns intend to solve. The following problems are considered: problems that can be formulated in a paradigm-independent way, problems that can be formulated only in terms of the OO paradigm, and problems that can be formulated only in terms of the AO paradigm. It is supposed that a problem of belonging to any of these groups can be solved in three different ways: using only OO mechanisms, using only AO mechanisms, and using both OO and AO mechanisms. Although from the first view it may look a little confusing that specific OO design problems can be solved using pure AO patterns or vice versa, it will be demonstrated later that it is not only possible, but, in some cases, even reasonable.

In the proposed classification, OO specific patterns (e.g. Prototype, Singleton, and Composite) belong to the OO solution column, while AO specific patterns (e.g. Border Control, Abstract Pointcut, Pointcut Method, Template Advice, Chained Advice, Elementary pointcuts, Pointcut Method) – to the AO solution column. Using the AO solution to solve OO specific problems, the pattern is composed of aspects only, but these aspects are bonded with the base OO program. Examples of such patterns are the Wormhole, Worker Object Creation, Cuckoo's Egg and Policy patterns. For example, the Wormhole pattern solves a problem how to pass context information from a caller object to some object deep in the call graph. The traditional OO solution is to add a context parameter to all the intermediate methods that is not needed, but only passed along the object that calls it. The Wormhole pattern proposes a more economic of solution. It provides a pattern-oriented aspect that uses a pointcut to capture the information when it is available, and advice to re-introduce it when it is needed (Laddad, 2003).

Mixed solutions depend on the kind of problem to be solved. For example, all aspectizations of paradigm-independent GoF patterns belong to this class. Such aspectizations are composed of pattern-oriented aspects and objects (Hannemann, Kiczales, 2002,). A mixed solution of OO specific GoF patterns (Prototype, Singleton, and Composite) together with pattern-oriented objects uses pattern-oriented aspects that are bonded with the base OO program (Hannemann, Kiczales, 2002,). The Director (Miles, 2004), Container introduction (Hanenberg, Unland, 2003) and Participant (Laddad, 2003) patterns are implemented in such a way. Finally, the mixed solution of AO specific problems is implemented by a pattern that is composed of pattern-oriented aspects and by an aspect-aware base OO program. The Exception introduction (Laddad, 2003) and Marker interface (Hanenberg, Unland, 2003) patterns belong to this class.

An interesting class is the class of OO solutions that solves specific AO design problems. In this case, any solution is related to naming and annotation conventions in the base program (Griswold, et al., 2006). For example, having aspects with complex and hard to understand pointcut definitions, it is necessary to modify the base program in order to make it more pointcut friendly. To solve that, it is necessary to design appropriate naming and annotation conventions for the base program.

Paradigm-independent design patterns can be used to solve problems that reoccur in the systems implemented using different paradigms. In present thesis, only two paradigms are investigated: AO paradigm and OO paradigm. In addition, it is supposed that aspects are built over the OO base program. In this context, aspects and classes differ in two main points. The first one is the ability of classes to be instantiated, whereas aspects are singletons by their nature. The second point is that an aspect is a collection of pointcuts and advice, whereas a class does not provide such kinds of constructions at all. Thus, most of the researchers sought to combine both paradigms and proposed various mixed AO and OO solutions to solve paradigm-independent design problems. As far as it is known, there are no publications that aim to

investigate the class of pure AO solutions solving such problems. However, (Hanenberg, Unland, 2003) use de facto pure AO implementation of the Template Method pattern in the Template Advice pattern, although they do not state this fact explicitly.

Since the class of pure AO patterns that solve paradigm independent design problems was not investigated at all to date, the remaining part of this chapter is devoted namely to this question. The 23 GoF design patterns are investigated, the fact that only 20 out of this class of patterns solve paradigm-independent design problems is demonstrated and a way how these patterns can be implemented using AOP constructs only is proposed.

3.2. Aspect-Oriented Solutions of Paradigm Independent Design Problems

If some of GoF patterns can be implemented in AspectJ by using AO constructs only, it can be considered as a pattern that, at least to respect of OO and AO paradigms, solves a paradigm-independent design problem. Despite the fact that, in such a case, both OO and AO patterns solve the same design problem, their applicability differs. The OO pattern solves this problem for objects, whereas the AO pattern solves it for aspects. The proposed methodology, to rewrite paradigm-independent 23 GoF design patterns for aspects is briefly considered.

Despite the fact that aspects and classes are different concepts, they have some similarities. AO paradigm language implementations, such as AspectJ, inherit elements of a larger scale base paradigm, on which it is built up. Resulted AspectJ language implementation still includes other small scale paradigm elements that are introduced by AOP (Vranić, 2001). This results in complex structures that can be problematic to be developed. Since crosscutting concerns can have and maintain states, the aspects, similarly as classes, can define data members and behaviours for crosscutting concerns (Laddad, 2003), be abstract, and implement interfaces. It is also possible to built inheritance hierarchies for abstract aspects. However, other than classes, aspects cannot be directly

instantiated. Although it is possible to have several instances of aspects in entire program, only one instance of the aspect can be created for any particular object or control flow in a program related to predefined pointcut. Thus, in the context of the present thesis, they are treated as singletons. Consequently, similar and/or slightly changed structure of OO GoF design patterns can be used to build the AO ones. The only necessary task is to replace OO language constructs by the appropriate AO language – AspectJ in this research – constructs.

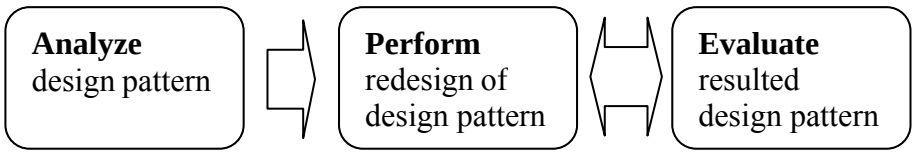


Fig. 4 Redesign technique

It can be done in 3 steps (Fig. 4):

- If a GoF pattern, possibly, with a reduced applicability, can be implemented using only singletons, this pattern is regarded as a candidate to be a paradigm independent pattern for rewriting in AspectJ.
- All the classes in the candidate pattern should be replaced with aspects and all object constructors should be replaced by the AspectJ static method *aspectOf*, which allows accessing the instance of the aspect. A constructor with arguments can be modelled by an appropriate aspect method or often even replaced simply by the assignment of appropriate default values to the data members in the aspect. Data members, behaviours, and inheritance relations in aspects mainly imitate that of the classes. The pointcuts and advices that trigger aspects should be modelled depending on the OO base program. For this reason, in each pattern at least one class as a placeholder for a join point that initiates the pattern is necessary.
- The candidate pattern should be analyzed in order to discover and remove irrelevant data members and methods. Some data members and methods can become irrelevant because the aspects which replaced the classes are singletons and because of transformation of some pattern

members to fit the pointcut model in the pattern. It may happen, that afterwards some design patterns (e.g. Singleton) "disappear", because they become so simple that cannot be regarded further as proper design patterns.

The next section provides some essential examples of the application of this approach, remaining descriptions of transformed design patterns are presented in APPENDIX B.

3.3. Investigation of the Applicability of GoF Patterns to Design the Aspects

If even at first glance, it might appear that pure AO design patterns can be defined by analogy to the OO design patterns, it is not true. However, some OO patterns become trivial for aspects because they are directly supported by AOP. For example, nobody needs the Singleton pattern for aspects because the aspects itself may be used as singletons. Some other patterns are not affected in any way by change of objects to aspects. For example, the Façade pattern is implemented in an analogous way for both, objects and aspects. It also seems that some OO patterns, for example, the Prototype, solve paradigm-dependent design problems and are senseless for aspects.

It is obvious that the GoF patterns – Singleton, Prototype, and Composite – are senseless in the aspect-oriented paradigm. The Singleton pattern becomes trivial after rewriting it in AspectJ and "disappears". The essence of Prototype pattern is the ability of objects to clone its instances (i.e. create several instances of the same class based on already existing instance). However, in AOP no one needs to clone the aspects. Even if it is possible to use several instances of aspects per object or per control flow, it is not possible to control instantiation in the way to support cloning. Thus, Singleton and Prototype design patterns are senseless in AO paradigm. Senseless is also the Composite pattern because, in the case of OO paradigm, its implementation requires to hold the references from one to another instance of Composite object. In the case of AO paradigm, the solution results in an eternal loop when only one

container aspect is defined and this aspect is referenced in a tree at least two times. Despite the fact that, theoretically, it is possible to create the AO implementation, in which the container aspect refers to only one instance of leaf aspect or in which all container instances are defined in a tree as separate aspects, such implementation is purposeless because the context to which it could be applied remains unclear and it is questionable whether this context still corresponds to the Composite design pattern.

The remaining 20 out of 23 GoF patterns can be adapted to solve the aspect design problems. They have been rewritten in AspectJ using only pure AO constructs. However, the AO implementation of 5 design patterns – Chain of Responsibility, Proxy, Interpreter, Memento, and Flyweight – in some way is more constrained than OO implementation because it is impossible to work with several instances of an aspect at the same time. For example, it is impossible to have several instantiation of the same Proxy aspect simultaneously.

Using the above described approach, examples of the AOP implementation of those out of GoF design patterns, which can be adapted to solve the aspect design problems, are considered. Although the implementation of all such patterns has been investigated in details, the 4 representative examples are described (other 16 design patterns are presented in shortened form of description in APPENDIX B): the simple Adapter design pattern, more complex Bridge design pattern, Factory Method design pattern and Chain of Responsibility design pattern. The Factory Method pattern is chosen as an example of creational design pattern. The Chain of Responsibility pattern is chosen as a most representative example for the above mentioned group of the design patterns (Proxy, Interpreter, Memento, Flyweight, and Chain of Responsibility). This pattern includes constraints on references as well as constraints on instantiation of aspects, which manifest itself also in other patterns of this group.

UML class diagrams are used to model both OO and AO patterns. To represent aspects in UML models following stereotypes are used: Aspect, Advice,

Pointcut, and Join point. The latter one represents the relation between the pointcut, described in the aspect, and its actual join points in classes. While modelling the AO patterns by UML, the traditional UML relations such as inheritance, association, and dependency are used. For a better understanding of the diagrams the AspectJ representations of AO design patterns are described.

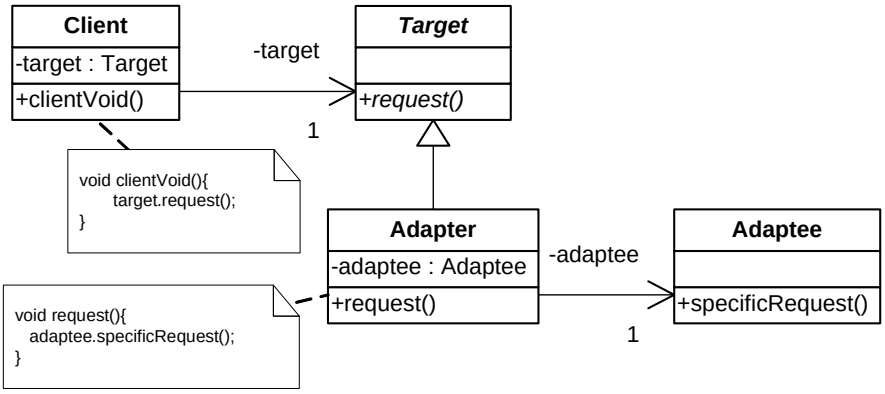


Fig. 5 Adapter design pattern (OO solution)

GoF Adapter design pattern is considered in (Fig. 5). The essential elements of this pattern are:

- *Client*, the class containing *clientVoid* method,
- *Target*, the abstract class containing an abstract request operation,
- *Adapter*, a subclass of the *Target* class that overwrites the request operation with the request method, and
- *Adaptee*, the class containing the *specificRequest* method that is adapted by the *request* method in the *Adapter* class.

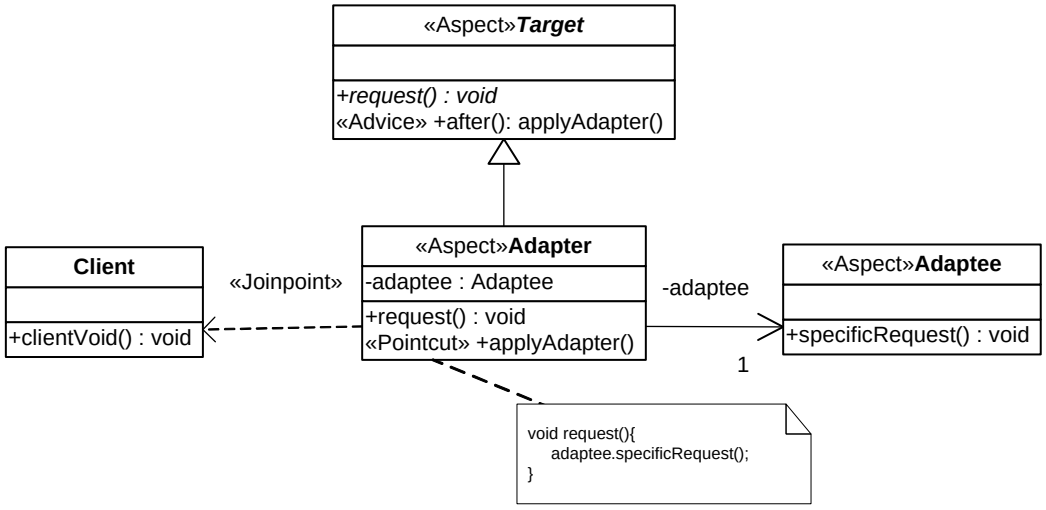


Fig. 6 Adapter design pattern (AO solution)

In order to rewrite the Adapter design pattern for aspects, the proposed transformation technique is applied. In the AO solution (Fig. 6) the classes *Target*, *Adapter* and *Adaptee* are replaced with the aspects *Target*, *Adapter* and *Adaptee*. The class *Client* remains. However, it is not considered as a part of resulted design pattern and serves rather as a placeholder for a join point that triggers the *Adapter* aspect. In other words, the *Client* class is a technical class that should not be regarded as a first order citizen. Therefore, a solution that consists only of aspects is received (Fig. 7).

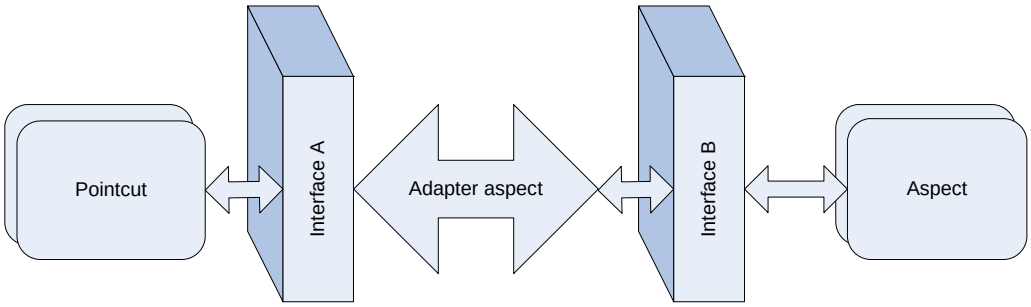


Fig. 7. The idea behind Aspect adapter

Example 2 presents the AspectJ code for this solution. Abstract aspect *Target* contains an abstract operation *request* and an advice body for pointcut *ApplyAdapter*. The aspect *Adaptee* contains the *specificRequest* method that must be adapted by the *Adapter* aspect. The *Adapter* aspect contains the concrete *request* method body and the concrete *applyAdapter* pointcut. The

Adapter aspect uses the *specificRequest* method defined in the *Adaptee* aspect inside the *request* method.

```
1  public abstract aspect Target {
2    void request() ;
3    after(): applyAdapter () {
4      request();
5    }
6  }
7
8  public aspect Adaptee {
9    public void specificRequest() {
10      System.out.printLn("Executing specific request..")
11   }
12 }
13
14 public aspect Adapter extends Target {
15    Adaptee adaptee = Adaptee.aspectof();
16    void request(){
17       System.out.println("Executing inherited request..");
18       adaptee.specificRequest();
19    }
20
21    pointcut applyAdapter()
22    :execution(public static void main())&&target(ClientClass);
23 }
```

Example 2 AspectJ code of the Adapter design pattern

This example demonstrates how to rewrite the Adapter and other simple object-oriented 23 GoF patterns in terms of the AO paradigm or, in other words, it demonstrates that it is possible to apply these patterns to solve aspect design problems. However the question arises as to how useful and for which purposes pure AO patterns are. In order to answer this question, some practical usage of the Adapter AO design pattern is demonstrated below.

The main intent of *Adapter* is to convert the programming interface of one entity into that of another (Fig. 7). In our case, entities are aspects. The complex Logger concern consisting of several aspects is considered (Fig. 8).

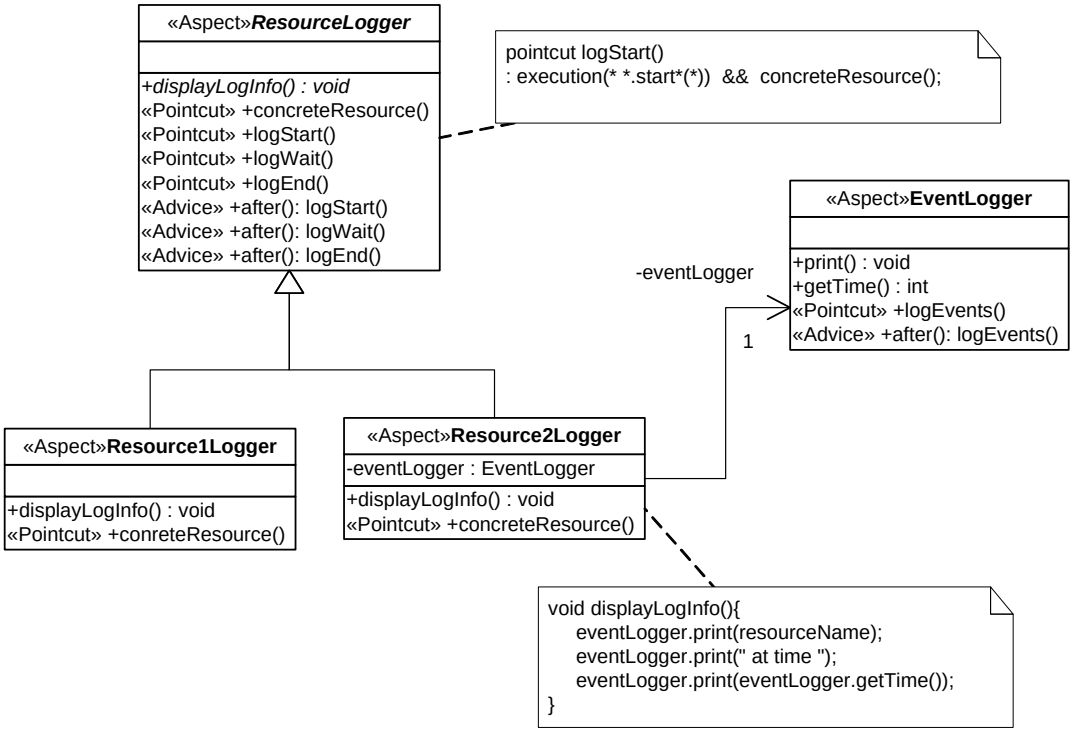


Fig. 8 Application of the AO design pattern Adapter

There are different kinds of things – events and resources – that must be logged by a Logger. Logging of these different kinds of things requires different behaviour. So, it is not reasonable to implement such a Logger as one aspect, because this aspect will have many unrelated pointcuts and a repeating code. To avoid that, different aspects to log each kind of things can be used. Thus, two aspects responsible for logging events and resources have been created (Fig. 8). However, the resources also may be different. For this reason the *ResourceLogger* aspect must be an abstract aspect that could be inherited by concrete resource loggers: *Resource1Logger* and *Resource2Logger*. In *ResourceLogger* there is an abstract operation *displayLogInfo* and an abstract pointcut *concreteResource* that is overridden in concrete resource loggers. The pointcut *concreteResource* is part of all the other pointcuts and helps to specialize them without rewriting each pointcut. In the *Resource2Logger* operations defined in the *EventLogger*, namely, *print* and *getTime*, should be used. In order to adapt these operations to *Resource2Logger*, the Adapter design pattern presented in Fig. 6 has been applied. In a similar way this problem may be also solved using the Template Method design pattern. In this

case, an abstract aspect should be created and the needed methods could be inherited by all the other aspects. However, it is not always desirable for all aspects to inherit these methods (e.g. some of particular loggers do not need to adapt them at all). Thus such a solution is applicable only in some cases.

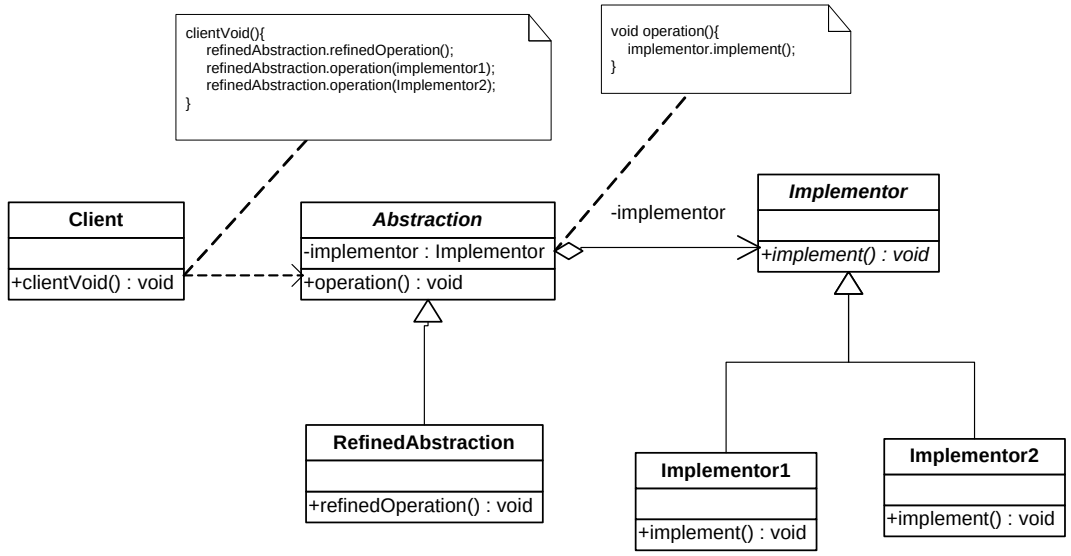


Fig. 9. Bridge design pattern (OO solution)

In order to demonstrate a more complex situation, the GoF *Bridge* pattern is considered (Fig. 9). The main intent of *Bridge* is to separate the abstract elements of a class from the implementation details. The essential elements of this pattern are:

- *Abstraction* defines the interface that the client uses for interaction with this abstraction. It is the only an interface that is known to the client and he makes requests directly to the *Abstraction* object. This object maintains a reference to an *Implementor* object. Through this reference the client's requests are forwarded by the *Abstraction* to the *Implementor*.
- *Implementor* defines the interface for any and all the implementations of the *Abstraction*. The *Abstraction* interface and the *Implementor* interface can differ and this is an additional source of flexibility provided by this pattern. According to Gamma, "*Typically the Implementor interface provides only primitive operations, and*

Abstraction defines higher-level operations based on these primitives." (Gamma et al., 1994)

- *RefinedAbstraction* is any and all the extensions to the *Abstraction* class, and
- Any *ConcreteImplementor* implements the interface defined by the *Implementor* class or, in other words, defines a concrete implementation of the *Abstraction*.

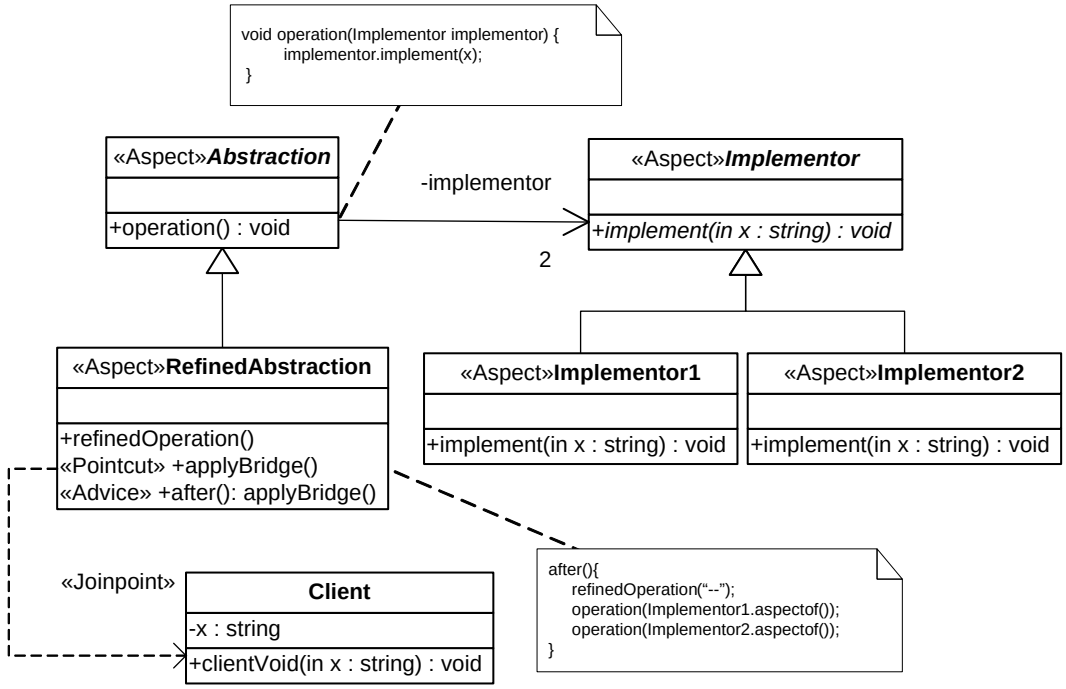


Fig. 10 Bridge Design pattern (AO solution)

Similarly as in the Adapter pattern, in the Bridge pattern (Fig. 10) classes are also replaced by aspects. However, some other changes have been made, too. It is because the situation, when the *Client* class sends request to the *Abstraction* class and asks to execute the abstract operation *operation*, cannot be modelled directly in the AO pattern. In our solution, the abstract operation *operation* of the aspect *Abstraction* is triggered by the pointcut *applyBridge* and the aspect *Abstraction* forwards to the aspect *Implementor* the reference to the required implementor as a parameter of the *AspectOf* method. As a result the solution that consists only of aspects is received (Fig. 11).

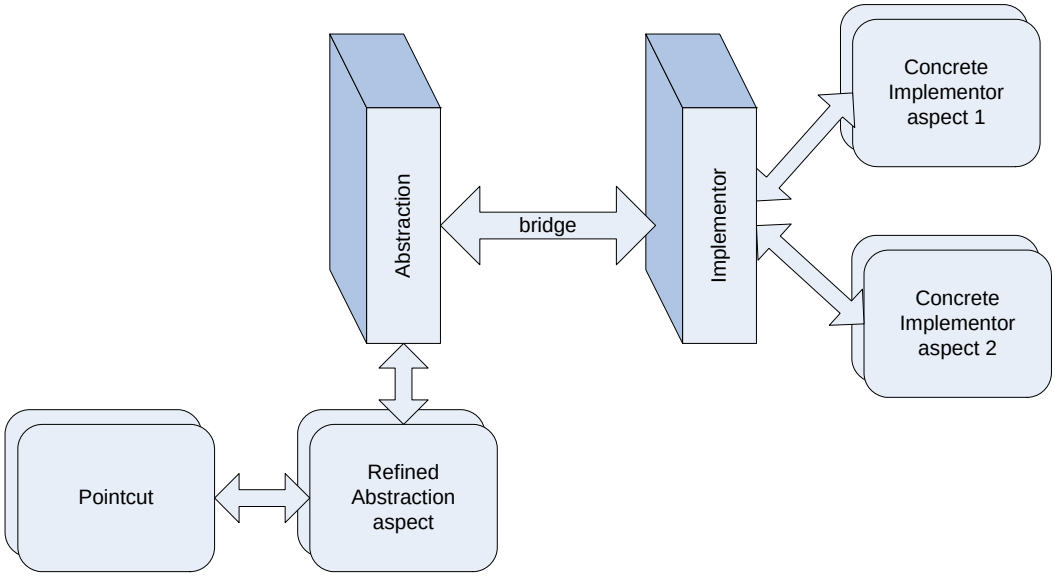


Fig. 11 The idea behind Aspect Bridge

Example 3 presents the AspectJ code for this solution. It is possible to see in this program (lines 12, 13) that the required implementor is invoked in a similar way as in the OO solution.

```
1  public abstract aspect Abstraction {
2       String x;
3
4       public void operation(Implementor implementor) {
5            implementor.implement(x);
6       }
7
8       after(String x): applyBridge(x) {
9
10           this.x = x;
11           operation(Implementor1.aspectof());
12           operation(Implementor2.aspectof());
13      }
14 }
15
16 public aspect RefinedAbstraction extends Abstraction {
17
18      public void operation(Implementor implementor) {
19
20           //refinement
21           x = “--”+x+“--”;
22
23           implementor.implement(x);
24      }
25
26      pointcut applyBridge(String x) :
27           call(public void clientVoid(String))&&args(x);
28
29 }
```

Example 3 AspectJ code of the Bridge design pattern

As far as the AO paradigm deals with the singletons only, it may seem that AO solutions for the creational design patterns have no sense. Nevertheless, the fact that aspects cannot be created or, be more precise, can only be created as one instance at a time, does not mean that AO analogues of *Abstract Factory* or *Factory Method* are senseless. Although in the AO world there are no factories, it is still necessary to obtain references to aspects for many times and the creational patterns are still very useful for this purpose. It will be demonstrated bellow what the AO solutions of creational patterns look like and such patterns can be applied.

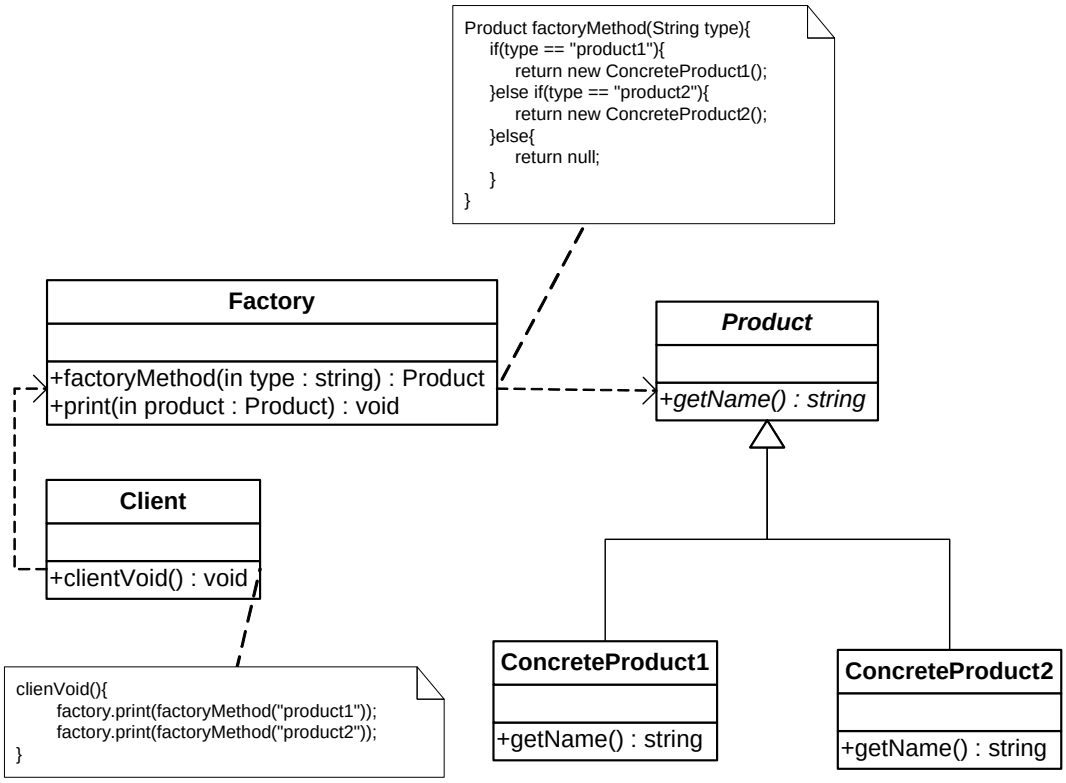


Fig. 12 Factory Method design pattern (OO solution)

The main purpose of the Factory Method design pattern is to define the interface for creating objects that belong to different classes. Usually the pattern defines an abstract method for creating the objects, which can then be overridden in subclasses with a view to specify the derived type of object that should be created. However, another variation of the pattern is used – the parameterized factory method (Fig. 12), in which the parameter that defines the

type of object is passed to the factory method (Gamma et al., 1994). The essential elements of the Factory Method pattern are:

- *Factory*, a class that contains the operation *factoryMethod* which returns the object of type *Product* depending on the requested parameter *type*,
- *Product*, an abstract class that contains the abstract operation *getName* and defines the interface of *Product* type objects,
- *ConcreteProduct1* and *ConcreteProduct2*, concrete *Product* classes that implement the *getName* operation using some concrete method, and
- *Client*, the class that invokes the *factoryMethod* of the *Factory* object.

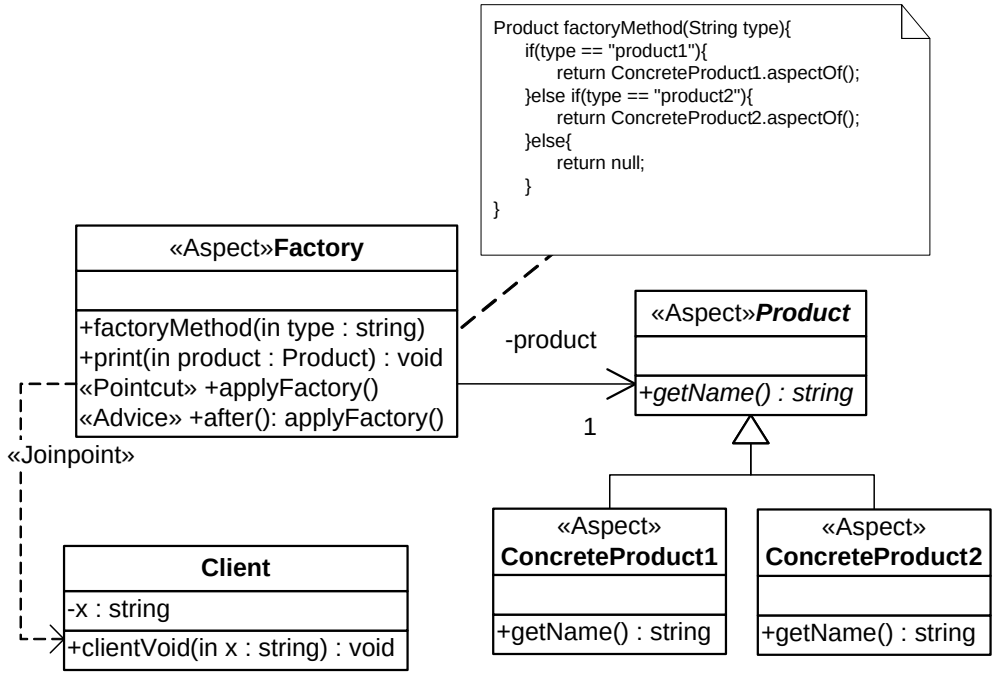


Fig. 13 Factory Method design pattern (AO solution)

In the AO solution (Fig. 13) the pattern helps to get a reference to the needed aspect that is defined by the given parameter. An analogous result as in the OO version of this design pattern is received. The difference is that instances of the classes are created each time the main factory method is executed, while in the AO pattern, the instance of an aspect is created only once. In Fig. 10, this method is named *factoryMethod* and is responsible for handling different references to aspects. The product aspects are defined as *ConcreteProduct1*

and *ConcreteProduct2* that extend the abstract aspect *Product* and a solution that consists only of aspects is received (Fig. 14).

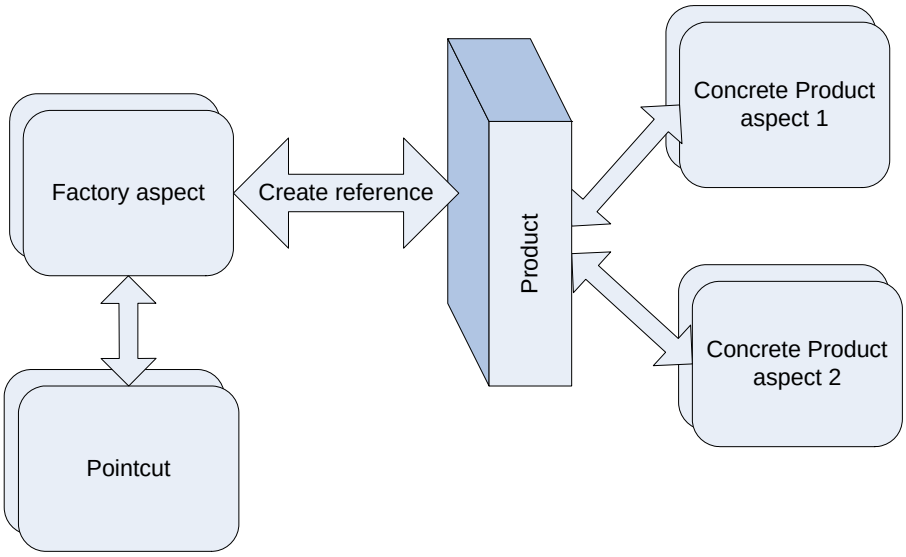


Fig. 14 The idea behind Aspect Factory Method

AspectJ code for this solution is presented in Example 4. The cardinality of *Product* association in Fig. 13 is set to one, because only one aspect at a moment could be used by *Factory* as defined in the code of the *Factory* aspect (Example 4).

```
1  public aspect Factory {
2
3       static public Product factoryMethod(String type){
4            if(type == "product1"){
5                 return ConcreteProduct1.aspectOf();
6            }else if(type == "product2"){
7                 return ConcreteProduct2.aspectOf();
8            }else{
9                 return null;
10           }
11      }
12
13      private void print(Product product){
14           System.out.printf(product.getName()+"\n");
15      }
16
17      pointcut applyRequest(String x) :
18           call(public void clientVoid(String))&&args(x);
19
20
21      after(String x): applyRequest(x) {
22           print(factoryMethod(x));
23      }
24 }
25
```

Example 4 AspectJ code of the Factory method design pattern

This code demonstrates that despite the fact that aspects are singletons. The AO pattern preserves all essential elements of the OO pattern. An example of the application of the AO Factory Method pattern is given in Fig. 15. In this example the complex Logger concern consisting of several aspects is used again (Fig. 8).

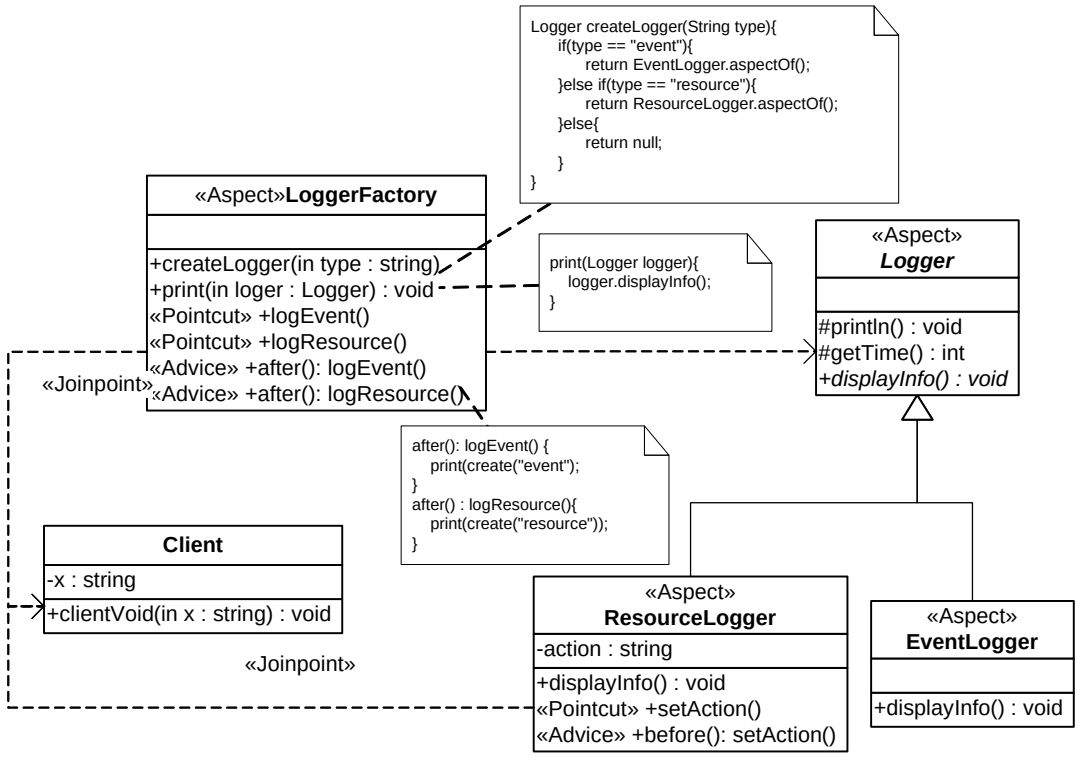


Fig. 15 Application of the AO Factory Method design pattern

In this case, the abstract aspect *Logger* represents product interface, the aspects *ResourceLogger* and *EventLogger* represent concrete products, and the aspect *LoggerFactory* represents a factory. The Factory operation *createLogger* represents a parameterized factory method and is responsible for referencing calls for the needed aspect. The pointcuts and advices in the factory *LoggerFactory* decide which logger should be handled by the *print* method. The pointcuts and advices are now separated from their behaviours that are defined in concrete logger methods named *displayInfo*. Such a structure of aspects is reasonable in the cases when concrete loggers need to have pointcuts and advices responsible for handling behaviours uncommon to other concrete loggers and defined directly in the concrete logger aspects as it is in the *ResourceLogger* aspect.

Finally, the Chain of Responsibility (CoR) design pattern as the most representative example of the AO design patterns with the reduced applicability is considered.

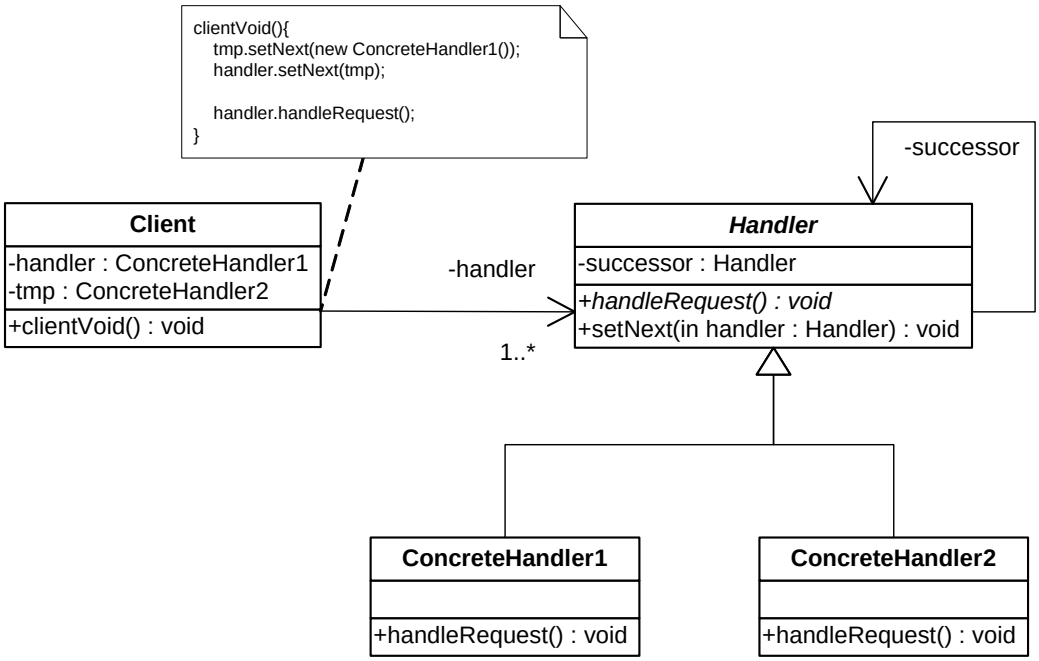


Fig. 16. Chain of Responsibility design pattern (OO solution)

The intent of the CoR design pattern is to "*chain the receiving objects and pass the request along the chain until an object handles it*" (Gamma et al., 1994). The essential elements of this pattern are (Fig. 16):

- *Handler*, an abstract class that contains the *handleRequest* operation and defines an interface of *Handler* type objects;
- *ConcreteHandler1* and *ConcreteHandler2*, concrete *Handler* classes that overwrite the *handleRequest* operation with a concrete method, that handles an appropriate request and forwards other requests to its successor in the chain; and
- *Client*, the class that invokes the *handleRequest*.

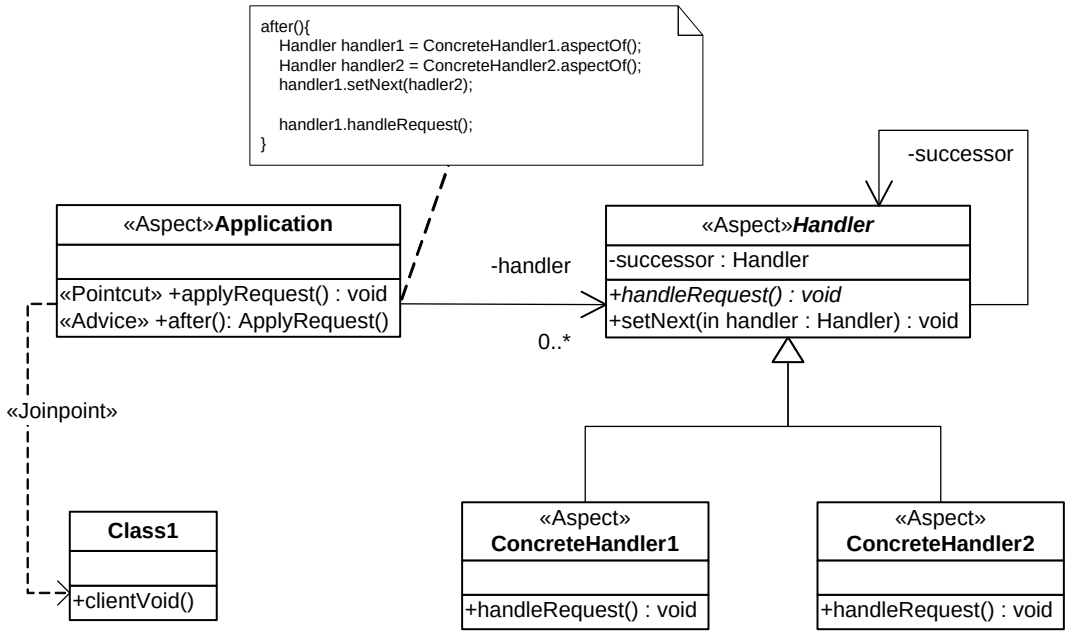


Fig. 17 Chain of Responsibility design pattern (AO solution)

In the AO solution (Fig. 17) of the CoR design pattern all classes are replaced by aspects as it is required by the proposed methodology. In this solution, differently than in the OO solution, it is impossible to use several instances of the same, concrete handlers (Fig. 17), because each concrete handler has one and only one instance. In the general case, the number of the concrete handlers is not limited. However, for the reasons of simplicity, Fig. 17 shows two concrete handlers only. One more restriction caused by the fact that aspects behave like singletons is impossibility to include the same aspect into the chain for several times, because in such a case the recursion created by the cyclic nature of the *successor* association (Fig. 17) falls into an eternal loop. Fig. 18 presents the problem solved by the CoR pattern consisting only of aspects.

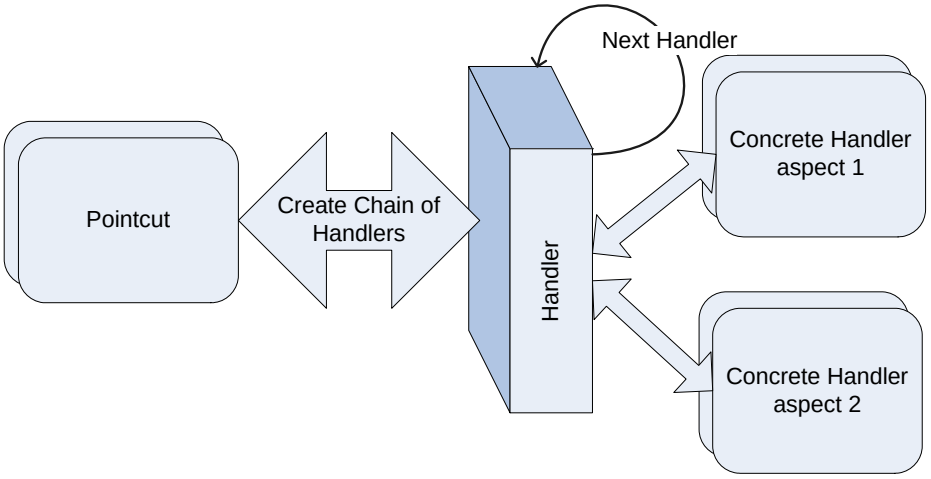


Fig. 18 The idea behind Aspect Chain of Responsibility

The example bellow (Fig. 19) demonstrates the applicability of the AO Chain of Responsibility pattern. In this example the same complex Logger concern consisting of several aspects is used (Fig. 8). The problem is changed slightly to be suitable to apply to the CoR design pattern.

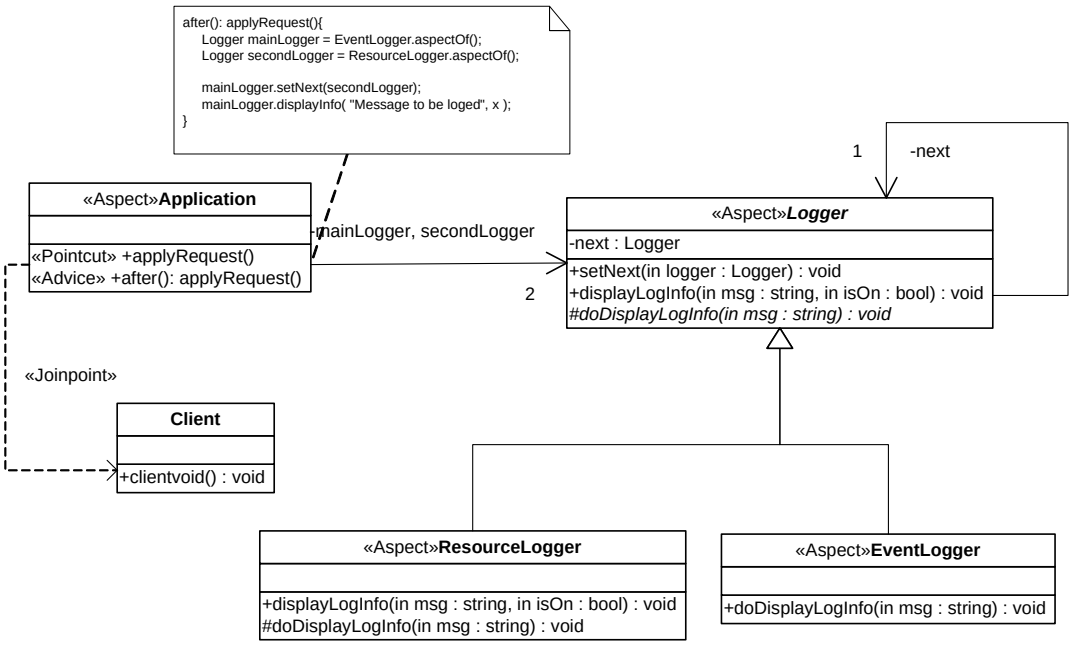


Fig. 19 Application of the AO Chain of Responsibility design pattern

Thus, there still are two different loggers – *ResourceLogger* and *EventLogger*, but there is a need to perform logging at some of join points using both of them, and using only one of them at some other join points. The rule when and how it should be done is defined by overwriting *displayLogInfo* in concrete

loggers. Concrete loggers can also have other defined pointcuts and advices that are specific only to concrete loggers *ResourceLogger* or *EventLogger*.

3.4. Summary

The chapter investigates the nature of software design patterns and demonstrates that some software design problems do not depend on a particular software engineering paradigm that is applied. However, it investigates in detail two paradigms only: aspect-oriented paradigm and object-oriented paradigm. The chapter proposes a classification of the ways of solving design problems using OO and AO design patterns. The proposed classification contributes to the better understanding of relations among the design problems and the design patterns. The subset of 23 GoF object oriented design patterns (20 GoF patterns) which solve paradigm-independent design problems and can be transformed into pure AO design patterns (GoF_{AO} patterns) has been identified. It has been proven that aspect-oriented constructs are sufficient to implement 20 of GoF design patterns, with regard that 5 of them are exposed to some reduced applicability. The rules how to transform 20 GoF design patterns into GoF_{AO} design patterns have been proposed and application of the transformation rules for the 23 GoF design patterns has been demonstrated. To our knowledge, the issues of the development of pure AO design patterns on the basis of the 23 GoF design patterns up to time were not investigated. In the aspect-oriented programming languages such design patterns can be implemented using only aspect-oriented constructs. The main conclusions of the chapter are as follows:

1. Although there are design patterns that provide mixed AO and OO solutions to solve paradigm-independent design problems, no publications that aim to investigate the class of pure AO design patterns that provide AO solutions solving paradigm-independent design problems have been presented.
2. Some software system design problems can be stated as independent from the particular software engineering paradigm that is applied.

However, this is confirmed for two paradigms only: aspect-oriented paradigm and object-oriented paradigm.

3. Taking into account that aspects and classes are similar constructs and that the main constraint for reusing OO structures to design aspects is that aspects are treated as singletons, it follows that similar and/or slightly changed structure of OO GoF design patterns can be used to build the pure AO design patterns. The only necessary task is to replace OO language constructs by the appropriate AO language constructs.
4. Although originally the 23 GoF design patterns have been proposed in the context of object-oriented systems, only two of these patterns – Prototype and Composite – solve specific object-oriented design problems. Design problems solved by 20 of 23 GoF patterns arise also in other paradigms including the aspect-oriented one.

The results of this chapter have been published in (Vaira, Čaplinskas, 2011b; Vaira, Čaplinskas, 2009).

Chapter 4

Empirical Evaluation of Application of Transformed Design Patterns

This chapter presents empirical evaluation results. Three case studies were performed for this aim. **Section 1** provides first case study which has been performed to evaluate design pattern transformation technique using one design pattern only, namely Factory Method design pattern. It is stated as the critical research case. **Section 2** provides second case study which has been performed to evaluate transformed GoF_{AO} design pattern applicability to redesign OO SimJ framework. The case corresponds to the demonstrative one. **Section 3** provides third case study which has been performed to evaluate transformed GoF_{AO} design pattern applicability to develop AO SimpleW framework from scratch. The case also corresponds to the demonstrative one.

4.1. Evaluation of the Hypotheses Using Case Studies

The main aim of this chapter is to present exemplary case studies showing how object-oriented design patterns can be redesigned into pure aspect-oriented design patterns and applied to design AO domain frameworks. Three case studies have been used for experimental evaluation of the proposed redesign technique. The case studies are performed to provide detailed analysis of redesign technique application to a real life system design. The above presented research consists mainly of theoretical reasoning and models of the redesigned patterns. However, it does not give any insights about practical application of the transformation technique except some hypothetical application context. The results of this research provide strong evidence in the form of implementation diagrams and detailed descriptions that such design

patterns are applicable in the design of real life systems. It can be stated as a qualitative experimental evaluation of the previous theoretical research.

A case study is an empirical research method that aims at the investigation of some phenomena in their context (Runeson, Höst, 2009). The thesis investigates the application impact of GoF_{AO} design patterns on the design of domain AO white-box frameworks. It is a positivist case study (Benbasat et al., 1987) because it measures variables, tests hypotheses and draws inferences from our samples to a whole population of AO domain white-box frameworks. An explanation of a given phenomena is desired but not in the form of a causal relationship. Both, the design results as well as the design process itself, are investigated. Different research methodologies can be applied for this aim. In the present thesis the constructive research methodology has been selected. According to Kari Lukka (Lukka, 2003), the constructive research is an experimental research procedure that can be used to test hypothesis by the development of an innovative construction, which implements the assumptions of these hypothesis. Generally, the novel construction should be an abstract notion with great, in fact infinite, number of potential realizations. In our case it is an AO domain framework. The innovative construction and its development process are considered as test instruments to validate, refine or even to develop entirely new hypothesis that is done by a profound analysis of what works (or does not work) in practice. Thus, the constructive research, in parallel with some other methodologies of experimental research, can be viewed as a kind of case research methodology. This methodology is "an alternative which applies a strong, problem-solving type of intervention and an intensive attempt to draw theoretical conclusions based on the empirical work" (Lukka, 2003). One of the advantages of the constructive research methodology is that it allows not only to test and investigate the properties of the innovative construction but also to study its development process. According to the conventional view, case studies should be used for falsification of the hypothesis only. Case study itself cannot prove any hypothesis and should be linked to some hypothetico-deductive model of

explanation. However, the closeness of the case study to real-world situations and its multiple wealth of details argue that this view is only partially correct. In some cases the results of case study can be successfully generalized (Flyvbjerg, 2004). It depends upon the case one is speaking of, and how it is chosen. The generalization ability of case studies can be increased by the strategic selection of cases (Ragin, 1992). The selected case should be either a critical or a typical case. A critical case is an atypical or extreme case that is used, in parallel with typical or representative cases, to test hypothesis in critical situations. From the point of view of our research, a representative example is the framework that is designed using at least one design pattern of each kind – creational, structural, behavioural – of AO GoF 20 patterns and a critical case is one that requires application of all AO GoF 20 patterns. For this experimental research one critical case and two representative cases have been selected.

Although this research similarly to any other case study cannot provide conclusions of statistical significance, different kinds of evidence, figures and statements are linked together to support strong and relevant conclusions. Some quantitative data are also used: such as code line number, number of data members, number of involved abstract and specialized entities, number of hook methods, number of defined abstract and specialized operations, number of invocations of these operations, etc. Mainly, the Guidelines for Conducting and Reporting Case Study Research in Software Engineering prepared by Per Runeson and Martin Höst (Runeson, Höst, 2009) are followed. Quantitative data has been collected by measurements, qualitative – by monitoring, analyzing, comprehending and generalizing the framework development process.

A case study approach has been used to test the stated hypothesis and the constructive research methodology (Crnkovic, 2010) was applied for experimental research on the application of aspect design patterns in the development of aspect-oriented application frameworks. In order to develop an aspect-oriented domain framework, one must design abstract aspects

representing hot spots. It is not an easy task to achieve. A number of object-oriented design patterns, first of all 23 GoF design patterns, have been proposed to ease the design of object-oriented frameworks (Gamma et al., 1994). A number of propositions (Hannemann, Kiczales, 2002; Noda, Kishi, 2001; Hachani, Bardou, 2003) have been proposed how to transform 23 GoF patterns in the aspect-oriented ones, however, with the purpose to develop more effective patterns for objects design. Of course, such patterns are not appropriate for aspects design. Present thesis demonstrates how 20 of GoF patterns can be transformed into pure aspect-oriented patterns (20 GoF_{AO} patterns) that are purported for aspects design. The experimental research has been designed with the aim to validate the following hypotheses:

- efficiency of designs is improved by the usage of pure AO design patterns combined with GoF design patterns;
- the usage of pure AO design patterns allows the designing of new kind of hot spots in white-box AO domain frameworks (i.e. hot spots represented by abstract aspects);
- the usage of pure AO designs patterns reduces crosscutting in AO domain frameworks;
- the development of AO domain frameworks using GoF_{AO} design patterns has no particular impact on the overall run-time performance of the applications developed using such frameworks.

In addition, this research investigates also the development of AO domain white-box frameworks considering that they are implemented in AspectJ and Java languages using 20 GoF_{AO} patterns. There exist two basic ways how an AO domain framework can be developed: 1) to develop the framework from scratch; 2) to transform some existing OO domain framework into aspect-oriented one.

4.2. A Case Study 1: Implementation of Pure Aspect-Oriented Factory Method Design Pattern

4.2.1. Research Methodology

The Factory Method design pattern has been chosen for this research to perform evaluation of the proposed design pattern transformation technique. The case of Factory Method design pattern can be treated as a critical case (Ragin, 1992) because it corresponds to the creational design patterns, which are less to be likely acceptable for redesigning them into aspects, because they are highly related to creation of objects. The creation of aspects is far different from the creation of objects, because aspects are singletons by their nature and its creation in most AO language implementations is handled by aspect weaver automatically. Hence, this case study presents strong evidence that even creational OO design patterns can be adapted to design AO design patterns. The main questions to be answered are if such AO design patterns are applicable in real life applications and if AO representation of Factory Method design pattern changes its purpose anyhow?

This research is based on qualitative data only. It is also slightly different from the remaining case studies. This case study analyzes only one design pattern and describes in details its transformation process. The resulted design of AO Factory Method design pattern is also applied in the second case study. The steps of this particular case study are highly related to the proposed transformation technique (Table 3).

Table 3 The research methodology of Case Study 1

Case study process steps	**Transformation of OO design pattern**
1. Analyze design pattern. Document observations and findings.	Analyze if OO design pattern can be implemented using singletons only. Decide whether it can be regarded as a candidate design pattern for rewriting it to AspectJ. Document the design using UML diagrams.
2. Perform redesign of design pattern. Design and implement aspects.	Replace all classes in the candidate pattern by aspects. Develop the necessary AspectJ code of aspects.
3. Evaluate resulted design pattern. Document observations and findings, and collect other qualitative data	Analyze the candidate pattern in order to discover and remove irrelevant data members and methods. Document the design using UML diagrams, describe observations and findings.
5. Apply resulted design pattern in the context of OO framework.	Rework the parts of the OO framework affected by some crosscutting of concerns. Develop the AspectJ code of aspects. Document the design using UML diagrams, describe observations and findings.
6. Analyze and generalize the collected data, evaluate hypothesis	Analyze collected data, comparing OO design, AO design and framework design of the analyzed design pattern.

4.2.2.Research settings

Factory Method GoF design pattern has been chosen for transformation into GoF_{AO} design pattern. According to (Gamma et al., 1994) Factory Method design pattern can have several variations of the final design structure. The one that has been described in the theoretical part of the present thesis relies on parameterized factory method, in which parameters are used to identify what type of product must be created. The design presented in this particular case study is based on inheritance mechanism, when objects are created by extending abstract factory class and defining a number of concrete implementations of factory method for creating each product.

OO simulation framework SimJ has been used as an experimental system in order to evaluate the application of the transformed Factory Method design pattern. SimJ framework contains only one crosscutting concern, namely, logging. SimJ is purported to design discrete events based simulation applications and can be regarded as a typical representative of simulation frameworks. All examples are presented using (Unified Modelling Language) UML class diagrams and stereotyped class diagrams for aspects. The resulted applications are implemented using Java and AspectJ (Kiczales et al., 2001) programming languages.

4.2.3. Observations and findings

In the case, when the Factory Method design pattern is used, it may seem that the AO solution has no sense, because Factory Method belongs to the creational pattern category and is highly related to creation of objects. In the AO paradigm in most cases one is dealing with the singletons only and in fact the creation of aspects cannot be managed directly by other aspects. However, it does not mean that the redesign technique can not be performed on Factory Method design pattern. The creation of aspects can be replaced by passing a reference to already created aspect. In order to do this *AspectOf* method instead of constructor method can be used. *AspectOf* corresponds to an analogue *InstanceOf* that is used for referencing singletons. It will be demonstrated that AO solution of Factory Method can be redesigned using proposed technique.

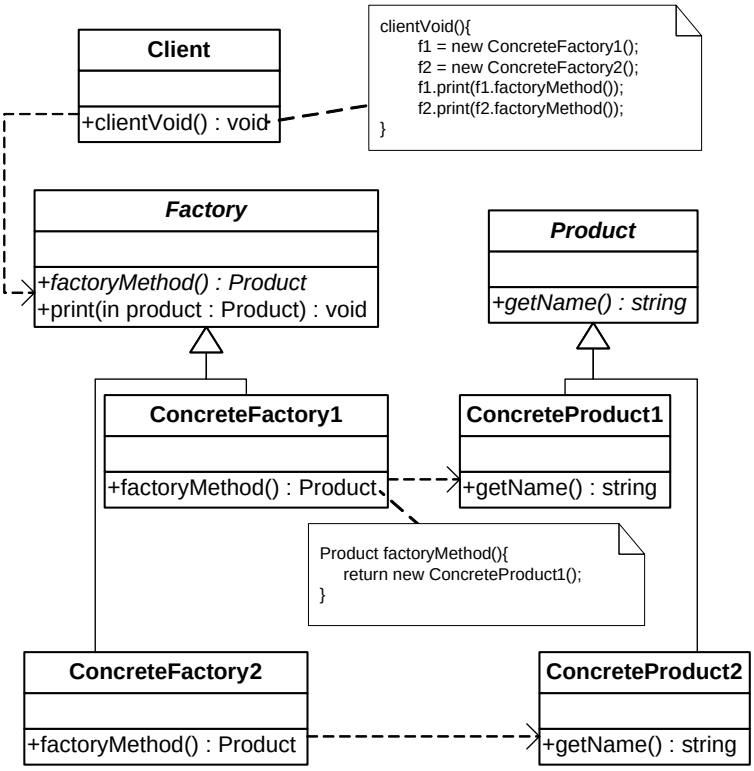


Fig. 20 Factory Method design pattern (OO solution)

The first step is to perform analysis of the pattern to inspect if it can be regarded as a candidate for rewriting. The Factory Method design pattern defines an abstract method that can be overridden by subclasses for creating objects that belong to different classes (Gamma et al., 1994). There are several other variations of the pattern (e.g. the parameterized factory method), but in this particular case the general one is used. The main elements of the general case of Factory Method (see Fig. 20) design pattern are:

- *Factory*, an abstract class that contains abstract operation *factoryMethod*, which is overridden by its subclasses,
- *ConcreteFactory1 and ConcreteFactory2*, concrete *Factory* classes overriding *factoryMethod*, which creates and returns the object of *ConcreteProduct1* or *ConcreteProduct2* respectively.
- *Product*, an abstract class that contains the abstract operation *getName* and defines the interface of *Product* type objects,
- *ConcreteProduct1* and *ConcreteProduct2*, concrete *Product* classes that implement the *getName* operation using some concrete method, and

- *Client*, the class that invokes the *factoryMethod* of the *Factory* object.

There is no critical reason indicating that Factory Method design pattern can not be implemented using singletons only. Abstract classes can be replaced by abstract aspects, subclasses by specializing aspects. The constructors of *ConcreteProduct1* and *ConcreteProduct2* can be replaced by *AspectOf*. All other operations remain the same as in classes.

When it is decided that the Factory Method is a candidate for redesigning, the second step can be performed in Fig. 21. The resulted AO Factory Method solution helps to get a reference to the necessary aspect defined by specialized Factory aspect. This is an analogous solution to that of OO Factory Method design pattern. The main difference is that instances of aspects are created only once and each time *factoryMethod* is executed particular *Product* instance is passed as an argument.

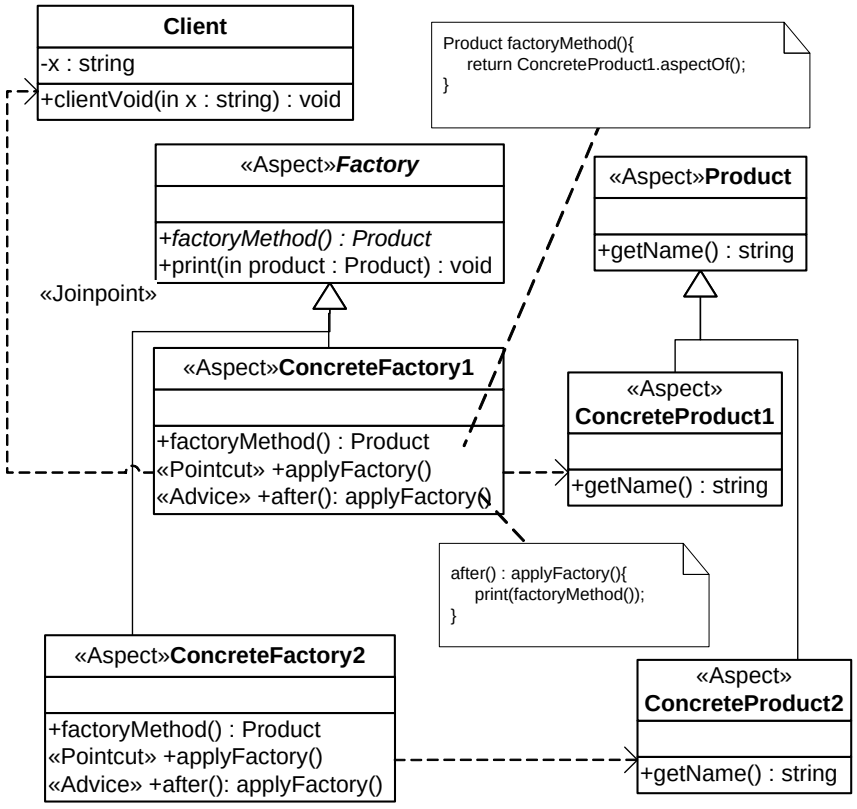


Fig. 21 Factory Method design pattern (AO solution)

The last step of evaluation of resulted pattern involves possible refactorings to enhance the resulted design and to test its applicability. The main variation of the pattern can be performed by changing or adding pointcuts and advice. The

current model includes pointcuts and advice in subaspects of Factory aspect and in this way it is defined when *factoryMethod* operation is invoked. Another place for defining pointcuts and advice could be subaspects of Product aspect. More comprehensive designs of pattern behaviour could be achieved by predefining some pointcuts or advice in abstract aspects. The important difference between AO design pattern and its OO analogue is that the developer is limited by the number of predefined subaspects that can be used at the same time (except the above mentioned cases of per object or per control flow aspects). However, it does not change the principal behaviour of this design pattern and demonstrates that AO design pattern preserves all essential elements of the OO pattern.

An example of the application of the AO Factory Method pattern is analyzed in the following part of the section. In this example, the complex logging concern in a simulation domain framework is analyzed.

SimJ simulation framework is used as an experimental system providing necessary context for implementing AO Factory Method design pattern. The main research interest is concentrated on logging concern, which has a crosscutting issues that need to be eliminated and the feature of logging that needs to be made customizable. SimJ is a simulation framework used for developing simulation applications based on discrete events.

The logging concern in a framework suffers from crosscutting. Pieces of the code belonging to it are scattered and tangled through the remaining part of a framework. The complexity of a logging functionality of this framework makes it a sufficient candidate to apply the AO Factory Method design pattern presented in Fig. 21. The framework has several different kinds of things to be logged and must remain customizable in a concrete specialization of a framework. The current version of the framework allows customizing logging. However, it is handled beyond the bounds of logging concern individually by every entity that needs to be logged. The main purpose of application of AOP is to exclude all pieces of code related to logging concern and combine them in

aspects. Although the design of these aspects is not an ordinary task to complete, design pattern could be applied to handle it.

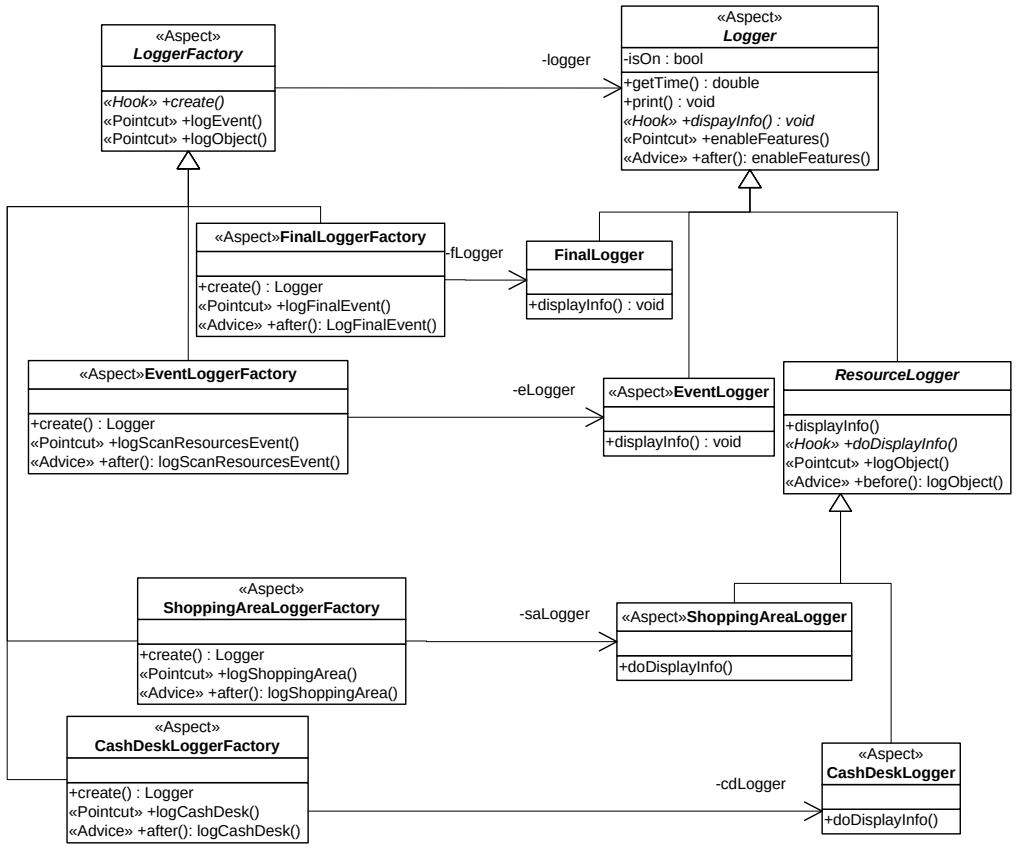


Fig. 22 Application of the AO Factory Method design pattern

The AO Factory Method design pattern was introduced in order to deal with the following issues: different logging behaviour for resources and several kinds of events were necessary as well as the triggering complexity of this behaviour required its separation. Different behaviour of logging was modelled using product hierarchy in Factory Method pattern. The triggering structure of logging behaviour was designed using hierarchy of factories Fig. 21. The resulted implementation of logging concern is presented in Fig. 22. The UML diagram contains complete design that includes two additional instances of Template Method (design pattern is usually used in composition with factories). The stereotype “Hook” is used to denote customizable framework methods in aspects.

Consequently, the following advantages can be noticed:

- all the logging functionality and related code is localized in one place,

- the customization of logging concern can be carried out separately from the remaining hot spots.

This also means that maintenance and unplug ability features of the logging were increased. This implementation allows flexible customization so that logging of events and resources can be done separately and the join points triggering logging behaviours can be customized independently. A high number of aspects can be considered as a shortcoming. This is probably related to the complexity of the logging concern behaviour. However, more often a higher number of smaller entities is considered as an advantage rather than a shortcoming.

4.3. A Case Study 2: Application of Pure Aspect-Oriented Design Patterns in the Redesign of Aspect-Oriented Frameworks

4.3.1. Research Methodology

For this particular case study the case that is constrained by the existing design of the OO framework is used. In such case some OO part of the framework design should be replaced by the relevant AO design. It is obvious that only those parts of a framework that are affected by some crosscutting of concerns should be reworked. If the tangled and scattered code over the whole framework is present or some Singletons are implemented, it is advisable to consider the reasonability of the implementation of hot spots in the form of aspects (Monteiro, 2006). The main steps of the research methodology are summarized in Table 4. It provides some cycle that is finished. The resulted data is compared at several iterations in order to reject or promote the hypothesis raised. The qualitative data produced by this research includes a brief description of the research steps performed, UML diagrams of the resulted design patterns and the summarization of the results confirming hypothesis. The quantitative data correspond to the data of the measurements carried out for each iteration of the cycle.

Table 4 The research methodology

Case study process steps	**Reworking of OO framework**
1. Identify what aspects should be designed.	Identify crosscutting, which should be implemented as aspects in the OO framework. Identify what parts of the framework are affected by crosscutting and should be reworked. Decide what new hot spots are to be added to the framework and which of aspects should be used to implement these hot spots.
2. Decide what design patterns should be applied to design identified aspects	Decide what aspect should be designed in order to implement new hot spots, examine what design problems should be solved designing these aspects, and determine which of the AO GoF 20 design patterns can be applied for this aim.
3. Design and implement aspects, document observations and findings, and collect other qualitative data.	Design required aspects: apply required AO GoF 20 patterns, document the design using UML diagrams. Observe and describe in details the whole design process. Rework the parts of the OO framework affected by some crosscutting of concerns, develop the AspectJ code of aspects.
4. Perform measurements, test code and collect quantitative data.	Use build-in tools of development platform (Eclipse, NetBeans) to collect static quantitative data. Prepare required test cases, perform measurements and collect quantitative dynamic data.
5. Evaluate the structure of the code according to the criteria.	Check whether the AspectJ code is already acceptable. Improve the design of code and go back to step 3 if the refactoring of code is still required.
6. Analyze and generalize the collected data, evaluate hypotheses	Analyze the collected data for each design pattern separately comparing both OO and AO framework designs.

4.3.2.Research Settings

OO simulation framework SimJ has been chosen for transformation into AO domain framework. SimJ is relatively small academic framework containing only one crosscutting concern, namely, logging. It is purported to design simulation applications based on discrete events and can be regarded as a typical representative of simulation frameworks. SimJ provides 5 hot spots

(simulation, events, resources, entities, entity factory). SimJ is relatively mature framework which has already been improved many times.

All required code for the framework has been written in Java and AspectJ programming languages. Eclipse SDK 3.6 and NetBeans IDE 6.9.1 development platforms have been used for developing and testing the framework. Eclipse SDK 3.6 has been used as run time environment for the SimJ. All measurements have been done on computer with AMD Athlon dual core 2.61 GHz processor, 2 GB of RAM, and Microsoft Windows XP SP3 operating system, using built-in tools of Eclipse SDK 3.6 and NetBeans IDE 6.9.1.

The design results are documented using UML-like notation. The stereotype <<hook>> is used to note the hooks. The hot spots are commented by appropriate notes.

4.3.3. Observations and Findings

The OO framework SimJ provides 5 hot spots and contains only one crosscutting concern, namely, logging. The framework is designed in such a way, that logging is split in 3 specialized parts (a part for each hot spot) that, using appropriate hooks, can be adapted independently for a particular application. Thus, the logging affects 3 of 5 hot spots. The code, related to logging, is scattered over in 7 classes. It has been decided to remove this code and to use it to develop abstract aspects that would implement new hot spot named Logger. It was necessary to remove this code in such a way that the remaining code is still correct. The amount of efforts required for reworking will not be discussed, because it is out of scope of this research. However, in our case it was not a big problem. The AO Template Method design pattern was applied to combine the removed code into aspects. This is the way in which 3 aspects that implement default behaviour to all resource logging have been designed. Such a solution allows customizing in the applications some part of this behaviour because, in our case, the AO Template Method pattern allows to provide an abstract method implementing a hook. Since the default

behaviour of the original OO framework provides only one kind of events, exactly one additional aspect to implement the default behaviour for event logging has been designed. It has no abstract methods and, consequently, does not provide any hooks. For the reasons of efficiency, it has also been decided to use this aspect to implement the subsidiary logging related functionality (printing messages, getting time values). However, this functionality should be shared with the resource logging as well. It has been decided to apply the AO Adapter design pattern as the most reasonable design decision to solve this problem. The resulting design is presented in Fig. 23. It provides one additional hot spot (Logger) that can be customized in the applications by overriding the provided hook method.

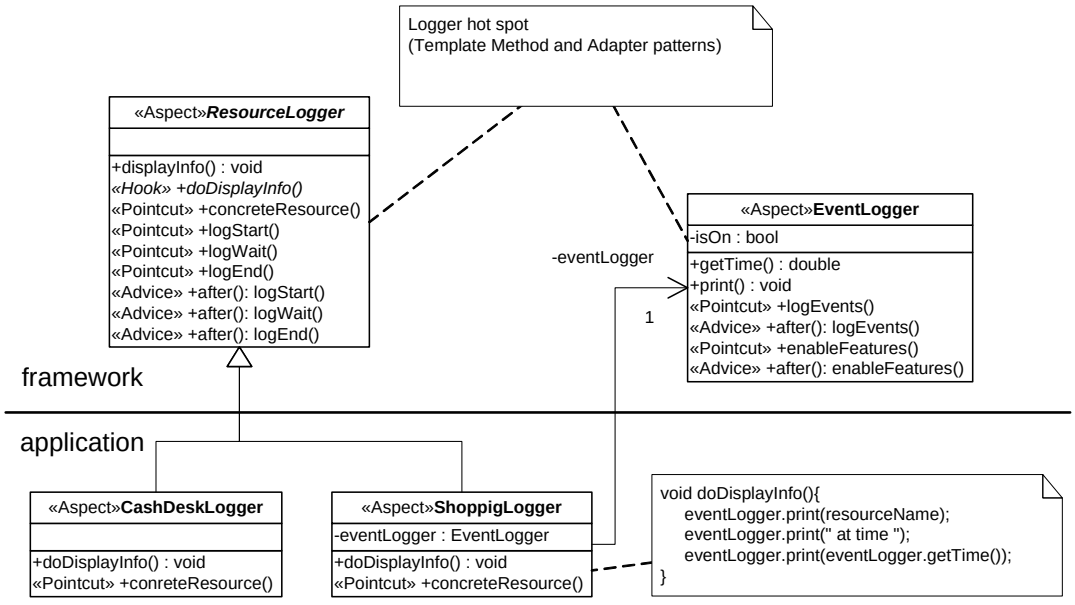


Fig. 23 SimJ Logger concern after first development iteration

This design improves maintainability and unplug ability of the logging comparing to the original OO framework because all the logging functionality and the related code is collected together and the resource logging can be customized using the additional hot spot. The quantitative data related to this design iteration will be presented and analyzed in the next section.

It is obvious, that the design can be further improved, because it does not allow to customize the event logging. For this reason the second design iteration has been performed. Since it is reasonable to model logging behaviour of

resources and events by the behaviour of a hierarchy of more specific loggers (Fig. 24), the AO Factory Method design pattern has been applied to build this hierarchy. This design pattern separates also the logging behaviour from the entities that trigger this behaviour, because it splits the hierarchy into the factories and product hierarchies. In the product hierarchy, all required operations can be lifted to the top, to the abstract aspect Logger, therefore the AO Adapter design pattern is no longer necessary (Fig. 24). On the other hand, the AO Template Method design pattern was applied to design hooks for Final Logger and Event logger. So, in the final design 3 additional hook methods were designed for the logging hotspot (Fig. 24).

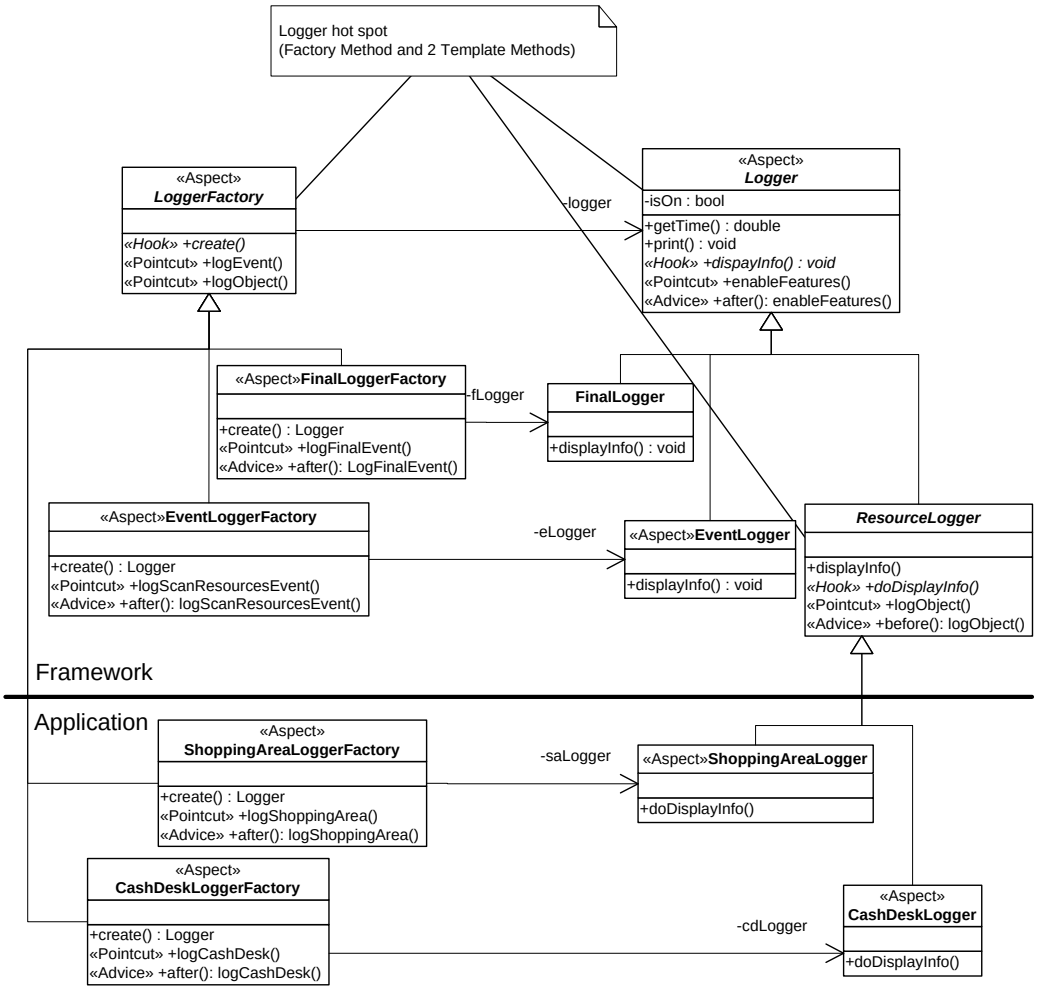


Fig. 24 SimJ Logger concern after second development iteration

Thus the final design is an evidence that AO GoF 20 design patterns allows to design abstract aspects that facilitates the extension of OO framework with the

new hot spots and that application of these patterns reduces crosscutting in the framework.

4.3.4.Measurements and Data Analysis

During both SimJ framework development iterations some quantitative data about the structure of code and about performance of applications produced using AO SimJ framework have been collected. They are presented bellow (Fig. 25, Fig. 26) by corresponding bar graphs. Every graph contains three bars: “O” bar corresponds to OO implementation, “A1” bar to AO implementation after first development iteration, “A2” bar to AO implementation after second development iteration. The measurements in Fig. 25 are presented as quantities and in Fig. 26a - 26b as milliseconds. Data about the structure of code (Fig. 25) demonstrate that the complexity of code generally decreases. Numbers of code lines and data members remain almost the same. The first AO development iteration produced less code than the OO analogue. However, the second design iteration increases the number of code lines and it becomes greater than in the OO analogue, but the change is insignificant and can be considered as acceptable. Besides, the increase of lines is caused by the extended capabilities of logging customization, but not as a cause of the application of AO design patterns. The greater number of entities (i.e. classes and aspects) is caused by finer granularity of the implementation code. It is useful because entities are becoming smaller and less complex. During both development iterations customization was extended by providing one additional AO hot spot. However, the number of hook methods has been decreased comparing to OO implementation. This is caused by reduced crosscutting of logging concern. The two additional hook methods have been provided by extended customization during second development iteration A2 than during A1. The number of methods, advice, calls, and pointcuts decreases also in both, A1 and A2 cases. The first development iteration produced fewer methods and less advice than the second, while the second iteration – fewer external calls and pointcuts than the first one.

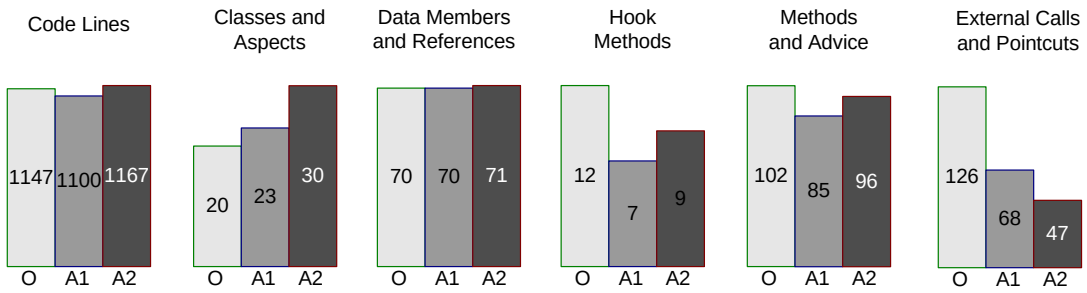


Fig. 25 static quantitative data of measurements (SimJ framework)

The tests of applications produced by the AO SimJ framework revealed some interesting data. An application has been produced after every design iteration and for each application two tests were performed. In the first test, the application was executed using the logging that aggregates the registered data (Fig. 26a), in the second test a usual logging functionality has been used (Fig. 26b). Each test has been executed 50 times in two different modes: 50 separate executions of the application (execution time) and 50 application executions in a continuous cycle (continuous execution time). All executions were performed using the same configuration of the application. Every test was performed for 1000000 simulation time units, which are equal to approximately 44000 cycles of simulation processing and 25 test executions per testing case. The results are presented as average values of all 50 executions.

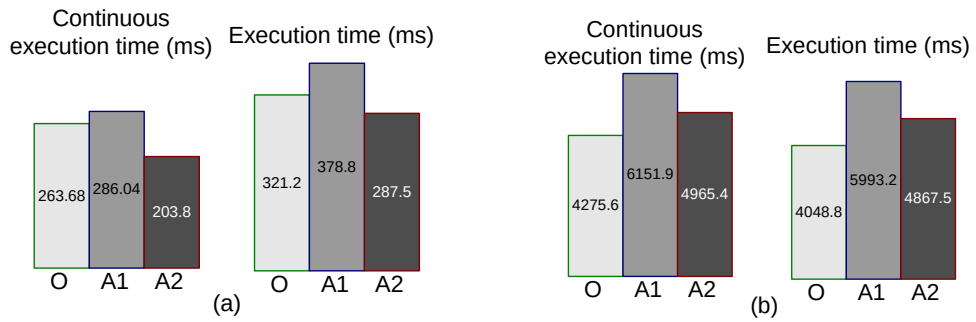


Fig. 26 testing data of measurements (SimJ framework)

After the first design iteration the performance of the application, especially in the second mode of execution, decreased a little, but it increased again after the second design iteration. This was an unexpected result that cannot be completely explained on the basis of our observations and requires further investigation. However, the most reasonable explanation suggests that the initial loss of performance in the second iteration and its restoration in the third

iteration is directly related to the particular design patterns that have been applied.

4.4. Application of Pure Aspect-Oriented Design Patterns in the Development of Aspect-Oriented Frameworks: A Case Study 3

4.4.1.Research Methodology

This case study is contrary to the second case study presented above. It is performed by developing the AO framework from scratch (Table 5).

Table 5 The research methodology

Case study process steps	**Developing AO framework**
1. Identify what aspects should be designed.	Identify modules that should be designed in a crosscutting manner and should be implemented as aspects. Decide which hot spots should be designed in using AOP and which OOP in the framework.
2. Decide what design patterns should be applied to design identified aspects	Decide what aspect should be designed in order to implement AO hot spots, examine what design problems should be solved designing these aspects, and determine which of the AO GoF 20 design patterns can be applied for this aim.
3. Design and implement aspects, document observations and findings, and collect other qualitative data.	Design required aspects: apply required AO GoF 20 patterns; document the design using UML diagrams. Observe and describe in details the whole design process. Rework the parts of the OO framework affected by some crosscutting of concerns. Develop the AspectJ code of aspects.
4. Perform measurements, test code, and collect quantitative data.	Use build-in tools of development platform (Eclipse, NetBeans) to collect static quantitative data. Prepare required test cases, perform measurements, and collect quantitative dynamic data.
5. Evaluate the structure of the code according to criteria.	Check whether the AspectJ code is already acceptable. Improve the design of code and go back to the step 3 if the refactoring of code still is required.
6. Analyze and generalize the collected data, evaluate hypothesis	Analyze the collected data for each design pattern separately, comparing both OO and AO framework designs.

In such case the crosscutting behaviours should be identified at the early development phases. The main steps of our research methodology are summarized in Table 5. It provides some cycle that is finished. The resulted data is compared at several iterations in order to reject or promote the hypothesis raised. The qualitative data produced by this research includes: brief description of the research steps performed, UML diagrams of the resulted design patterns, and the summarization of the results confirming hypothesis. The quantitative data correspond to the data of the measurements carried out for each iteration of the cycle

4.4.2.Research Settings

SimpleW AO framework has been developed aiming to implement simple personal web portals from scratch. The framework has four crosscutting concerns (logging and error detection, synchronization of content navigation processes, synchronization of content configuration processes, data security and validation) and provides three hot spots (Logger, Navigation, Configuration) implemented as aspects. It also provides some non aspect-oriented hot spots to specialize various types of interactive resources (menus, links, and contents) as well as features such as content presentation language, user management mechanisms, etc. It was designed for the research purposes.

All required code for the framework has been written in Java and AspectJ programming languages. Eclipse SDK 3.6 and NetBeans IDE 6.9.1 development platforms have been used for developing and testing the framework. Apache Tomcat 6.0 and MySQL 5.1 were used as run time environment for the SimpleW framework. All measurements have been done on computer with AMD Athlon dual core 2.61 GHz processor, 2 GB of RAM, and Microsoft Windows XP SP3 operating system, using built-in tools of Eclipse SDK 3.6 and NetBeans IDE 6.9.1.

The design results are documented using UML-like notation. The stereotype <<hook>> is used to note the hooks. The hot spots are commented by appropriate notes.

4.4.3. Observations and Findings

During the initial analysis of the requirements for the second framework it has been decided that 13 modules have to be designed: configuration, database, file, language, logging, resource, menu, breadcrumb navigation, security, session, system menu, user and web tier. Four modules were identified as the crosscutting concerns of the framework: configuration, security, breadcrumb navigation and logging. Every non-crosscutting module can be added by using OO module factory hotspot and every module that is related to content demonstration can be added by using resource OO factory method. Several core modules also contain OO hotspots: data module, menu and web tier. Data module hot spot allows the development of additional database handlers. Menu hot spot allows the development of any number of required menus with contents. Web tier contains several OO hot spots allowing the development of various web interface components. AO hot spots have been designed for the following crosscutting modules: configuration module, breadcrumb navigation module and logging module. Configuration module allows additional context loading features to be implemented using context loading hot spot. Breadcrumb navigation module provides navigation hot spots that can be used for adding new types of navigations and defining additional behaviours to the existing ones. Logging module also has several hot spots for developing different logging performers and logging behaviours in a module. Security module, at the moment, does not require any hot spot to be designed. The details of OO hot spots and modules, except some quantitative data, are not discussed in this research. The main focus of this case study is on the crosscutting modules. All four crosscutting modules are explained using UML diagrams and implementation code examples.

Configuration module is responsible for defining and loading framework configuration parameters. Some general framework parameters can be defined and loaded using configuration module. However, there are many parameters related to other modules: language module, data module and session module. Parameter defining can be implemented separately by using OO design.

However, the loading of the necessary parameters every time the web site is accessed or the parameters are changed requires to design a configuration module in a crosscutting manner. This is the main reason that context loader hot spot has been designed. The design of configuration module has been carried out using 3 design iterations.

During the first design iteration four different aspects have been designed: *ConfigurationContextLoader*, *DataContextLoader*, *LanguageContextLoader* and *SessionContextLoader* (Fig. 27).

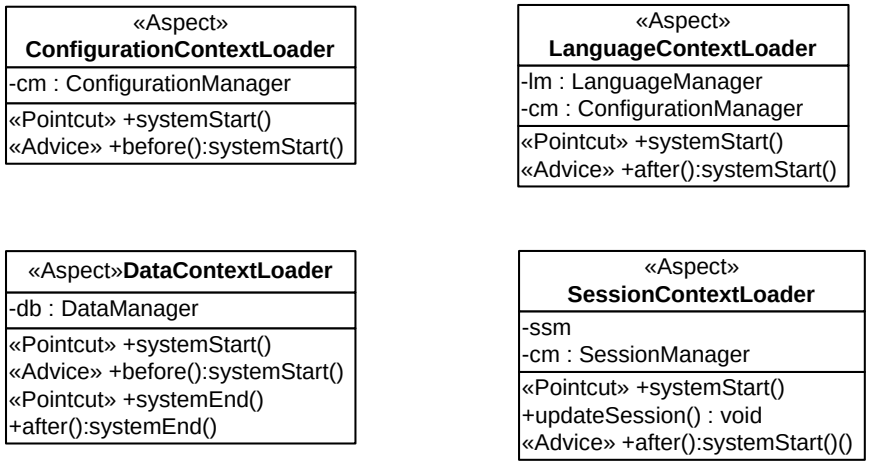


Fig. 27 SimpleW Context Loader concern after first development iteration

Although the pointcuts in the aspects are represented by the same name the actual contents of a pointcuts is slightly different. The difference is caused mainly by the necessity of maintaining a particular ordering of context loading. Such design could be optimized by choosing common pointcut for all aspects and defining the context loading order in advice. That's what exactly has been done in the second design iteration of context loader module (Fig. 28).

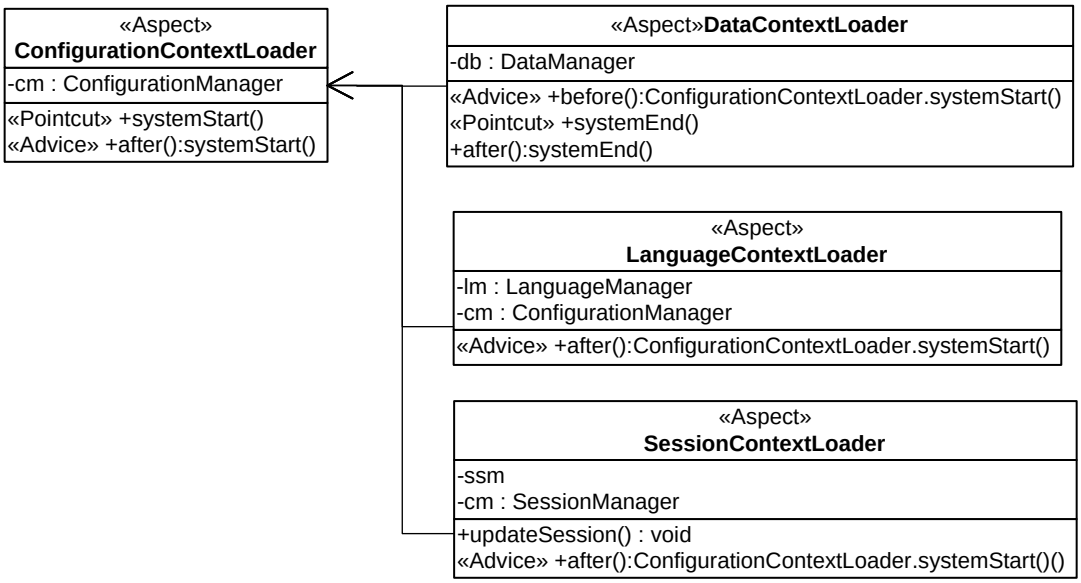


Fig. 28 SimpleW Context Loader concern after second development iteration

Such context loader design can easily be supplemented by a necessary behaviour just developing another context loader. However, one of the main goals of this research is to provide white-box framework hot spots. The hot spots of such type are designed by introducing abstract aspects and inheritance mechanisms. All in (Fig. 28) defined aspects seem to follow very similar algorithm which behaviour only partially changes in the particular aspect. The AO template method design pattern can be used to solve the situated design problem (Fig. 29).

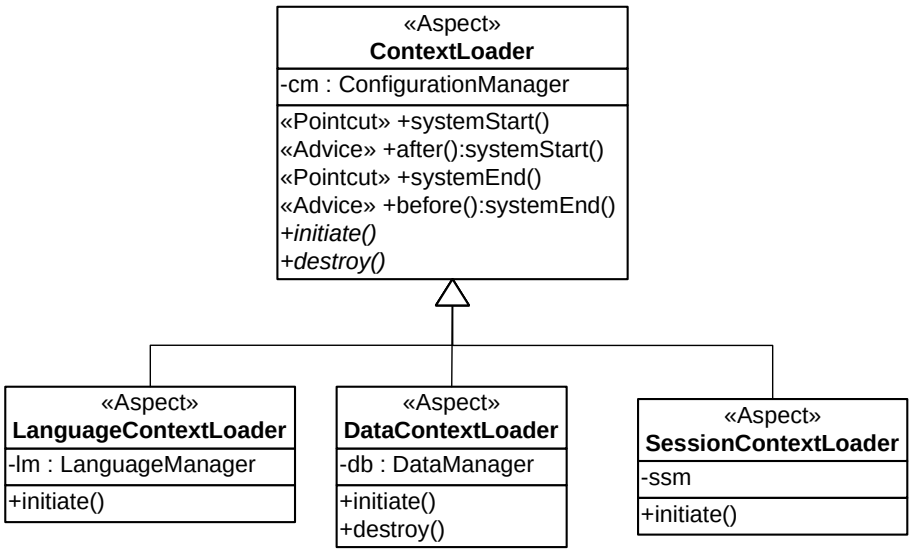


Fig. 29 SimpleW Context Loader concern after third development iteration

The resulted context loader design contains abstract *ContextLoader* with predefined context loading behaviour and two template hook methods. The context loading ordering problem has been solved by using precedence declarations in concrete context loader aspects. The precedence of one aspect over another can be declared in any of the concrete aspects if it is necessary. The loading of general parameters of the framework has been assigned to the *ConfigurationManager* class and is considered as a placeholder for defining the start and end system join points, so it is no more necessary to deal with it in aspects.

Breadcrumb navigation module resulted as several independent aspects and a part of system menu class after its first development iteration (Fig. 30).

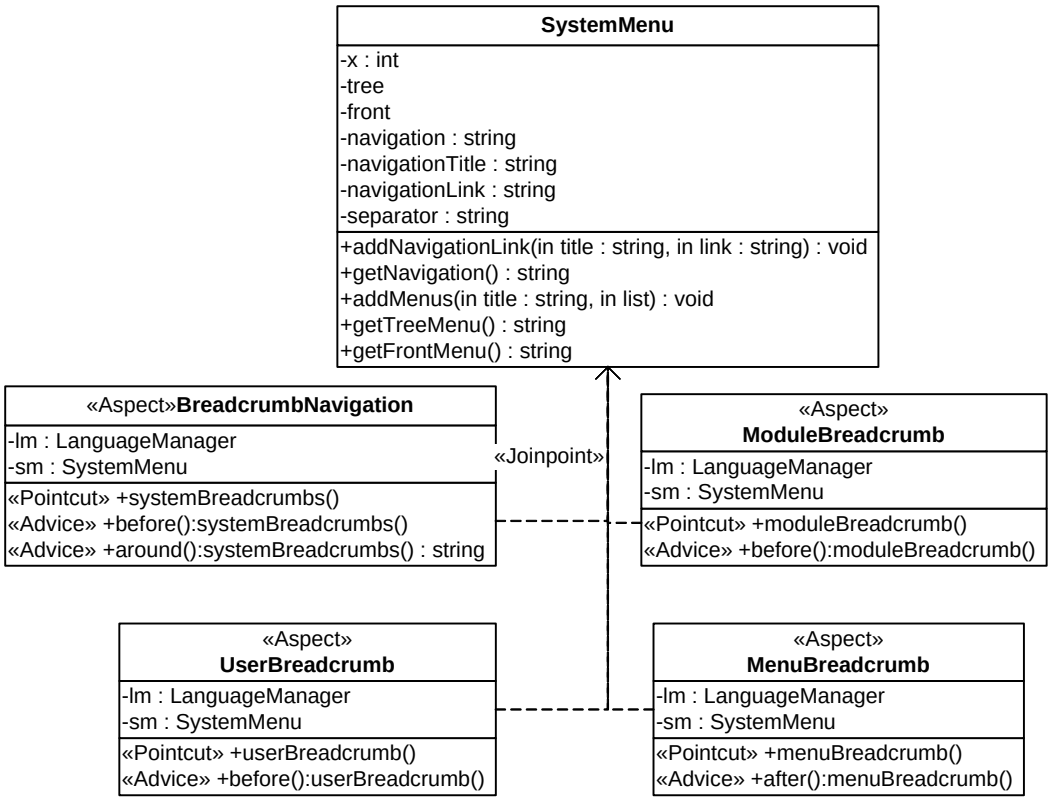


Fig. 30 SimpleW Breadcrumb Navigation concern after first development iteration

The main operations of breadcrumb navigation are still hardwired in a system menu class which is responsible for creating and representing system administration menus. Default behaviour of the breadcrumb navigation is managed by *BreadcrumbNavigation* aspect and breadcrumb behaviour of other modules in corresponding aspects. Any additional behaviour could be added as

a separate aspect using system menu as a placeholder for join points and its operations for altering breadcrumb navigation behaviour. However, system menu class is not proper for holding operations of a breadcrumb module. It is also forced to be a Singleton class to fit the needs of breadcrumb module.

After the second development iteration all the necessary operations and attributes have been relocated into *BreadcrumbAspect* and the system menu class has been transformed into a traditional class which can have as many instances as necessary (Fig. 31).

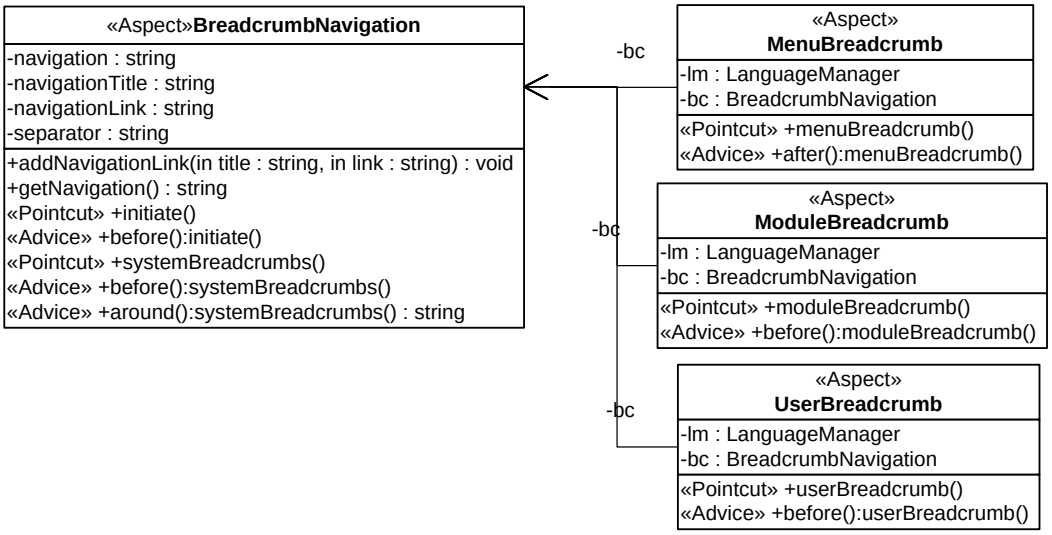


Fig. 31 SimpleW Breadcrumb Navigation concern after second development iteration

Such design is more modular, therefore it is not related to system menus anymore. Thus, direct accessing of *BreadcrumbNavigation* aspect instance introduces some coupling. White-box type of hot spots requires inheritance mechanisms to be provided. It would also be desirable that several types of breadcrumb navigations could be defined in the framework.

On the other hand, it is necessary to preserve the *BreadcrumbNavigation* aspect ability to attach additional functionality dynamically. Such design problem formulation suggests that Decorator GoF_{AO} design pattern should be applied. The resulted design completely satisfies all the required needs (Fig. 32).

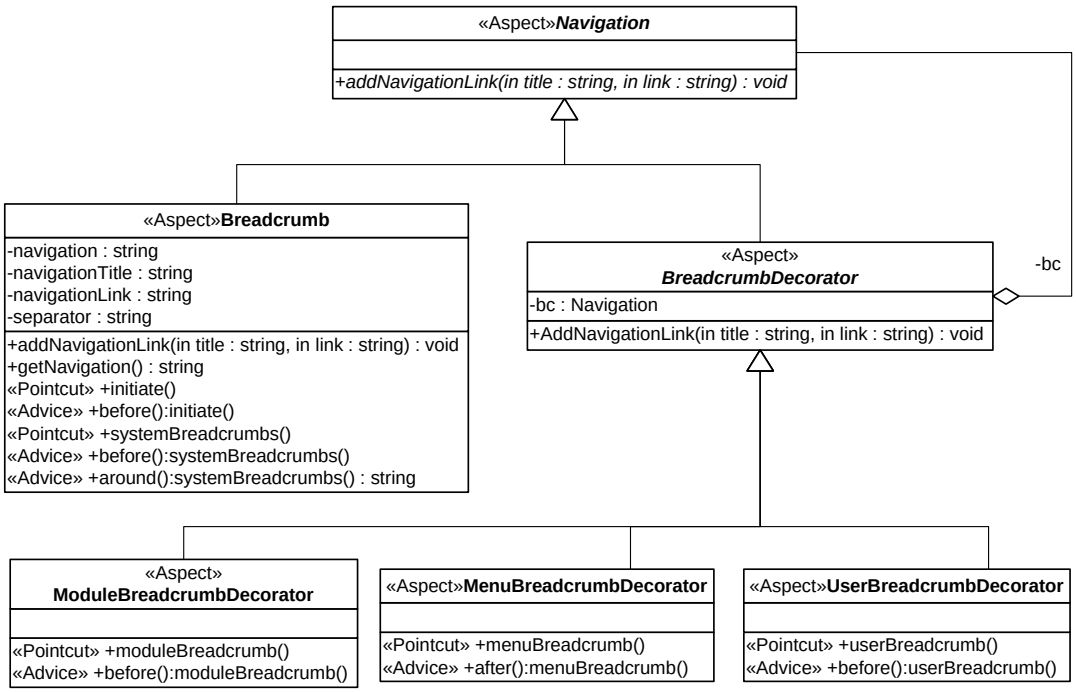


Fig. 32 SimpleW Breadcrumb Navigation concern after third development iteration

There are two hot spots developed in the breadcrumb navigation module. *Navigation* aspect defines interface of breadcrumb navigation, so any number of breadcrumb navigations can be developed in addition. Breadcrumb decorator defines default behaviour describing how the additional functionality of breadcrumb navigation should be added. Although the behaviour is predefined, the customizable part of this hot spot still remains. The aspects inheriting abstract *BreadcrumbDecorator* define pointcuts and advice, specifying exact join point where such behaviour should be applied.

Both concerns described above undergo some changes during all the development iterations. However, some relatively small and not complex concerns may not require so many iterations to be performed. Security concern is one of the smaller concerns in SimpleW framework. It has taken only two iterations for the final design to be developed. The first security concern design has resulted as stand alone *SecurityFiltering* aspect (Fig. 33).

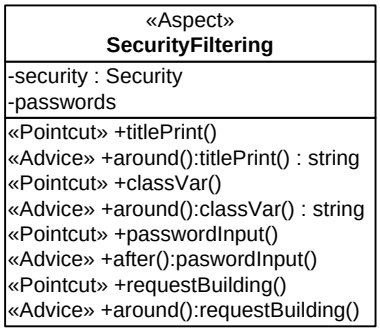


Fig. 33 SimpleW Security Filtering concern after first development iteration

The requirements for the framework security are already fully covered by such design and no security hot spots are necessary. All the required behaviour of the security filtering is defined in several pointcuts and advice of the same aspect. However, some part of the two pairs of pointcut and advice do not only perform filtering of the necessary data, but also trace password inputs. It is desirable to transfer unrelated functionality to a new aspect in order to get a more comprehensive representation (Fig. 34).

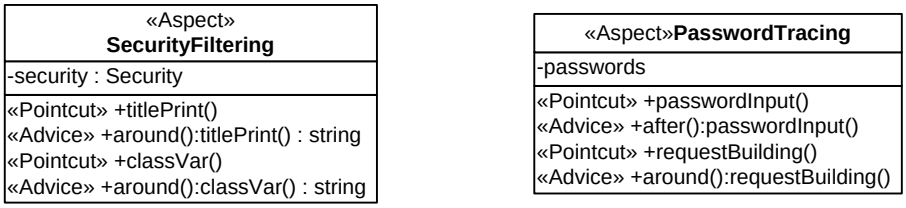


Fig. 34 SimpleW Security Filtering concern after second development iteration

The *PasswordTracking* aspect is actually designed using Wormhole design pattern (Laddad, 2003). This pattern is one of the AO paradigm specific design patterns that solves object design problem. It allows capturing some data members of one object and transferring them to other objects that may even not know about the existence of the data owner object. In the *PasswordTracing* aspect Wormhole pattern is applied to trace all possible password inputs and transfer the collected data to security object. It must be noticed that the application of this design pattern has been performed indirectly and as such patterns are out of scope of this particular investigation the overall impact of application of such patterns has not been analyzed in details. On the other hand, it confirms the idea that paradigm-independent design patterns should be

used together with paradigm-dependent design patterns in order to achieve optimal design results.

The last crosscutting concern, namely Logging concern, corresponds to the most complex crosscutting concern of this framework. In contrast to the Security concern, which has been developed without any hot spots and any GoF_{AO} design patterns, Logging concern has been designed with three hot spots and three design patterns. The overall design process covers 3 iterations. The first development iteration design has been developed using three different aspects: *ErrorHandler*, *MessageHandler*, *ExceptionHandler* (Fig. 35).

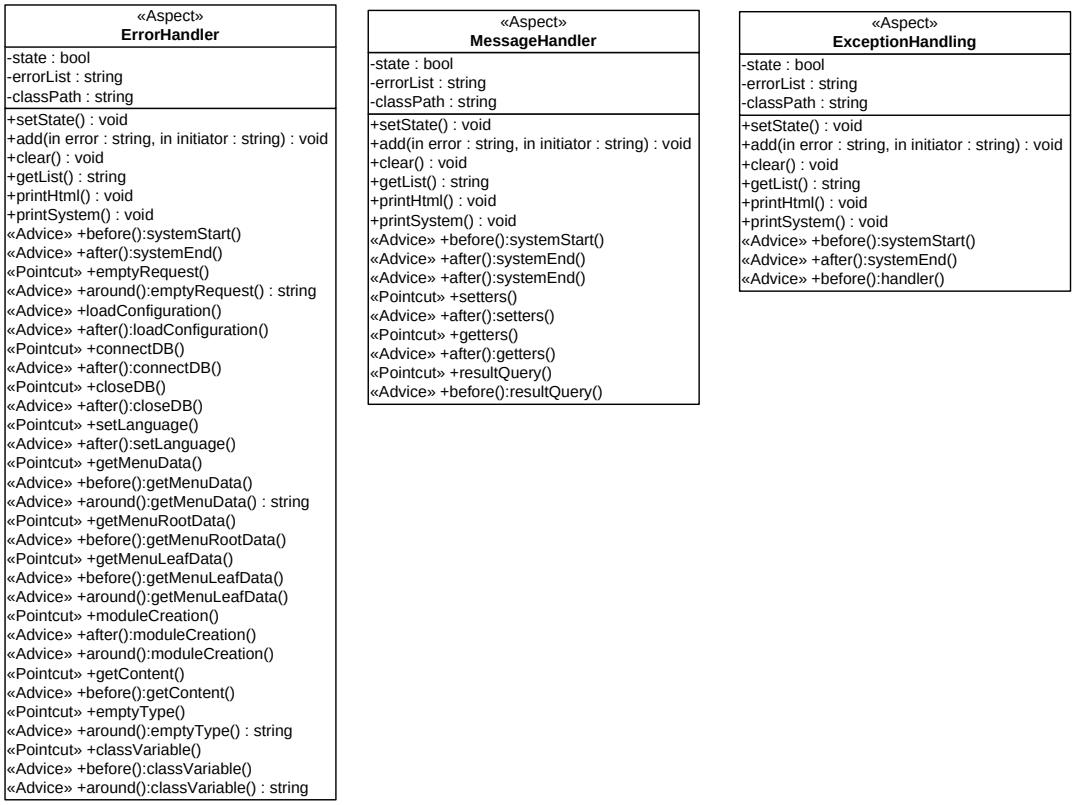


Fig. 35 SimpleW Logging concern after first development iteration

Although all the required functionality is covered by this particular design, the design itself contains some negative issues: it is hard to maintain, it contains repeating functionality and it provides no hot spots. The repeating code should be combined into one abstract aspect. Pointcuts and advice should be handled by several different aspects for better maintenance. At least three different hot spots are required – logging should be able to provide ways for customizing: logging (loggers), logging behaviour (handlers) and logging writing (writers).

After performing a set of optimizations and introducing a Template method design pattern the following Logger design has been elaborated (Fig. 36).

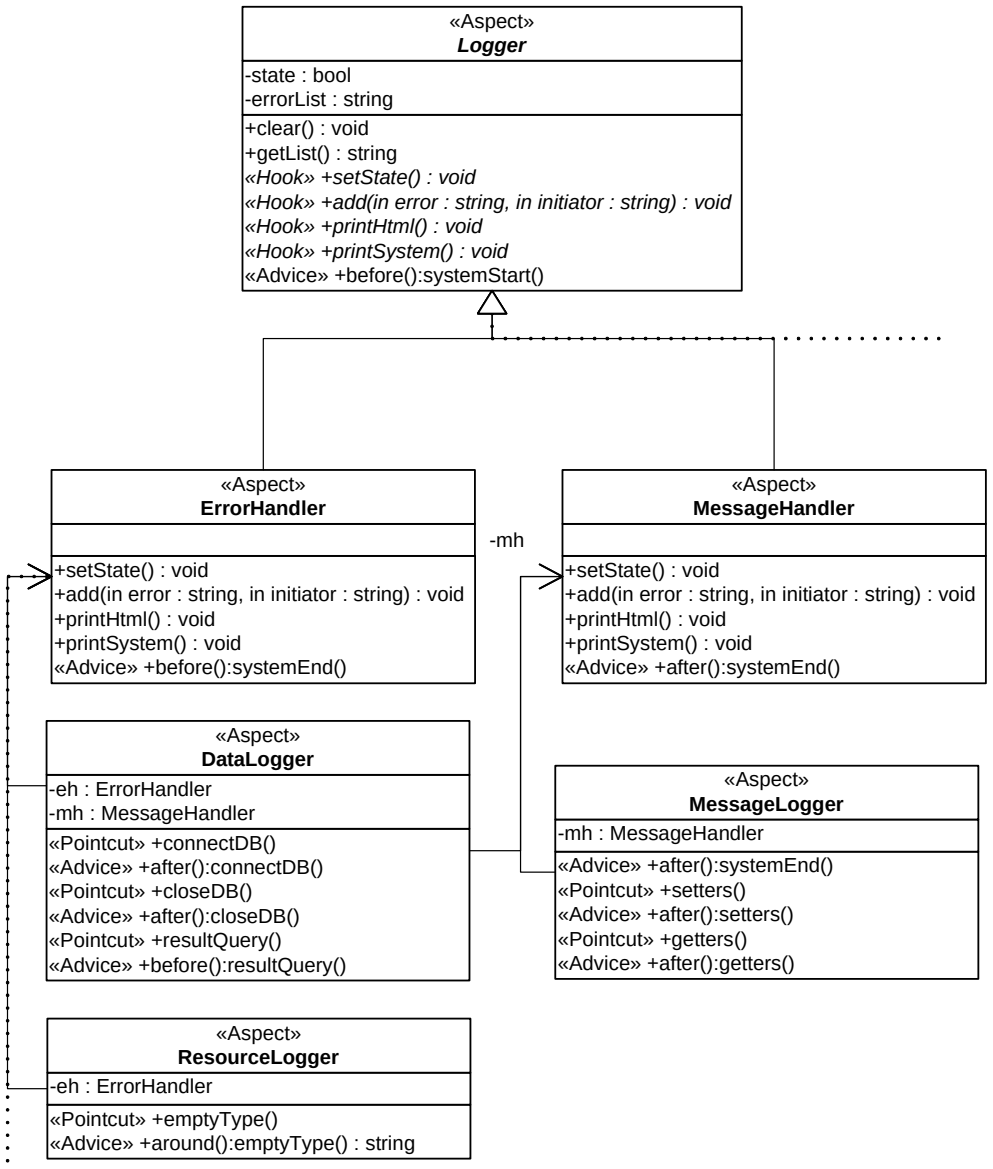


Fig. 36 SimpleW Logging concern after second development iteration (full version can be found in APPENDIX D)

Such design provides customization of handlers and loggers. However, there is only one hot spot designed for this aim. It consists of four hook methods and allows customization of handlers. Loggers are introduced without using inheritance based hot spots. Another problem, not solved by this particular design, is customization of logging writers. Writers are hardwired inside hook methods *printHtml* and *printSystem*. The only way to introduce additional writer behaviour is to provide a new method. Thus, it is not acceptable.

In order to solve logger customization problem (situation is similar to the one with breadcrumbs navigation) GoF_{AO} Decorator design pattern has been applied. The writer customization requires reconfiguration of the writing behaviour so that it could be performed by classes instead of methods. To solve this design problem GoF_{AO} Command design pattern has been applied. The resulted design completely satisfies all the requirements (Fig. 37).

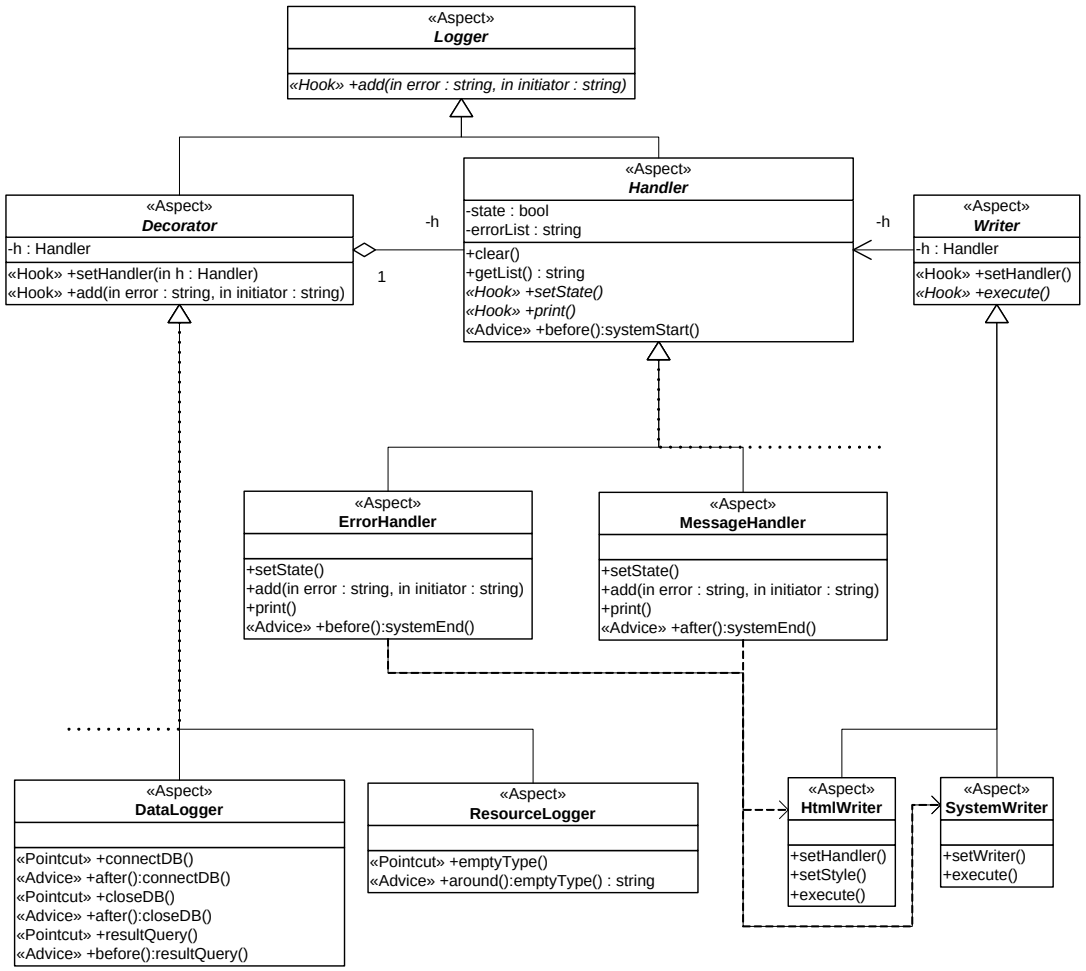


Fig. 37 SimpleW Logging concern after third development iteration (full version can be found in APPENDIX E)

Eventually, logging concern can be customized by three hot spots: *Decorator*, *Handler* and *Writer* aspects. *Decorator* aspect contains two hook methods and can be customized by introducing additional loggers. *Handler* aspect has three hook methods (including the inherited *add* method) and can be customized by introducing additional handlers. *Writer* aspect behaviour is no longer

hardwired inside the *Handler* aspect and can be customized by introducing new writers using two hook methods.

4.4.4.Measurements and Data Analysis

SimpleW framework has been developed by performing three development iterations. During all of them quantitative data on the structure of code and on performance of applications produced (using AO SimpleW framework) have been collected. All the data is presented bellow (Fig. 38, Fig. 39) by the corresponding bar graphs. Every graph contains three bars: "A1" bar corresponds to the implementation after the first development iteration, "A2" bar – to AO implementation after the second development iteration, "A2" bar – to AO implementation after the third development iteration. The measurements in Fig. 38 are presented as quantities and in Fig. 39a, Fig. 39b and Fig. 39c as milliseconds. Data about the structure of code (Fig. 38) demonstrate that parameters have been influenced by minor changes, except two of them. Numbers of code lines, data members and references, methods and advice, external calls and pointcuts remain almost the same. The second development iteration produced a slightly greater amount of data members and external calls. On the other hand, the same development iteration produced some fewer amount of methods and advice. However, the second and third development iteration increases the number of Aspects and Classes, as well as Hook methods. The greater number of entities (i.e. classes and aspects) is caused by finer granularity of the implementation code. It is useful because entities are becoming smaller and less complex. The increase of Hook methods indicates that customization was extended by providing additional AO hot spots.

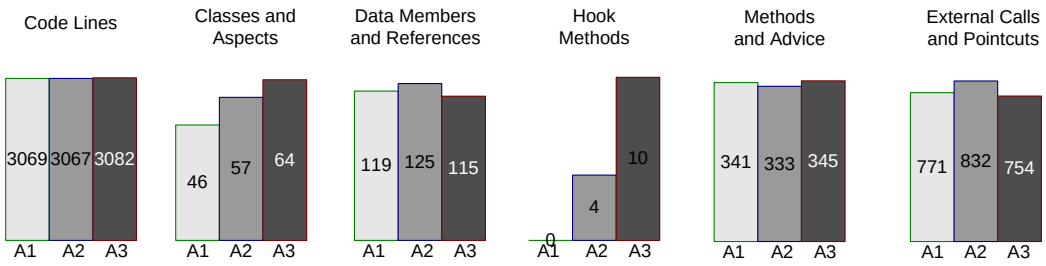


Fig. 38 static quantitative data of measurements (SimpleW framework)

An application has been produced after every design iteration and three tests have been performed for each application. All three tests have been performed by executing different parts of the application. In the first test (Fig. 39a), the representative part of the web application, in the second (Fig. 39b) – user registration part of the web application and in the third (Fig. 39c) – administration part of the web application have been executed. Each test has been executed 50 times. All executions have been performed using the same configuration and data representation of the application.

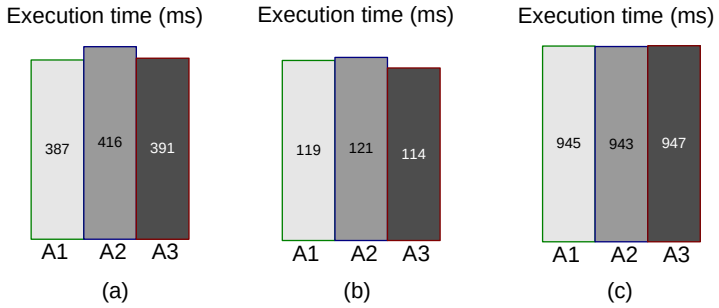


Fig. 39 testing data of measurements (SimpleW framework)

After the second development iteration in some parts (a and b) of the web application a tiny decrease of the performance can be observed. However, in contrary to the first two parts, administration part (c) test demonstrates a tiny increase of the performance. The differences of data measurements are insignificant and do not follow any pattern. Consequently, tiny differences of data can be stated as a result of possible biases.

4.5. Hypotheses evaluation

The hypothesis that AO GoF 20 design patterns decrease the complexity of the code has been confirmed by both, design iterations of case study 2 and by all design iterations of case study 3. The hypothesis has been fully confirmed by all three case studies that the AO GoF 20 design patterns allow to design abstract aspects which facilitate the extension of framework with new hot spots. The hypothesis that AO GoF20 design patterns reduce crosscutting in the framework has been confirmed by case study 1 and case study 2 because all logging implementation code has been successfully collected in logging aspects. The hypothesis that GoF_{AO} design patterns have no particular impact

on the overall run-time performance of the applications has been rejected in the case study 2 and partially confirmed in case study 3. After the first design iteration of case study 2 the average of 31 % loss of the performance in both execution modes has been observed. However, the average of performance loss in the second iteration of case study 2 in both execution modes is approximately 0.8 %.

4.6. Summary

The critical case research has demonstrated that design patterns solving similar design problems in both, AO and OO paradigms, could be used to deal with crosscutting and to design customizable aspects in frameworks. It has been validated that the usage of GoF_{AO} design patterns reduces crosscutting in AO domain frameworks. The investigated case of Factory Method design pattern shows that even creational design patterns can be applied for this purpose. It promotes the elimination of crosscutting behaviour and localization of scattered implementations. Moreover, this crosscutting behaviour can be designed as a reusable hot spot in a framework and customized in a framework application. The purpose of Factory Method design pattern in AOP is slightly changed comparing to OOP. Instead of creating factories it only passes reference to the necessary aspect.

The remaining of the case studies has demonstrated that other AO GoF 20 design patterns can be used to design AO frameworks. During the second case study research two AO versions of OO SimJ framework have been designed and detailed evaluation of applied design patterns have been presented. The case study has confirmed the hypothesis that the usage of GoF_{AO} design patterns (next to 23 GoF design patterns) improves the efficiency of domain frameworks designs. It decreases code complexity, eliminates crosscutting and allows designing additional AO hot spots in the framework. Performance tests have revealed that GoF_{AO} design patterns in some cases may reduce the overall run-time performance of the applications. Besides, it depends on the optimization of design and the more design refinement steps are performed the

better performance can be achieved. It also depends on the particular design patterns that are applied and on the skills of designers – that is, on how proper design patterns he/she is able to choose. Of course, it is a kind of art.

During the third case study research three AO versions of OO SimpleW framework have been developed from scratch and detailed evaluation of applied design patterns has been presented. It has been proven that the usage of GoF_{AO} design patterns allows designing a new class of hot spots in white-box AO domain frameworks (namely, hot spots represented by abstract aspects). The case studies have also revealed that in some cases the possible loss of performance of applications designed using GoF_{AO} design patterns can be expected.

In general, the AO GoF 20 design patterns are insufficient to optimize the design and additional AO design patterns are still necessary, particularly, pointcut and advice related design patterns are required. Patterns proposed in (Hanenberg et al., 2003; Laddad, 2003; Miles, 2004; Bynens, Joosen, 2009) should be used in compositions with AO GoF 20 design patterns. The main conclusions of the chapter are as follows:

1. In the aspect-oriented programming languages design patterns solving paradigm-independent design problems can be implemented using only AOP constructs. It follows that aspects can be used as collaborative entities, which means that it is possible to establish dependencies and associations among aspects and to create their hierarchies. However, in some cases, it can result in the crosscutting among aspects. It can be expected that the crosscutting can be eliminated by using higher level aspects or that it is possible to avoid such crosscutting by using some anti-patterns.
2. The execution of one critical and two demonstrative case studies has demonstrated that 20 GoF_{AO} design patterns can be used to design aspect-oriented frameworks.
3. The case studies have confirmed that 20 GoF_{AO} design patterns decrease code complexity, eliminate crosscutting, and allow designing additional

AO hot spots in frameworks. Performance tests have revealed that in some cases the loss of performance can be expected. However, it depends on the particular design pattern that is applied and on the skills of designers – that is, on how proper design patterns he/she is able to chose. Besides, it depends on the optimization of design and the more design refinement steps are performed, the better performance can be achieved.

4. In general, the 20 GoF_{AO} design patterns and patterns proposed in (Hanenberg et al., 2003; Laddad, 2003; Miles, 2004; Bynens, Joosen, 2009) are insufficient to optimize the design and additional AO design patterns are still necessary, particularly, pointcut and advice related design patterns are required.
5. Aspect-oriented framework design from scratch case study provides constructive research steps that have been proven to be used as a basic development steps to develop aspect-oriented frameworks.

The results of this chapter have been published in (Vaira, Čaplinskas, 2011; Vaira, Čaplinskas, 2011a).

Chapter 5

Discussion of Issues and Limitations

There are several debatable issues that must be discussed. The first one is the use of aspects as collaborative entities. The designs that include abstract aspect hierarchies, hold references and invoke calls to other aspects, help to create reusable and flexible implementation structures. These are the main features used to create collaborations of classes in OOP. However, such structures also increase the tangling of the implementation code which is an issue that AOP has to deal with. It is not always clear what the constraints of collaborations in aspects are and when a threat of creating too complex designs of aspects appears. Collaborations mean the capability to organize aspects into hierarchical structures and to model dependencies and associations among them. It is assumed that collaborations of aspects are beneficial unless the collaborations of aspects overstep the boundaries of related concern (i.e. introduces crosscutting between aspects). Such assumption is confirmed by the results of the performed case studies. There is no evidence that collaborations of aspects, if designed carefully, can in some way reduce the overall efficiency of the applications. The expected increase of efficiency is observed by analyzing the measured data.

The Singleton nature of aspects is the second issue. Although aspects in AspectJ are by default singletons, in special cases aspects can be also instantiated per object or per control flow. From this perspective it is still questionable whether aspects should be treated as singletons or not. For example, in AspectJ language direct instantiations of aspects are forbidden. Aspects can be globally referenced only using static method *aspectOf*. Such

referencing of aspects is different from referencing of objects. Objects require instantiation in order to be referenced. Another problem is that if it were allowed to create several instances of the same aspect at a time, the behaviour advised by aspects might repeat several times or act in other unexpected ways. As a result, there may be difficulties related to aspect instantiation control. This is the main reason why the Singleton nature of aspects is suggested to be followed and to be treated per object and per control flow aspects as special cases of singletons.

Although the results of the thesis are comprehensive and applicable to real world software design, some limitations can be observed:

- The case studies have been performed to design white-box domain application frameworks only. No black-box frameworks have been investigated.
- All implementations of aspect-oriented designs have been performed using AspectJ programming language. Despite the fact that AspectJ is first and the most popular AOP language, implementations in other languages are required.
- The use of GoF design patterns can also be considered as a limitation. The main reason why these patterns have been chosen is that they are widely-used and have been well investigated by other researchers. Therefore, the results of such research can be easily compared with the results of other researches. However, the redesign of other design patterns to pure aspect-oriented patterns must be performed and their applicability to design aspect-oriented applications should be investigated.
- There are a number of other existing metrics that have not been used in the present thesis.

5.1. Open problems

Not all of the 20 GoF_{AO} design patterns have been investigated by applying them to concrete context. Five GoF_{AO} design patterns have been stated as

exposing some limited applicability. It means that they are in some way more constrained than OO implementation because it is impossible to work with several instances of an aspect at the same time. Such design patterns have to be investigated in more details. The intent of such design patterns should be revised and some other changes in the pattern structure may be required. Only then it may serve as an acceptable solution to the aspect-oriented design. Otherwise they have to be removed from the initial list of the successfully transformed aspect-oriented design patterns if there is no verification of their applicability.

Another open problem is how this technique could be applied to other software engineering paradigms and this requires an additional set of researches to be executed.

The following open problems have to be investigated as well:

- which other OO design patterns beside the surveyed 23 GoF design patterns solve paradigm-independent design problems at least in respect of OO and AO paradigms;
- which design patterns can be developed and used to solve AOP-specific problems;
- in which way could pure AO design patterns be incorporated into the aspect-oriented design methodology.

Conclusions

1. Aspect-oriented design patterns, developed using direct code rewriting techniques and represented using constructs, provided by both, aspect-oriented and object-oriented paradigms, are not sufficient for complete separation of concerns, do not allow to implement hot spots of aspect-oriented domain frameworks as abstract aspects and are not universal enough, therefore can be applied only to a specific application context.
2. The 20 out of 23 object-oriented design patterns (GoF patterns) proposed by Gamma et al. (Gamma et al., 1994) solve the design problems that are also relevant in the context of aspect-oriented software engineering paradigm. Constructs provided by aspect-oriented paradigm are sufficient to implement these design patterns – that is, to implement them as pure aspect-oriented design patterns without using any specific object-oriented constructs such as classes and objects. It means that aspects can be used as collaborative entities, making it possible and reasonable to create hierarchies of aspects and establish dependencies and associations among aspects.
3. The usage of pure aspect-oriented designs patterns reduces crosscutting in aspect-oriented domain frameworks and allows the designing of a new kind of hot spots, namely, the hot spots represented by abstract aspects in white-box AO domain frameworks.
4. The case studies have confirmed that 20 pure aspect-oriented design patterns decreases code complexity, eliminates crosscutting and allows designing additional AO hot spots in frameworks. Performance tests have revealed that in some cases the loss of performance is expected. However, it depends on the particular design pattern that is applied and

on designer skills – that is, how he/she is able to choose proper design patterns.

5. The case studies and the analysis of aspect-oriented domain framework construction process demonstrate that the following construction steps are necessary in order to achieve successful design results:
    a. identify aspects representing modules that have to be designed in a crosscutting manner by analyzing requirement specification;
    b. decide which hot spots have to be designed using objects and which – using aspects; examine what design problems have to be solved and determine the design patterns that can be applied for this purpose;
    c. design and implement the required aspects and objects;
    d. prepare necessary test cases; check whether the resulted design is already acceptable; improve the design and go back to step *c* if the refactoring of code is still required.

References

Adair, D. (1995). *Building Object-Oriented Frameworks*. AIXpert. Feb. 1995

Aksit, M.; Bergmans, L. and Vural, S. (1992). An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In *Proceedings of O. Lehrman Madsen, editor, European Conference on Object-Oriented Programming (ECOOP), Utrecht, The Netherlands, June/July, 1992*. Springer Verlag Lecture Notes in Computer Science Vol. 615, 72-396.

Alexander, C.; Ishikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I. and Angel, S. (1977). A Pattern Language. Oxford University Press, New York.

Ambler, A. L.; Burnett, M. M. and Zimmerman, B. A. (1992). Operational Versus Definitional: A Perspective on Programming Paradigms. Computer, September 1992, 28-43. Accessible at ftp://ftp.engr.orst.edu/pub/burnett/Computer-paradigms-1992.pdf

Appleton, B. (1997). Patterns and software: Essential concepts and terminology. [Accessed 2011-09-15] Available at http://www.cmcrossroads.com/bradapp/docs/patterns-intro.pdf

Arnold, K.; Gosling, J. and Holmes, D. (2005). *Java™ Programming Language, 4th Edition*. Prentice Hall, August 27, 2005.

Arnout, K. and Meyer, B. (2006). *Pattern componentization: the factory example*. Innovations in Systems and Software Engineering, 6 May 2006, 2:65–79.

Arpaia, P.; Bernardi, M.L.; Lucca, G. Di; Inglese, V. and Spiezia, G. (2008). Aspect Oriented-based Software Synchronization in Automatic Measurement Systems. In *Proceedings of Instrumentation and*

Measurement Technology Conference, IMTC 2008, IEEE, 1718 – 1721, 12-15 May 2008.

Benbasat, I.; Goldstein, D. K. and Mead, M. (1987). The Case Research Strategy in Studies of Information Systems. *MIS Quarterly*, Vol. 11, No. 3 (Sep., 1987), pp. 369-386.

Bernardi, M.L. and Lucca, G.A. Di (2005). Improving Design Pattern Quality Using Aspect Orientation. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05), 24-25 Sept. 2005*, IEEE Computer Society, 206 – 218.

Booch, G.; Jacobson, I. and Rumbaugh J. (2000). OMG Unified Modeling Language Specification 1.3. [Accessed 2011-09-20] Available at http://www.omg.org/spec/UML/

Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerland P. and Stal M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

M. Bynens, B. Lagaisse, W. Joosen, and E. Truyen (2007). The elementary pointcut pattern. In *Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development, New York, NY, USA, 2007*. ACM, Article No 2.

Bynens, M. and Joosen W. (2009).Towards a Pattern Language for Aspect-Based Design. In Proceedings of the 1st workshop on Linking aspect technology and evolution (PLATE '09), Charlottesville, Virginia, USA, March 2 - 6, 2009. ACM, 13-15.

Cacho, N.; Figueiredo, E.; Sant´Anna, C.; Garcia, A.; Batista, T. and Lucena, C. (2005). *Aspect-oriented Composition of Design Patterns: a Quantitative Assessment*. Monografias em Ciência da Computação - No. 34/05. Pontifícia Universidade Católica do Rio de Janeiro, Brasil.

Crnkovic G. D. (2010). *Constructive Research and Info-Computational Knowledge Generation*. Model-based reasoning in science and technology, Studies in Computational Intelligence, 2010, Volume 314/2010, 359-380.

Cunha, C.A.; Sobral, J.L. and Monteiro, M.P. (2006). Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms. In R.E. Filman (ed.). *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*. ACM, 134-145.

Czarnecki, K. and Eisenecker, U.W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison Wesley.

Dantas, D. S.; Walker, D.; Washburn, G. and Weirich, S. (2008). *AspectML: A Polymorphic Aspect-oriented Functional Programming Language*. ACM Transactions on Programming Languages and Systems, 30(3), (May 2008), 71-130.

Denier, S.; Albin-Amiot, H. and Cointe, P. (2005). Expression and Composition of Design Patterns with Aspects. In *Proceedings of the 2nd French Workshop on AspectOriented Software Development JFDLPA 2005*. Hermès, 19-34

Denier, S.; Albin-Amiot, H. and Cointe, P. (2006). Expression and Composition of Design Patterns with AspectJ. *L'Objet*, 12, 2-3, 41-61.

Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K. and Ossher, H. (2001). *Discussing aspects of AOP*. Communications of the ACM, 44(10):33-38, October 2001.

Fayad, M. E. and Schmidt, D. C. (1997). *Object-Oriented Application Frameworks*. Communications of the ACM, Vol. 40, No. 10, pp. 32-38.

Fiege, L. ; Mühl, G. and Gärtner, F. C. (2002). *Modular event-based systems*. The Knowledge Engineering Review, Vol. 17:4, pp. 359–388. 2002, Cambridge University Press.

Filman, R.E. and Friedman, D.P. (2001). *Aspect-Oriented Programming is Quantification and Obliviousness*. Research Institute for Advanced Computer Science, RIACS Technical Report 01.12.

Fleury, M. and Reverbel, F. (2003). The JBoss extensible server. In Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03). Volume 2672 of Lecture Notes in Computer Science., Springer-Verlag (2003) 344–373.

Floyd, R. W. (1979). The Paradigms of Programming. Comm. ACM, Vol. 22, No. 8, Aug. 1979, pp. 455-460.

Flyvbjerg, B. (2004). Five misunderstandings about case-study research. In *C. Seale, G. Gobo, D. Silverman (eds.). Qualitative Research Practices.* London and Thousand Oaks, CA: Sage, 420-434.

Fowler, M. (2003). UML distilled: Brief Guide to the Standard Object Modeling Language (3rd Edition). Addison-Wesley Professional, September 25, 2003.

Froehlich, G.; Hoover, J.; Liu, L. and Sorenson, P. (1998). *Designing object-oriented frameworks*. CRC Handbook of Object Technology, CRC Press, pp. (25)1-21, 1998.

Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional;

Garcia, A. (2004). *From Objects to Agents: An Aspect-Oriented Approach*. Doctoral Thesis, Rio de Janeiro, Brazil, PUC-Rio, 2004.

Garcia, A.; Sant'Anna, C.; Figueiredo, E. and Kulesza, U. (2005). Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD'05), Chicago, USA, 14-18 March 2005*. ACM Press, 3-14.

Griswold, W. G.; Sullivan, K.; Song, Y. and Shonle, M. (2006). N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Softw*are, 23(1), 51–60.

Gosling, J.; Joy, B.; Steele, G. and Bracha, G. (2005). *Java™ Language Specification, The (3rd Edition)*. Addison Wesley, 2005.

Hachani, O. and Bardou, D. (2002). Using Aspect-Oriented Programming for Design Patterns Implementation. In Proceedings of 8th International Conference on OOIS 2002, Position paper at the Workshop on Reuse in Object-Oriented Information Systems Design. Montpellier, France - Sept. 2-5 2002.

Hachani, O. and Bardou, D. (2003). On Aspect-Oriented Technology and Object-Oriented Design Patterns. In Proceedings of European Conference on Object Oriented Programming ECOOP 2003, Position paper at the workshop on Analysis of Aspect-Oriended Software. Darmstadt, Germany, 2003.

Hanenberg, S. and Costanza, P. (2002).Connecting Aspects in AspectJ: Strategies vs. Patterns. In Y. Coady (ed.). First Workshop on Aspects, Components, and Patterns for Infrastructure Software, AOSD, Enschede, The Netherlands, April 22-26, 2002. TR-2002-12. The Department of Computer Science, University of British Columbia, Vancouver, B.C., 40-45.

Hanenberg, S. and Schmidmeier, A. (2003). Idioms for building software frameworks in AspectJ. In Y. Coady, E. Eide, D. H. Lorenz (Eds.) *Proceedings of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Boston, Massachusetts, 2003*. NU-CCIS-03-03. College of Computer and Information Science, Northeastern University, Boston, Massachusetts, pp. 55-60.

Hanenberg, S.; Unland, R. and Schmidmeier, A. (2003). AspectJ Idioms for Aspect-Oriented Software Construction. In the *Proceedings of 8th*

European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany, 25th–29th June, 2003. pp. 617–644.

Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, ACM Press, 161-173.

Harrison, W. and Ossher, H. (1993). Subject-oriented programming (a critique of pure objects). In *Proceedings of Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 411- 428, 1993.

Hart, C. (1998). Doing a literature review: Releasing the social science research imagination. London, SAGE Publications.

Hejlsberg, A.; Wiltamuth, S.; Golde, P. (2003). *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

Hirschfeld, R.; Lämmel, R. and Wagner, M. (2003). Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration. In Proceedings of the 3rd German Workshop on Aspect-Oriented Software Development (AOSD-GI 2003), p. 25-32.

Johnson, R. E. and Foote, B. (1988). *Designing Reusable Classes*. Journal of Object-Oriented Programming, June/July 1988, 1(2): 22-35. [Accessed 2011-04-20] Available at http://www.laputan.org/drc/drc.html

Johnson, R. E. (1997). *Frameworks = (components + patterns)*. Communications of the ACM, October 1997, 40(10): 39-42.

Kaisler, S. H. (2005). *Software paradigms*. John Wiley & Sons, Inc.

Kiczales, G.; Rivieres, J. Des and Bobrow, D.G. (1991). *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J. M. and Irwin, J. (1997). Aspect oriented programming. In

Proceedings of European Conference on Object Oriented Programming, ECOOP, 1997, vol. 1241, pp. 220–242.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. and Griswold, W. G. (2001). *Getting started with AspectJ*. Communication of the ACM, October 2001, 44(10): 59-65.

Kulesza, U.; Alves, V.; Garcia, A.; Lucena, C. J. P. de and Borba, P. (2006). Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In *Proceedings of Intl Conference on Software Reuse (ICSR)*, Torino, Italy, pp. 231-245, 2006.

Laddad, R. (2003). AspectJ in Action: practical aspect-oriented programming. Manning Publications Co.

Laddad, R. (2010). AspectJ in Action, Second Edition: enterprise AOP with spring applications. Manning Publications Co.

Lagaisse, B. and Joosen, W. (2006). Decomposition into elementary pointcuts: A design principle for improved aspect reusability. In *the Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT) affiliated with AOSD 2006, March 21, 2006*. Bonn, Germany, 64 - 69.

Lämmel, R. and Visser J. (2002). Design patterns for functional strategic programming. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, Pittsburgh, Pennsylvania, USA, 2002, ACM 2002, pp. 1-14.

Laurence, S. and Margolis, E. (2003). *Concepts and conceptual analysis*. Philosophy and Phenomenological Research, 67: 253-282.

Lieberherr, K.J.; Silva-Lepe, I. and Xiao, C. (1994). Adaptive object-oriented programming using graph-based customization. Communications of the ACM, 37(5):94:101, May 1994.

Lopes, C.V. (2005). Aspect-Oriented Programming: A Historical Perspective *(What's in a Name?). In Aspect-Oriented Software Development, Add*ison-Wesley, 97–122.

Lorenz, D.H. (1998). Visitor Beans: An Aspect-Oriented Pattern. In *Proceedings of the ECOOP'98 Workshop on Aspect-Oriented Programming*, pp. 431-432.

Lukka, K. (2003). The constructive research approach. In: *L. Ojala, O-P. Hilmola (eds.) Case study research in logistics*. Publications of the Turku School of Economics and Business Administration, Series B 1: 2003, p. 83-101.

MacDonald, S.; Szafron, D.; Schaeffer, J.; Anvik, J.; Bromling, S. and Tan, K. (2002). Generative design patterns. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*. IEEE Computer Society, 23-34.

Maioriello, J. (2002). What Are Design Patterns and Do I Need Them? Online publication, *developer.com*, October 2002, QuinStreet Inc. [Accessed 2011-09-20] Available at http://www.developer.com/design/article.php/1474561/What-Are-Design-Patterns-and-Do-I-Need-Them.htm

Martin, R. (2000). *Design Principles and Design Patterns*. [Accessed 2011-10-15] Available at http://www.objectmentor.com/resources/articles/Principles-and-Patterns

Menkyna, R.; Vranić, V. and Polášek, I. (2010). Composition and Categorization of Aspect-Oriented Design Patterns. In Proceedings of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010, January 2010, Herľany, Slovakia, IEEE, 129-134.

Merriam-Webster Online Dictionary (2011). Definition of a concept – paradigm. [Accessed 2011-11-24] Available at http://www.merriam-webster.com/dictionary/paradigm

Meslati, D. (2009). *On ASPECTJ and Composition Filters: A Mapping of Concepts*. Informatica, 2009, Vol. 20, No. 4, 555–578, 2009 Institute of Mathematics and Informatics, Vilnius.

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

Monteiro, M. P. (2006). Using Design Patterns as Indicators of Refactoring Opportunities (to Aspects). In *Proceedings of AOSD 2006 workshop on Linking Aspect Technology and Evolution (LATEr)*, Bonn, Germany, 20 March 2006.

Miller, S. K. (2001). *Aspect-Oriented Programming Takes Aim at Software Complexity*. IEEE Computer, vol. 34, no. 4, pp. 18-21 2001.

Noble, J.; Schmidmeier, A.; Pearce, D.J. and Black, A.P. (2007). Patterns of Aspect-Oriented Design. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLOP), July 2007, Bavaria*. Hillside Publishers, 769-796.

Noda, N. and Kishi, T. (2001). Implementing Design Patterns Using Advanced Separation of Concerns. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, FL, USA, 2001.

Nordberg, M.E. (2001). Aspect-Oriented Dependency Inversion. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, FL, USA, 2001.

Nordberg, M.E. (2001a). Aspect-Oriented Indirection – Beyond Object-Oriented Design Patterns. In *Proceedings of OOPSLA 2001, Position paper at workshop "Beyond Design: Patterns (mis)used"*, 2001.

Papapetrou, O. and Papadopoulos, G. A. (2004). Aspect Oriented Programming for a component based real life application: A case study. In *Proc. ACM Symposium on Applied Computing*, Nicosia, Cyprus, pages 1554 – 1558, 2004.

Piveta, E. K. and Zancanella, L. C. (2003). Observer Pattern using Aspect-Oriented Programming. In *Proceedings of the 3rd Latin American Conference on Pattern Languages of Programming*, Porto de Galinhas, PE, Brazil, August 2003.

Ragin, C. C. (1992) '"Casing" and the process of social inquiry', In *Charles C. Ragin and Howard S. Becker (eds)*, *What is a Case? Exploring the Foundations of Social Inquiry*. Cambridge: Cambridge University Press, pp. 217–26. . Cambridge: Cambridge University Press, pp. 217–26.

Rausch, A.; Rumpe, B. and Hoogendoorn, L. (2003). Aspect-Oriented Framework Modeling. In *Proceedings of the 4th AOSD Modeling with UML Workshop*, UML Conference 2003, October 2003.

Rossum, G. van (1993). An Introduction to Python for UNIX/C Programmers. In *Proceedings of the NLUUG najaarsconferentie (1993)*. Dutch Unix user group.

Runeson, P. and Höst, M. (2009). *Guidelines for conducting and reporting case study research in software engineering* .Empirical Software Engineering, Volume 14 Issue 2, April 2009, 14:131–164.

Sant'Anna, C.; Garcia, A.; Chavez, C.; Lucena, C. and Staa, A. (2003). On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings of Brazilian Symposium on Software Engineering SBES'03), Manaus, Brazil*, 19-34, 2003.

Santos, A. L.; Lopes, A. and Koskimies, K. (2007). Framework specialization aspects. In *Proceedings of AOSD '07 the 6th international conference on Aspect-oriented software development*, ACM New York, NY, USA 2007, 14 - 24.

Shalloway, A. and Trott, J.R. (2001). *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Software Patterns Series. Addison-Wesley Professional.

Schmidmeier, A.; Hanenberg, S. and Unland, R. (2003). Implementing Known Concepts in AspectJ. In B. Bachmendo, S. Hanenberg, S. Herrmann, G. Kniesel (eds.). *Proceedings of the third German Workshop on Aspect-Oriented Software Development*. University of Duisburg-Essen Institute for Computer Science and Business Information Systems (ICB), 65-70.

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, July 2004*.

Tešanović, A. (2004). What is a pattern? Course note, at Linköping University, Sweden. [Accessed 2011-10-10] Available at http://www.idi.ntnu.no/emner/dt8100/papers2005/P-a10-tesanovic04.pdf

Tonella, P. and Antoniol, G. (1999). Object Oriented Design Pattern Inference. In *Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99*, Oxford, UK, IEEE Computer Society, 230 – 238.

Vranić, V. (2001). AspectJ Paradigm Model: A Basis for Multi-Paradigm Design for AspectJ, In Jan Bosch, editor, Proc. of the Third International Conference on Generative and Component-Based Software Engineering (GCSE 2001), LNCS 2186, Erfurt, Germany, September 2001, pp. 48-57, Springer.

Wampler, D. and Clark, T. (2010). Guest Editors' Introduction: Multiparadigm Programming. Software, IEEE, 27(5), 20 – 24, Sept.-Oct. 2010.

Zimmer, W. (1995). Relationships Between Design Patterns. In J. O. Coplien and D. C. Schmidt (eds.) Pattern Languages of Program Design. Addison-Wesley, 1995, pp. 345- 364

List of Publications

Vaira, Ž. and Čaplinskas, A. (2011). Case Study Towards Implementation of Pure Aspect-oriented Factory Method Design Pattern. In *Proceedings of 3rd International Conference on Pervasive Patterns and Applications, PATTERNS 2011, September 25-30, 2011 - Rome, Italy*

Vaira, Ž. and Čaplinskas, A. (2011a). *Application of pure aspect-oriented design patterns in the development of AO frameworks: A case study*. Information sciences, 2011-56, pp. 146–155.

Vaira, Ž. and Čaplinskas, A. (2011b). *Paradigm-independent design problems, GoF 23 design patterns and aspect design*. Informatica, 22(2), pp. 289–317.

Vaira, Ž. and Čaplinskas, A. (2009). Compositional aspect-oriented design pattern properties. In *Proceedings of 50th conference of Lithuanian union of mathematicians*, 123-453, 2009.

Vaira, Ž. (2009). Aspect-oriented software design method. In *Proceedings of 12th Student Scientific Society conference "Fundamental Research and Innovation in Science Integration"*. Klaipeda University Faculty of Natural Science and Mathematics, 2009, Klaipeda, Lithuania.

APPENDICES

APPENDIX A AspectJ language preliminaries

The join points in the base object-oriented program could be defined using one of the following items: method execution or call, constructor execution or call, field access, exception processing, class initialization, object initialization and advice execution. After successful identification of the necessary join points one can start defining aspects. Aspect structure can be divided in to 3 different parts: inter-type declarations, pointcut and advice. Inter-type declarations are made by aspects for the definition of interface, class, or aspect types. They consist of a member or method introductions, type-hierarchy modifications, and are used to implement the so called static crosscutting. Static crosscutting is not directly affected by pointcuts and advice. Pointcuts and advice define dynamic crosscutting of the system. Join points are defined by using pointcut. The functionality that should be performed at the join point is defined by using advice.

Join point in the system is understood as a concrete place in a running system. The pointcut itself defines a set of several concrete join points in this system. Aspect can include one or several pointcuts. Pointcuts are defined using the syntax that can be demonstrated by an example (Example 5).

```
1  protected pointcut pointcutName(Context c):
2  call(public TypeName ClassName.operation())&&this(c);
```

Example 5 General pointcut syntax

Pointcut structure mainly consists of: pointcut name (*pointcutName*), context data (*Context c*), pointcut type (*call*), and expression of the join point (*ClassName.operation*). The current example (Example 1) defines join points in the system where a defined operation is called. A star (*) could be placed instead of pointcut context and pointcut expression in order to make the pointcut more abstract.

The behaviour that must be injected at the join points is defined by the pointcut described in advice. In a simple form it is a concrete code that must be performed at the join point. The advice can be performed in exact places corresponding to a particular join point: before, around (instead) and after the join point. The advice inner content is very similar to the content of the method. In Example 6 a general example of advice is presented.

```
1  after(Context c) :pointcutName(c){
2         someObject.doSomethingWith(c);
3  }
```

Example 6 General advice syntax

The general syntax of advice includes: name of an advising pointcut (*pointcutName*), context of the pointcut (*Context c*) and the word denoting exact execution place (*after*). The code inside advice will be executed after every join point has been matched to the defined pattern of the pointcut.

Full aspect representation will be received by combining pointcut, advice and inter-type declarations. Aspect in its own way is the main entity of an aspect-oriented programming as in a similar way classes and objects are the main entities of an object-oriented programming. Aspects may contain data members and methods as classes do. Aspects can be also defined as abstract aspects which must be inherited by other aspects (differently from classes, concrete aspects can not be inherited). All the syntax for defining abstract aspects, data members, methods and accessibility is the same as the one used in Java language. Example 7 demonstrates the syntax of an aspect construct.

```
1  public aspect AspectName {
2
3         private TypeName className.type;
4
1         protected pointcut pointcutName(…): execute(…) && target(…);
2
3         after(…): pointcutName(…) {
4
5               …
6         }
7  }
```

Example 7 General aspect syntax

Although aspects are defined in a similar way as classes, aspects cannot be instantiated as objects. Instead, aspects can be referenced by using *aspectOf*

method. It is done in a very similar way as it is in a Singleton design pattern for objects.

Thus, it must be mentioned that aspects can be defined not only as singletons. By associating an aspect with some entity in a base program, that is, defining per object or per control flow aspect, one can have several instances of the same aspect at the same time.

APPENDIX B Remaining List of Transformed GoF_{AO} Design Patterns

To complete the list of successfully transformed GoF_{AO} design patterns the remaining of the design pattern structure diagrams and short descriptions are presented. The 4 GoF_{AO} – Adapter, Bridge, Factory Method and Chain of Responsibility – design patterns already have been explained in details.

Abstract Factory

The intent of the aspect-oriented Abstract Factory is to provide an interface for referencing several related or dependent aspects without specifying the name of concrete aspects (Fig. 40).

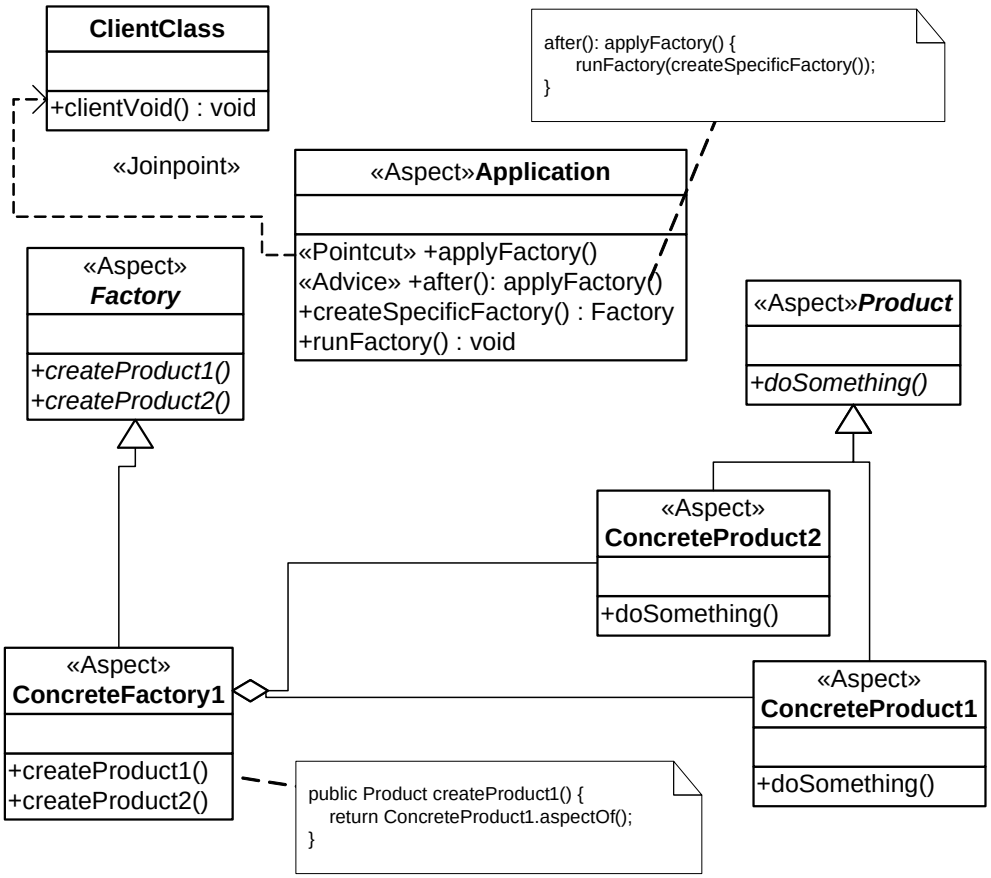


Fig. 40 Abstract Factory design pattern (AO solution)

The essential elements of this pattern are:

- *Factory* aspect, declares an interface for operations used to reference *Product* aspects,
- *ConcreteFactory1* aspect implements the operations for referencing *Product* aspects,
- *Product* aspect declares an interface for a concrete *Product* aspects,
- *ConcreteProduct1* and *ConcreteProduct2* aspects define a *Product* aspect for a corresponding *ConcreteFactory1* or *ConcreteFactory2* and implement the *Product* aspect interface.
- *Application* aspect provides operations for referencing specific factories and holds pointcut and advice for an invocation of a pattern,
- *Client*, the class that invokes the *applyFactory* pointcut.

Builder

The intent of the aspect-oriented Builder design pattern is to separate the construction of a complex aspect from its representation in a way that alteration of the same construction can still provide different representations (Fig. 41).

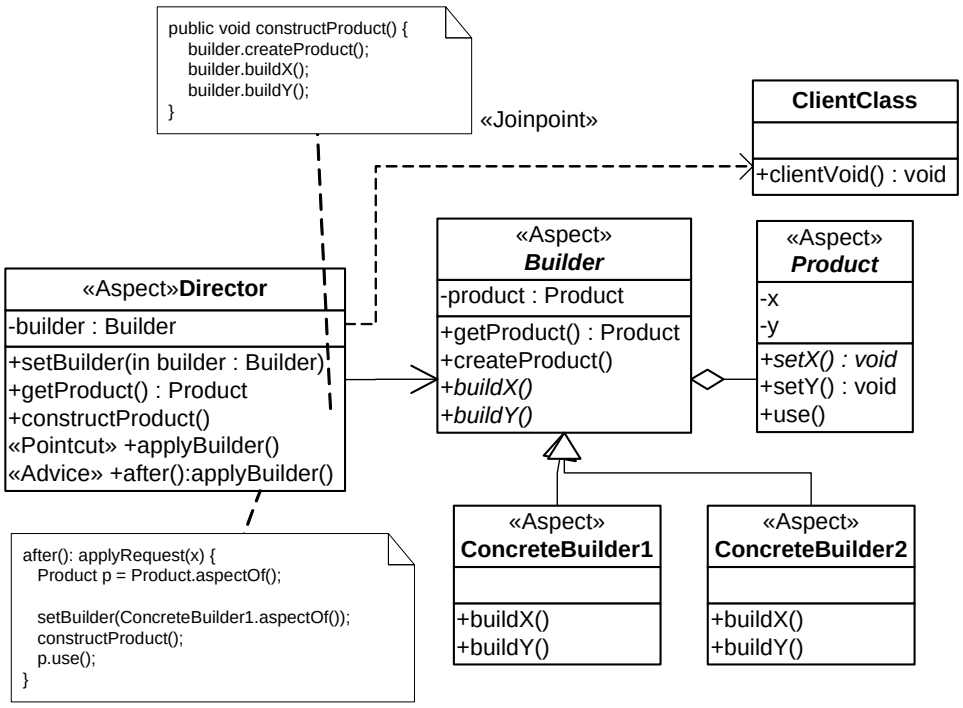


Fig. 41 Builder design pattern (AO solution)

- *Builder* aspect specifies an abstract interface for altering parts of a *Product* aspect,
- *ConcreteBuilder1* and *ConcreteBuilder2* aspects alter a parts of the *Product* aspect by implementing the *Builder* aspect interface, keep the reference to the altered *Product* aspect, and provide an interface for retrieving the *Product* aspect,
- *Product* aspect represents the complex Aspect which construction can be altered by a *Builder*,
- *Director* aspect alters complex *Product* structure using the *Builder* aspect interface and provides pointcut and the advice used for an invocation of a pattern,
- *Client* is the class that invokes the *applyBuilder* pointcut.

Command

The intent of the aspect-oriented Command design pattern is to encapsulate request as a reference to an aspect, thereby allowing to parameterize the invocations with different requests, queue or log requests, and support undoable operations (Fig. 42).

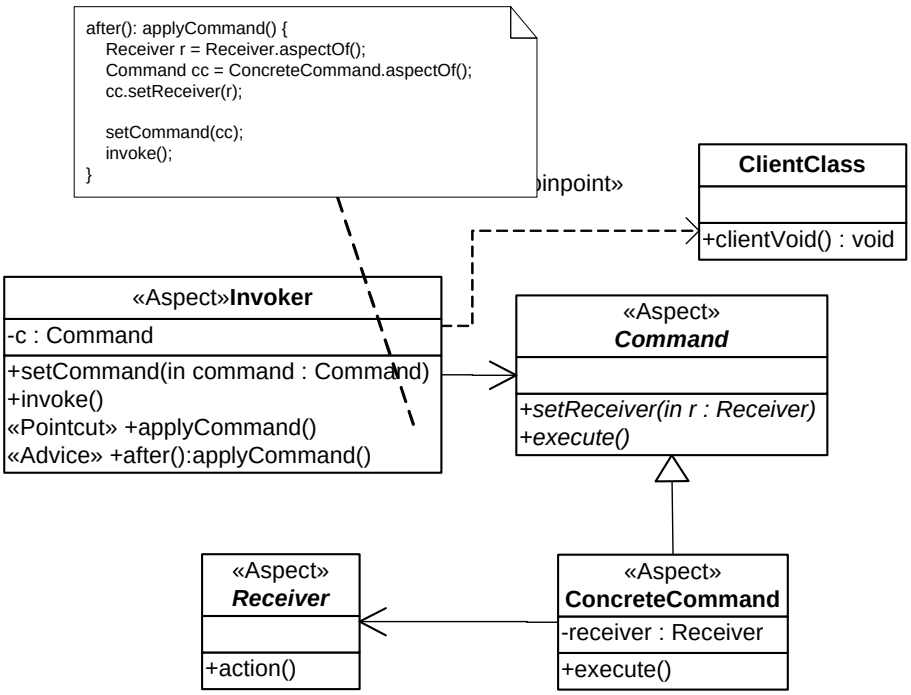


Fig. 42 Command design pattern (AO solution)

- *Command* aspect declares an interface for executing an operation,
- *ConcreteCommand* aspect defines a binding between a *Receiver* aspect and an action, implements *Execute* operation by invoking the operations on *Receiver* aspect,
- *Invoker* aspect asks the *Command* aspect to carry out the request,
- *Receiver* aspect performs the operations associated with carrying out a request and provides pointcut and the advice for an invocation of a pattern,
- *Client*, the class that invokes the *applyCommand* pointcut.

Decorator

The intent of the aspect-oriented Decorator design pattern is to attach additional functionality to an aspect without extending it (Fig. 43).

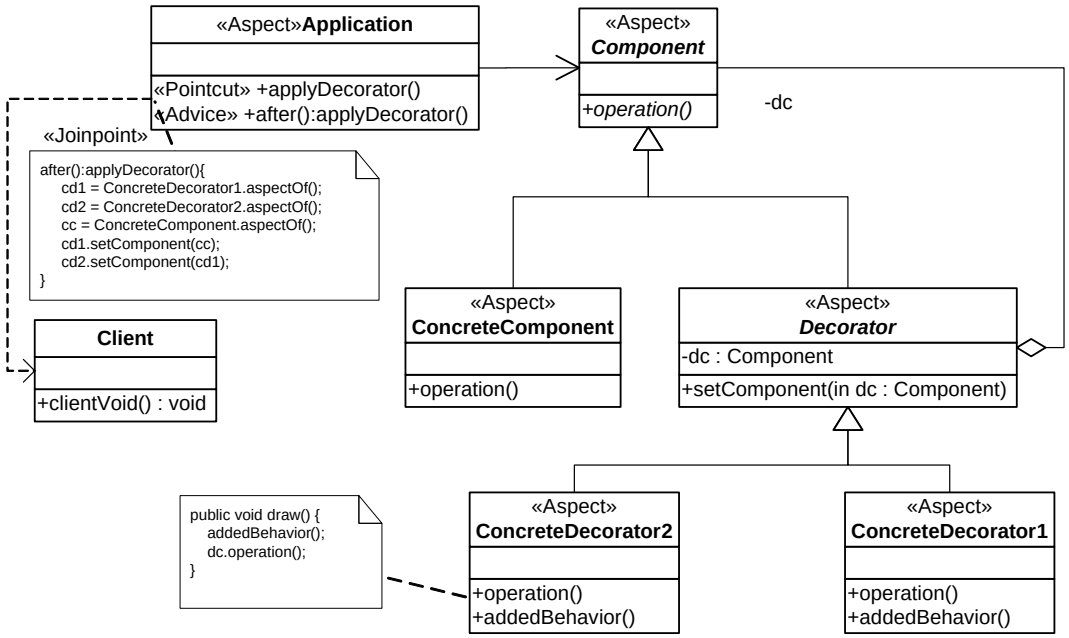


Fig. 43 Decorator design pattern (AO solution)

- *Component* aspect defines the interface for aspects that can have the dynamically added responsibilities.
- *ConcreteComponent* aspect defines an aspect to which the additional responsibilities can be attached.
- *Decorator* maintains a reference to a *Component* aspect and defines an interface that conforms to *Components* interface.
- *ConcreteDecorator1* and *ConcreteDecorator2* provide additional responsibilities to the *Component* aspect.
- *Application* aspect provides pointcut and the advice for an invocation of a pattern,
- *Client* is the class that invokes the *applyDecorator* pointcut.

Façade

The intent of the aspect-oriented Façade design pattern is to "*provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level*

interface that makes the subsystem easier to use" (Gamma et al., 1994) (Fig. 44).

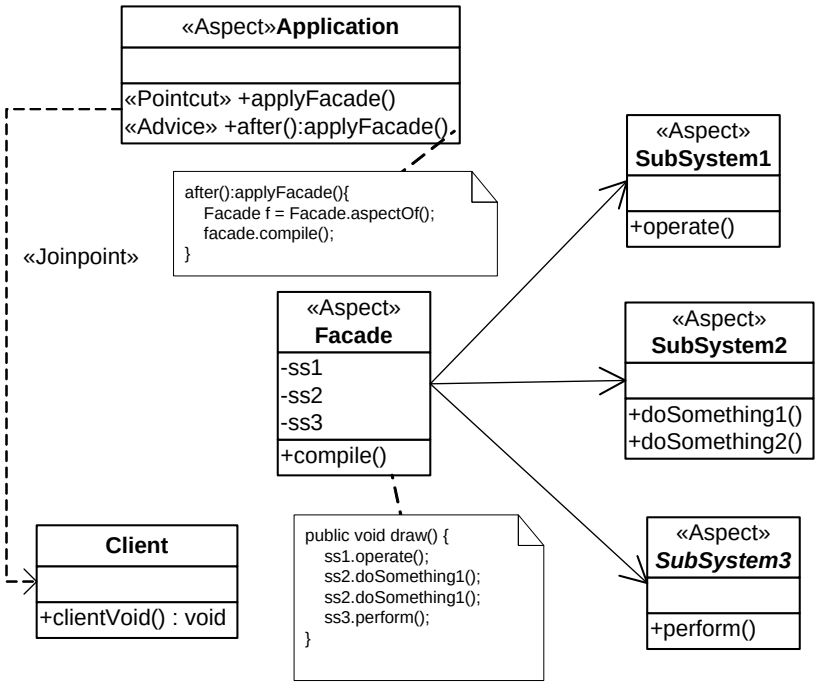


Fig. 44 Façade design pattern (AO solution)

- *Façade* aspect knows which subsystem aspects are responsible for a request, delegates pointcut invocation requests to appropriate subsystem aspects,
- *SubSystem1, SubSystem2, and SubSystem3* aspects implement subsystem functionality and handle work assigned by the Facade aspect, have no knowledge about the *Façade* aspect.
- *Application* aspect provides pointcut and the advice for an invocation of a pattern,
- *Client* is the class that invokes the *applyCommand* pointcut.

Flyweight

The intent of the aspect-oriented Flyweight design pattern is to use sharing to support the use of references to aspects efficiently (Fig. 45).

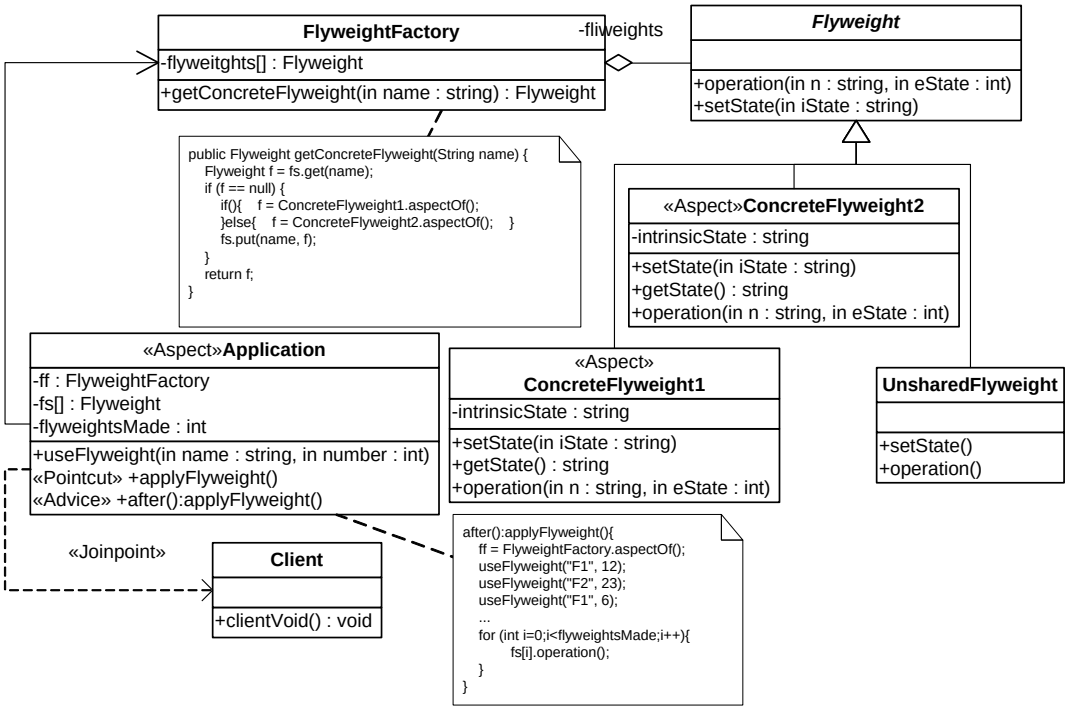


Fig. 45 Flyweight design pattern (AO solution)

- *Flyweight* aspect declares an interface through which *Flyweight* aspects can receive and act on extrinsic state,
- *ConcreteFlyweight1* and *ConcreteFlyweight2* aspects implement the *Flyweight* aspect interface and add storage for intrinsic state, if any. A *ConcreteFlyweight1* and *ConcreteFlyweight2* aspects must be sharable. If only one *Flyweight* aspect for storing several different internal states is required the storing must be implemented elsewhere,
- *UnsharedFlyweight* not all *Flyweight* aspects need to be shared,
- *FlyweightFactory* aspect assigns and manages references to *Flyweight* aspects, ensures that Flyweight aspects are shared properly. When a *Flyweight* is requested, the *FlyweightFactory* aspect supplies an existing reference or assigns one, if none exists.
- *Application* aspect provides operation for assigning extrinsic data to *Flyweight*s and pointcut and the advice used for an invocation of a pattern,
- *Client* is the class that invokes the *applyFlyweight* pointcut.

Interpreter

The intent of the aspect-oriented Interpreter design pattern is to "*define a representation of the grammar of a language and an interpreter that uses the representation to interpret a sentence of a defined language*" (Gamma et al., 1994) (Fig. 46).

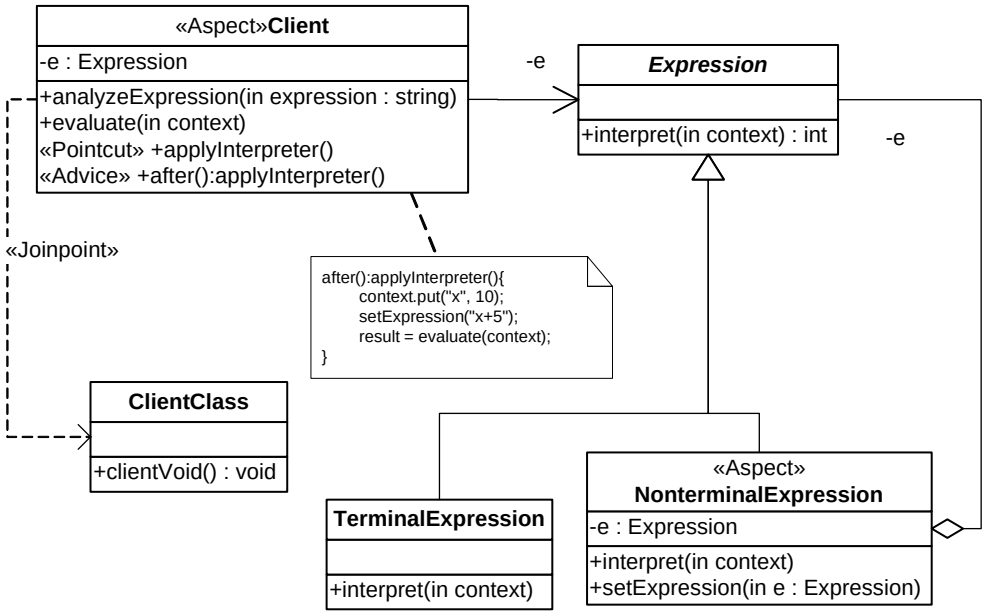


Fig. 46 Interpreter design pattern (AO solution)

- *Expression* aspect declares an abstract *interpret* operation,
- *TerminalExpression* implements an *interpret* operation associated with terminal symbol in the sentence. It is a limitation of this pattern, because only one terminal symbol could be associated with one concrete *TerminalExpression*,
- *NonterminalExpression* one such aspect is required for every rule in the grammar, implements an *interpret* operation for nonterminal symbols in the grammar. The aspect-oriented construction of such design pattern could be performed only with simple expressions with exactly one variable and terminal expression. To avoid this terminal and variable expressions should by handled elsewhere.

- *Client* aspect builds a sentence using defined expressions and invokes the *interpret* operation and provides pointcut and the advice for an invocation of a pattern.
- *ClientClass* is the class that invokes the *applyInterpreter* pointcut.

Iterator

The intent of aspect-oriented Iterator design pattern is to provide a way to access the elements of an aggregate aspect sequentially without exposing its underlying representation (Fig. 47).

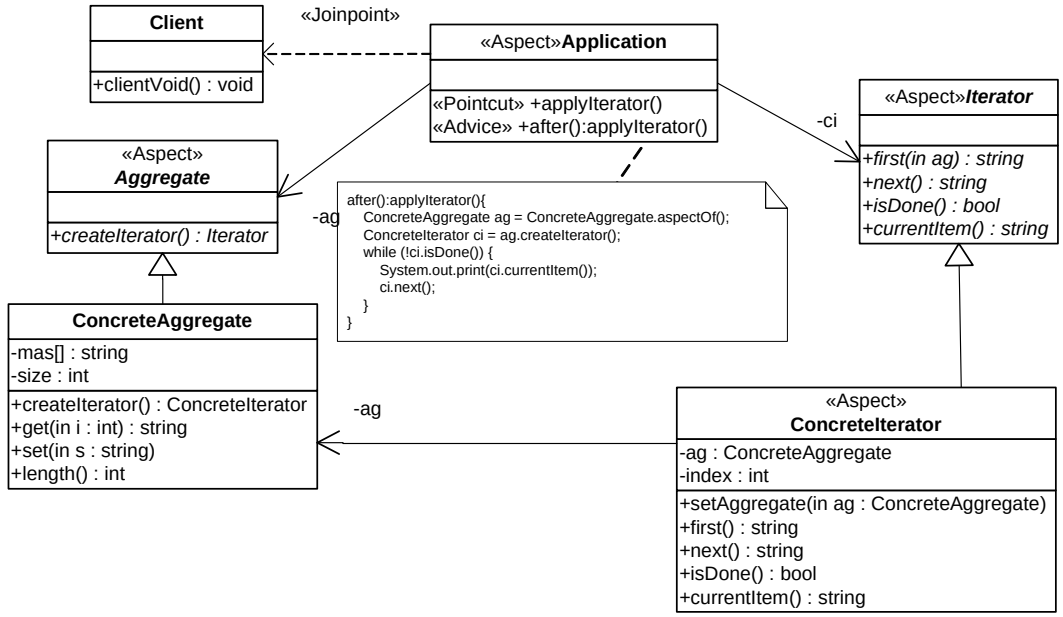


Fig. 47 Iterator design pattern (AO solution)

- *Iterator* aspect defines an interface for accessing and traversing elements.
- *ConcreteIterator* aspect implements the *Iterator* aspect interface and keeps track of the current position in the traversal of the aggregate.
- *Aggregate* aspect defines an interface for referencing an *Iterator* aspect.
- *ConcreteAggregate* aspect implements the *Iterator* aspect referencing interface to return a reference of the proper *ConcreteIterator* aspect.
- *Application* aspect provides pointcut and advice for an invocation of a pattern,
- *Client* is the class that invokes the *applyIterator* pointcut.

Mediator

The intent of the aspect-oriented Mediator design pattern is to define an aspect that encapsulates how a set of aspects interact (Fig. 48).

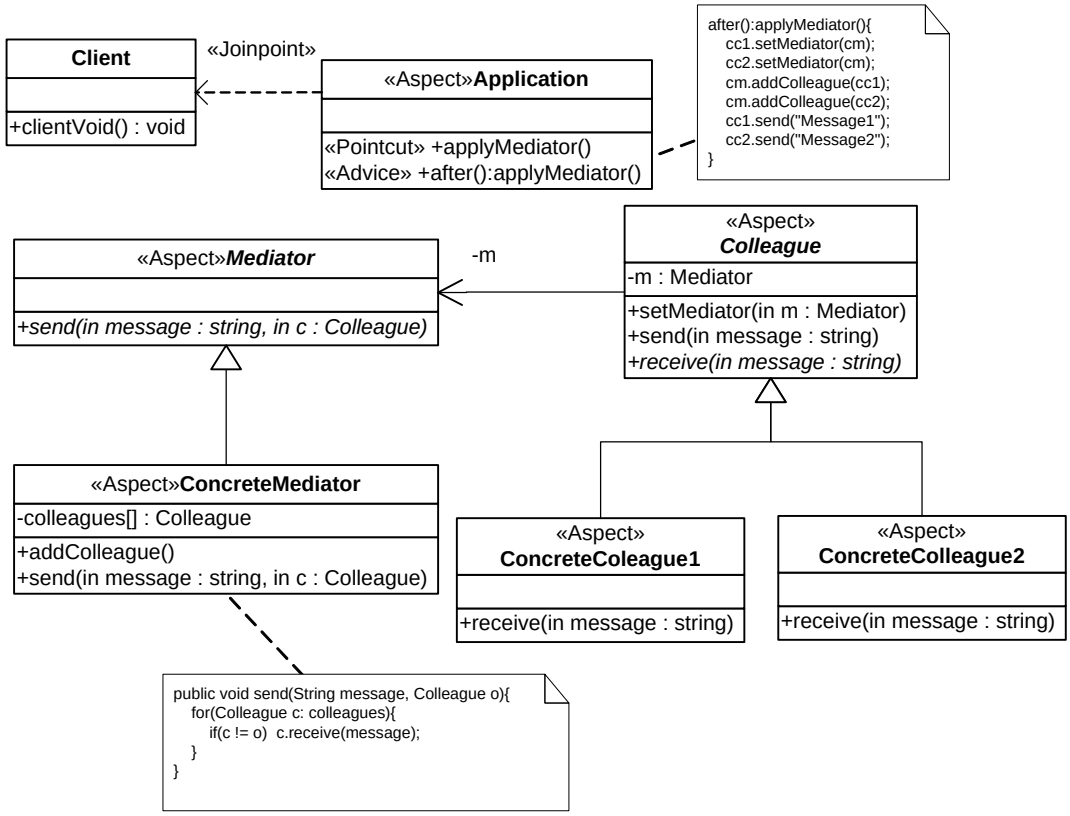


Fig. 48 Mediator design pattern (AO solution)

- *Mediator* aspect defines an interface for communicating with *Colleague* aspects,
- *ConcreteMediator* aspect implements behaviour of *Mediator*, knows and maintains its *Colleague* aspects,
- *Colleague* aspect knows its *Mediator* aspect and communicates with its mediator,
- *ConcreteColleague1* and *ConcreteColleague2* aspects implement *Colleague* aspect,
- *Application* aspect provides pointcut and advice for an invocation of a pattern,
- *Client* is the class that invokes the *applyMediator* pointcut.

Memento

The intent of the aspect-oriented Memento design pattern is to capture and externalize aspects internal state so that the aspect can be restored to this state later (Fig. 49).

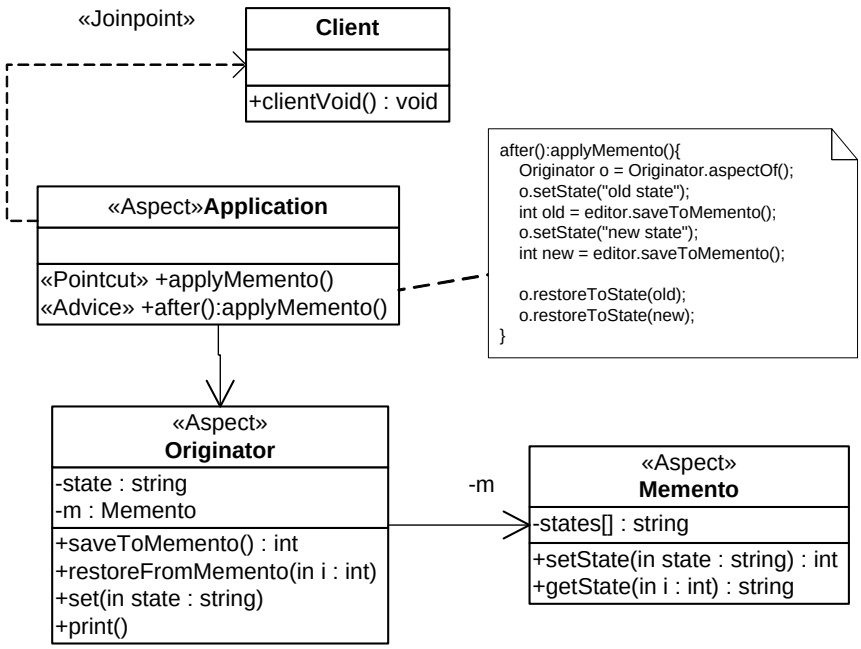


Fig. 49 Memento design pattern (AO solution)

- *Memento* aspect stores internal state of the *Originator* aspect. The memento may store as much as the internal states are necessary,
- *Originator* aspect uses a *Memento* aspect to save its current internal state and to restore its any previous state.
- *Application* aspect provides pointcut and advice for an invocation of a pattern,
- *Client* is the class that invokes the *applyMemento* pointcut.

Observer

The intent of the aspect-oriented Observer design pattern is to define a one-to-many dependency between aspects so that when one aspect changes state, all its dependents are notified and updated automatically (Fig. 50).

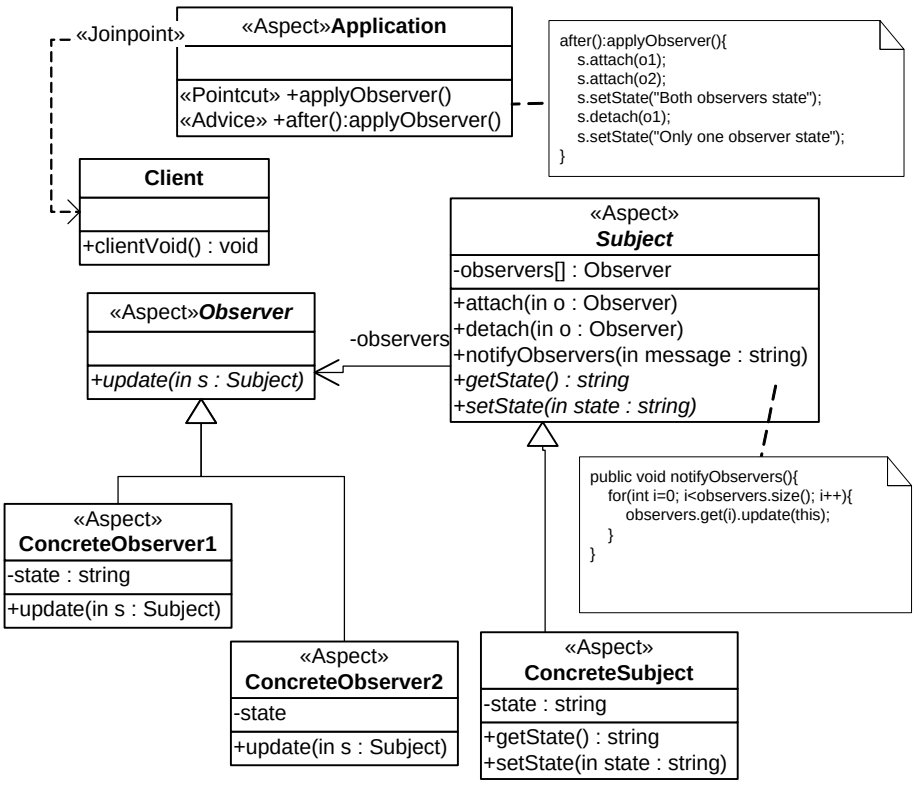


Fig. 50 Observer design pattern (AO solution)

- *Subject* aspect knows its observers, can be observed by any number of *Observer* aspects, and provides an interface for attaching and detaching *Observer* aspects,
- *Observer* aspect defines an updating interface for aspects that should be notified of changes in a *Subject* aspect,
- *ConcreteSubject* aspect stores state of interest to *ConcreteObserver1* or *CocreteObserver2* aspects and sends a notification to *ConcreteObserver1* or *CocreteObserver2* when its state changes,
- *ConcreteObserver1* and *ConcreteObserver2* aspects maintain a reference to a ConcreteSubject aspect, store state that should remain the same as the *Subject* aspect state and implement the *Observer* aspect updating interface,
- *Application* aspect provides pointcut and advice for an invocation of a pattern,
- *Client* is the class that invokes the *applyObserver* pointcut.

Proxy

The intent of the aspect-oriented Proxy design pattern is to provide a surrogate or a placeholder aspect for another aspect to control access to it (Fig. 51).

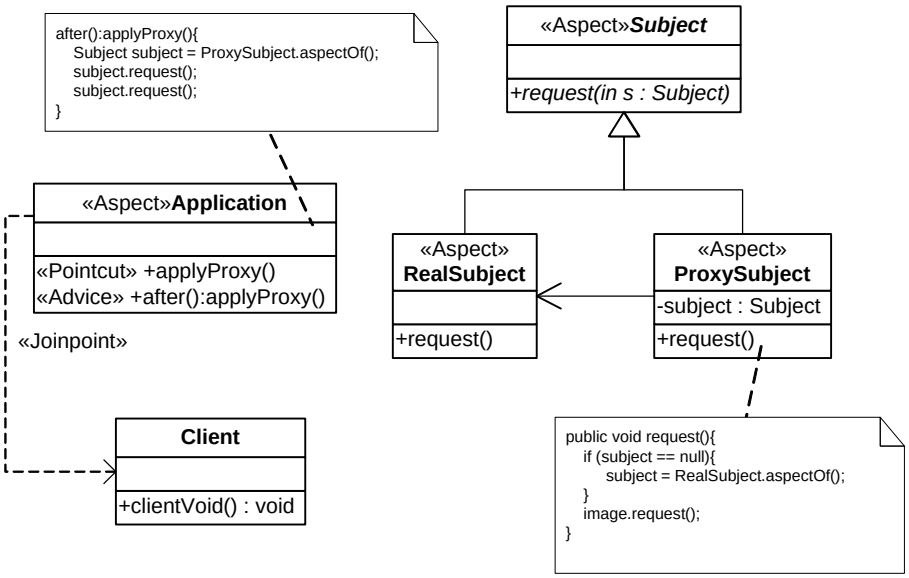


Fig. 51 Proxy design pattern (AO solution)

- *ProxySubject* aspect maintains a reference that lets the proxy access the *RealSubject* aspect. The interfaces of a *RealSubject* and *ProxySubject* aspects are the same,
- *Subject* aspect defines the common interface for *RealSubject* and *ProxySubject* aspects,
- *RealSubject* defines the real aspect that the *ProxySubject* represents,
- *Application* aspect provides pointcut and advice for an invocation of a pattern,
- *Client* is the class that invokes the *applyProxy* pointcut.

State

The intent of the aspect-oriented State design pattern is to allow an aspect to alter behaviour when the internal state changes (Fig. 52).

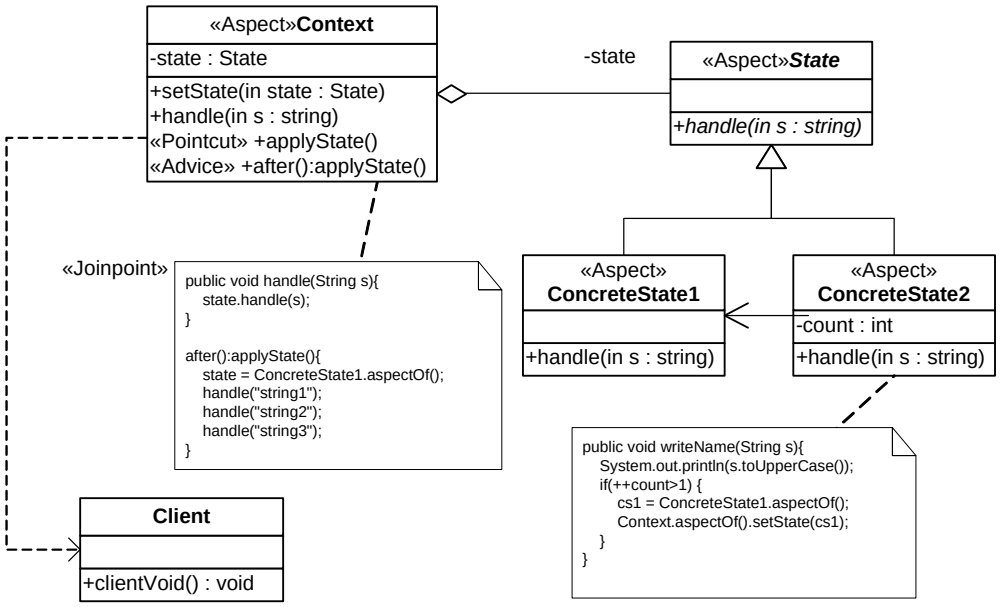


Fig. 52 State design pattern (AO solution)

- *Context* aspect defines the operations for operating *State* aspects, maintains an instance of a *ConcreteState1* aspect that defines the current state, and provides pointcut and advice for an invocation of a pattern,
- *State* aspect defines an interface for encapsulating the behaviour associated with a particular state of the *Context* aspect,
- *ConcreteState1* and *ConcreteState2* aspects implements a behaviour associated with a state of the *Context* aspect,
- *Client* is the class that invokes the *applyState* pointcut.

Strategy

The intent of the aspect-oriented Strategy design pattern is to "*define a family of algorithms that can vary independently from its usage*" (Gamma et al., 1994) (Fig. 53).

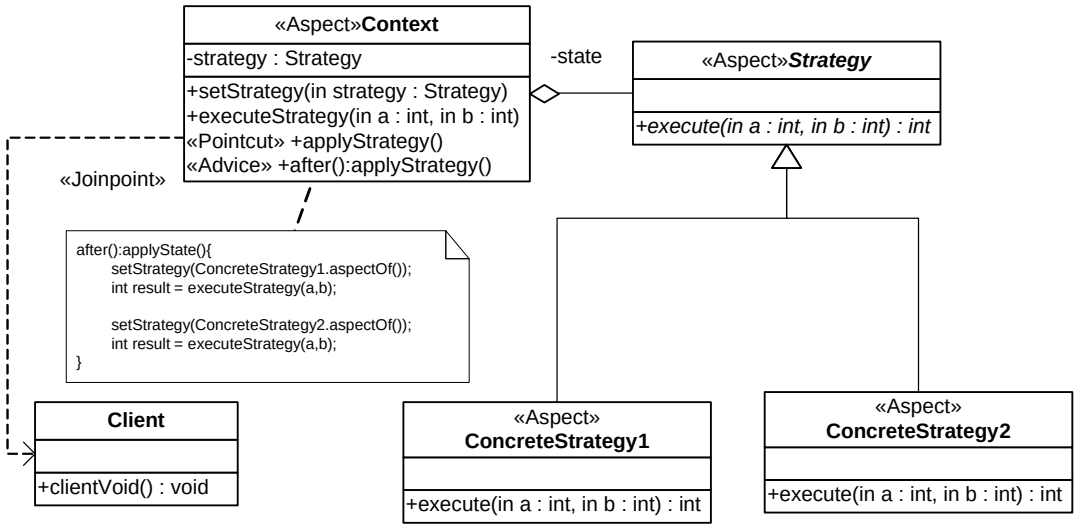


Fig. 53 Strategy design pattern (AO solution)

- *Strategy* aspect declares an interface common to all supported algorithms,
- *ConcreteStrategy1* and *ConcreteStrategy2* aspects implement the algorithm using the *Strategy* aspect interface,
- *Context* aspect uses the interface of *Strategy* aspect to call the algorithm defined by a *ConcreteStrategy*, maintains a reference to a *Strategy* aspect, may define an interface that lets *Strategy* aspect access its data and provides pointcut and advice used for an invocation of a pattern,
- *Client* is the class that invokes the *applyStrategy* pointcut.

Template Method

The intent of the aspect-oriented Template Method design pattern is to define the skeleton of an algorithm in an operation, leaving some steps to complete for subaspects. Template Method allows subaspects redefine certain steps of an algorithm without changing the algorithm's structure (Fig. 54).

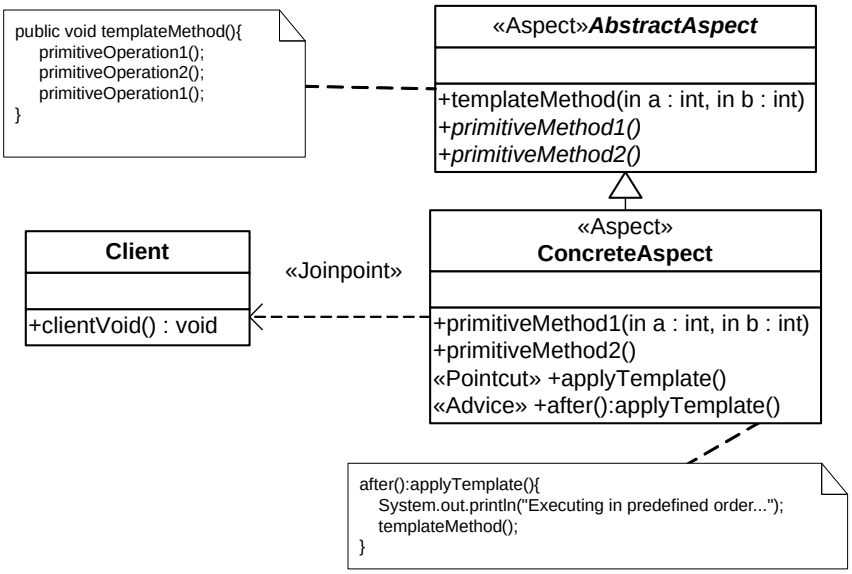


Fig. 54 Template Method design pattern (AO solution)

- *AbstractAspect* aspect defines abstract primitive operations that concrete aspects define to implement steps of an algorithm and implements a template method defining the skeleton of an algorithm,
- *ConcreteAspect* aspect implements the primitive operations completing the steps of an algorithm and provides pointcut and advice for an invocation of a pattern,
- *Client*, the class that invokes the *applyTemplate* pointcut.

Visitor

The intent of the aspect-oriented Visitor design pattern is to represent an operation to be performed on the aspect in elements structure. It lets you define a new operation without changing the aspects of the elements on which it operates (Fig. 55).

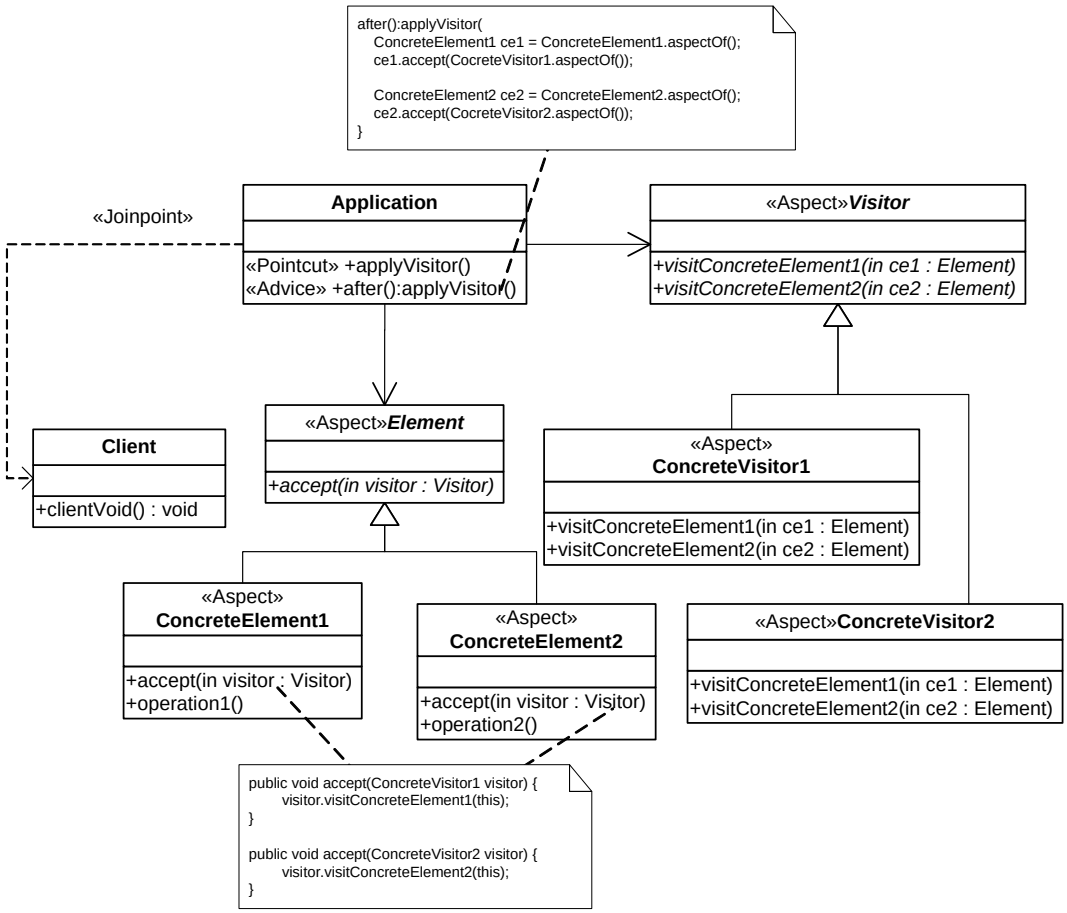
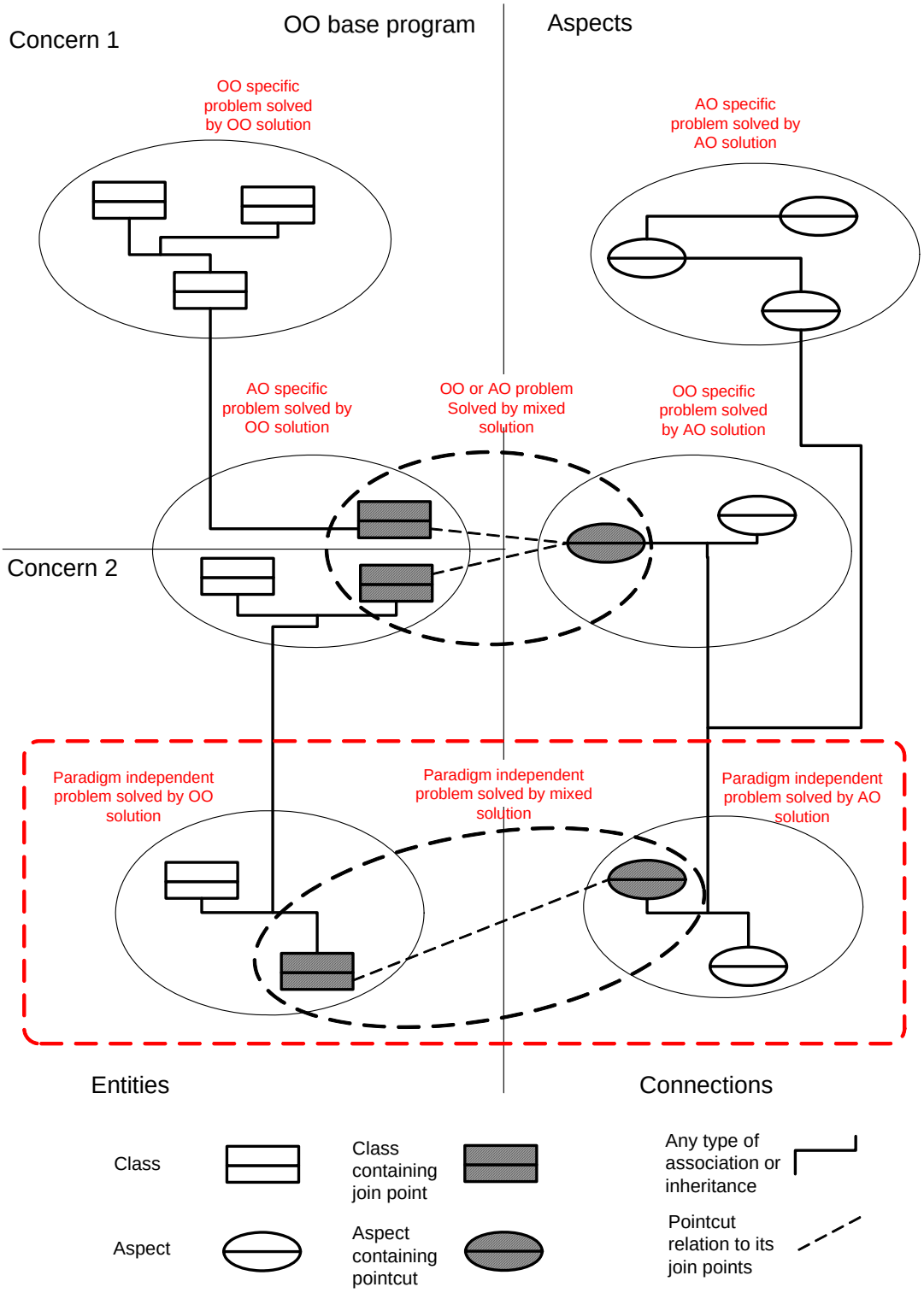


Fig. 55 Visitor design pattern (AO solution)

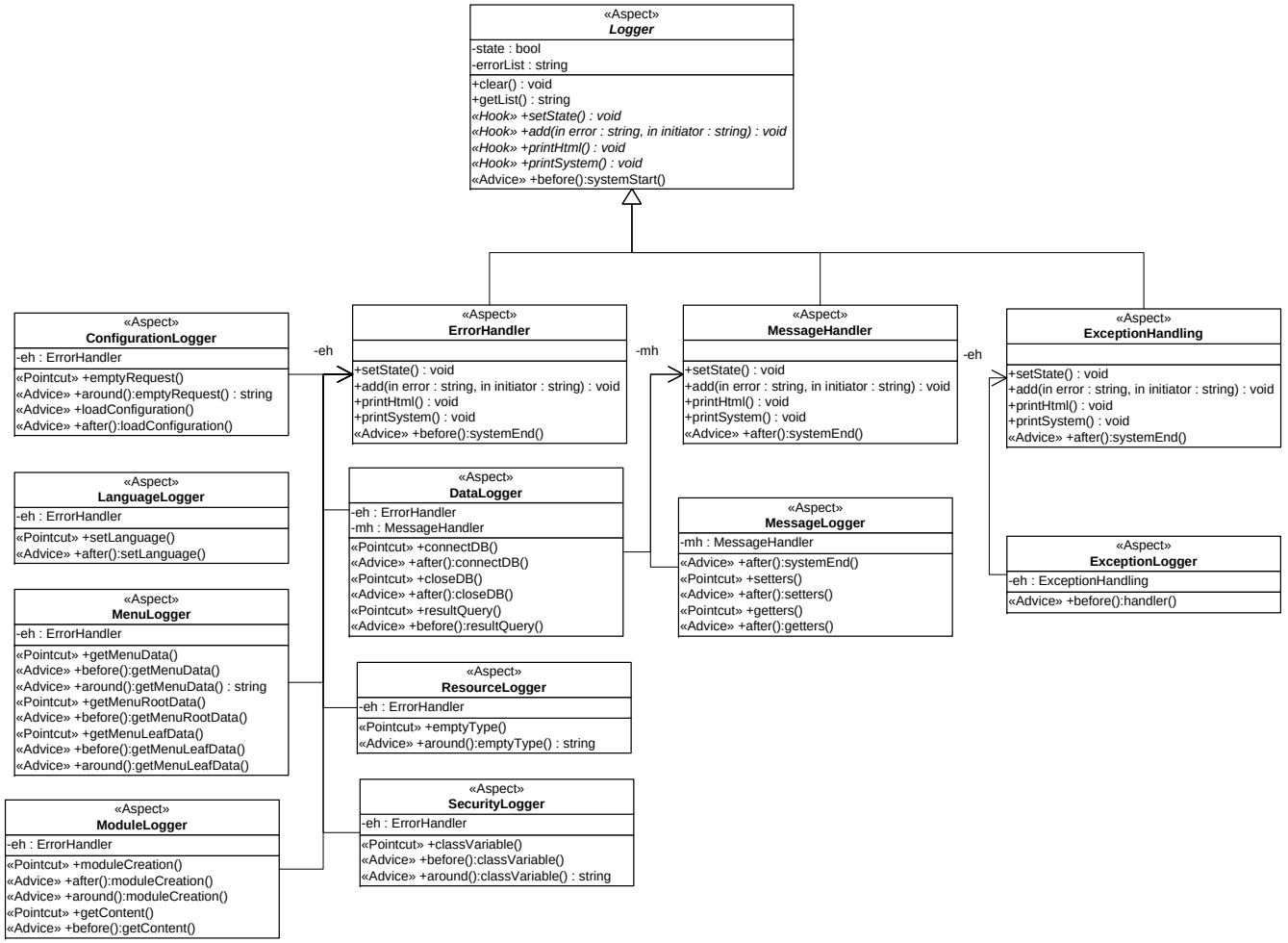
- *Visitor* aspect declares a *visit* operation for each *ConcreteElement1* and *ConcreteElement2* aspects in the *Element* aspect structure. The operation's name and signature identifies the aspect that sends the *visit* request to the *Visitor* aspect. That lets the *Visitor* aspect determine the concrete aspect of the element aspect being visited. Then the *Visitor* aspect can access the *Element* aspect directly through its particular interface,
- *ConcreteVisitor1* and *ConcreteVisitor2* aspect implement each operation declared by *Visitor* aspect. Each operation implements a fragment of the algorithm defined for the corresponding aspect in the structure of *Element* aspect. They provide the context for the algorithm and stores its local state,
- *Element* aspect defines an *accept* operation that takes a *Visitor* aspect as an argument.

- *ConcreteElement1* and *ConcreteElement2* implements an *accept* operation that takes a *Visitor* aspect as an argument,
- *Application* aspect provides pointcut and advice for an invocation of a pattern,
- *Client* is the class that invokes the *applyVisitor* pointcut.

APPENDIX C Graphical diagram illustrating the classification presented in Table 2



APPENDIX D SimpleW Logging concern after second development iteration



APPENDIX E SimpleW Logging concern after third development iteration