INFORMATICA, 2011, Vol. 22, No. 2, 289–317
© 2011 *Vilnius University*

Software Engineering Paradigm Independent Design Problems, GoF 23 Design Patterns, and Aspect Design

Žilvinas VAIRA, Albertas ČAPLINSKAS
Vilnius University Institute of Mathematics and Informatics
Akademijos 4, LT-08663 Vilnius, Lithuania
e-mail: albertas.caplinskas@mii.vu.lt

Received: October 2010; accepted: May 2011

Abstract. The aim of the paper is to investigate applicability of object-oriented software design patterns in the context of aspect-oriented design. The paper analyses which of the GoF 23 patterns have a meaning in this context and how they are affected by it. The main assumption is that there exist design patterns that solve software engineering paradigm independent design problems and that such patterns, in the contrast to the patterns solving paradigm-specific design problems, can be expressed in terms of any software engineering paradigm. However, the paper deals only with two paradigms: aspect-oriented (AO) paradigm and object-oriented (OO) paradigm. It proposes a classification of design problems based on this assumption and a technique for redesigning object-oriented patterns into pure aspect-oriented ones. It presents a number of examples how to apply this technique and discusses the results obtained. The results show that 20 of the GoF 23 design patterns solve such design problems that are common at least for both mentioned paradigms and demonstrate in which way these patterns can be adapted for the needs of aspect-oriented design.

Keywords: aspect-oriented programming, object-oriented programming, design patterns, paradigm-independent and paradigm-specific design problems, aspect-oriented patterns and idioms.

1. Introduction

A number of software engineering paradigms, methodologies and approaches exist to-day. 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. An object-oriented 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 concern separation itself is not enough to develop a new mature software engineering paradigm. It is also necessary to provide some solutions that allow us to cope with other important software engineering issues, including software reuse.

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 Martin (2000):

> "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."

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 Laddad (2003):

> "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."

An example is the Java idiom for ending a program when the window is closed (Example 1).

```
addWindowListener(
  new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
);
```

Example 1. Java idiom for ending a program.

Mainly, the patterns define relationships between the entities in the implementation domain (Shalloway and Trott, 2001) or, in other words, some parameterised collaborations. However, it is difficult to develop a single body of code or even a framework that adequately solves each problem in the family because 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 Tešanović (2004):

"... 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."

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

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 hide 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. We suggest 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 generalising the patterns already defined for some paradigm, defining them in a paradigm-independent way and then specialising 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 GoF 23 patterns solve paradigm-independent design problems and can be adapted to other paradigms. The paper 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.

In the object-oriented programming (OOP), each design pattern defines some parameterized object-oriented collaboration, that is, a parameterized "*relationships between classes and objects with defined responsibilities that act in concert to carry out the solution*" (Maioriello, 2002). The OOP design patterns have already been used for some time and became even "*part of the cutting edge of object-oriented technology*" (Shalloway and Trott, 2001). Many such patterns, for example, the Visitor, Decorator or Observer, are already well researched and the techniques of their application are elaborated in detail. The aspect-oriented programming (AOP) has grown out directly from OOP, but, together with objects, it provides a new kind of entities, namely, aspects. Due to this fact, two new pattern related problems arise: how to use aspects in order to improve the object-oriented design patterns (aspect-oriented implementation of OO design patterns) and how the design patterns can help to design aspects (AO patterns).

The problem of the aspect-oriented implementation of OO design patterns is one far from simple. A detailed analysis should be made in order to understand the implementation of which OO design patterns is affected by the usage of aspects and how. All OO design patterns must be analyzed and redefined from the perspective of AO. 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 compositional properties of aspect-oriented implementation of OO design patterns have been investigated in Hannemann and Kiczales (2002), Garcia *et al.* (2002), Cacho *et al.* (2005) and other publications. Other aspects of the aspect-oriented implementation of OO design patterns have been investigated in Hirschfeld *et al.* (2003), Hachani and Bardou (2002), Noda and Kishi (2001), Nordberg (2001, 2001a), Arnout and Meyer (2006), Bernardi and DiLucca (2005), Piveta and Zancanella (2003), and Cunha *et al.*, (2006). However, the following questions still have no final answers: *How to implement better OO patterns in the aspect-oriented manner*? *Is the aspect-oriented implementation better in some way than the object-oriented one*? *How to measure this*? It is the program for further research.

In parallel with design patterns meant for design objects and classes, in the AO paradigm we also need design patterns purported to design aspects. The problem of the development of AO patterns is even more complicated than the problem of aspect-oriented implementation of OO patterns. At first sight it seems, that patterns for aspects can be defined by analogy to the patterns for objects. 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. Finally, we should discover some useful specific paradigm-dependent AO patterns, if they exist. Although several researches have been pursued on this topic (Lorenz, 1998; Noble, 2007; Bynens and Joosen, 2009), the problem of the development of AO patterns is still far less thoroughly investigated than the problem of the aspect-oriented implementation of OO patterns. A number of important questions are still open, for example: *Does the AO paradigm generate any new patterns that are specific only to this paradigm*? *Which and how OO patterns can be adapted to the AO paradigm*?

The remaining part of this paper is organized as follows. Section 2 analyses related works, Section 3 describes the proposed classification of design problems and Section 4 proposes an approach for rewriting paradigm-independent GoF23 design patterns in terms of aspects and demonstrates the applicability of this approach. Finally, Section 5 concludes the paper.

2. Related Works

2.1. *Aspectization of OO Design Patterns*

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 and Kiczales (2002), Piveta and 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 languages is to localize concern crosscutting. However, as suggested by Filman and Friedman (2001), AOP can be thought of 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."

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 and Ossher, 1993), etc. Although all these techniques have been proposed for separation of concerns, they are different in their nature and, according to 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 and Friedman, 2001), intentional programming, meta-programming, generative programming (Czarnecki and Eisenecker, 2000) and others. All these approaches provide some means to express and to implement quantified statements. However, they differ by the implementation issues they address to. 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 aspect-oriented programming languages introduce special concepts used to describe such quantifications. 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 aspect-oriented 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 and 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 and Friedman, 2001). It means that the AO paradigm is not an independent paradigm like the OO paradigm, but a paradigm that is built by "aspectization" of some other paradigm that remains beyond it. However in this paper we consider only the case where the aspect-oriented paradigm is built over the object-oriented paradigm. In this case, all OO design patterns can be redefined for the AO paradigm. It can be done 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 and Kiczales 2002);
- A native AO solution is introduced to the same problems that are addressed by the OO design pattern (Hachani and Bardou, 2002; Hirschfeld *et al.*, 2003).

A number of metrics have been proposed to measure the effectiveness of the implementation. Hannemann and Kichales (2002) demonstrate that in terms of code locality, reusability, composability, and (un)pluggability even for 17 out of GoF23 patterns the implementation was improved by rewriting from Java to AspectJ. The quantitative assessment of Java and AspectJ implementations for the GoF 23 patterns has also been done in Garcia *et al.* (2005) and Cacho *et al.* (2005). To this end, the authors use the metrics suite, defined in Sant'Anna *et al.* (2003), Garcia (2004). This suite is constituted of metrics, based on separation of concerns, coupling, cohesion, and code size. Garcia *et al.* demonstrate that aspect-oriented implementations of most of the GoF23 patterns improve these patterns regarding the metrics for separation of concerns. 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. Cacho *et al.* (2005) presented an empirical study that investigated whether aspect-oriented implementations support the improved composability of design patterns. Since in the context of a software system the design patterns are composed in many different ways and crosscut 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 (Cacho *et al.*, 2005).

A number of researchers (Lorenz, 1998; Noda and Kishi, 2001; Hachani and Bardou, 2002; Hachani and Bardou, 2003; Schmidmeier *et al.*, 2003) investigated the benefits of implementing GoF23 patterns in AspectJ by direct rewriting their implementation from Java to AspectJ. The research in Hachani and Bardou (2002), Hachani and Bardou (2003) focuses on the confusion, indirection, encapsulation breaching, and inheritance related problems raised by the use of OOP 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 GoF23 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 and 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, in AOP is not to emulate OOP 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 "aspectization" of OO implementations, we can expect that patterns will be implemented 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 23, 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. We prefer to see it rather as an architectural pattern. Nevertheless, the research carried out by Nobel *et al.* demonstrates that the native approach is really promising.

The native approach how to implement some of the GoF23 patterns (Template method, Creational patterns, Factory method) has also been proposed in Hanenberg and Schmidmeier (2003). In addition, 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 approaches consider AO design patterns as patterns that describe the interactions among objects and aspects. In other words, aspectization 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.

2.2. *Development of AO-Specific Design Patterns*

Let us discuss now the question of the development of the patterns solving AO-specific design problems or AOP-specific design patterns. This question has been investigated by Hanenberg and Costanza (2002), Hanenberg and Schmidmeier (2003), Laddad (2003), Schmidmeier (2004), Miles (2004), Griswold *et al.* (2006), Lagaisse and Joosen (2006), Bynens *et al.* (2007), Bynens and Joosen (2009), Menkyna *et al.* (2010). However, the research is still at its early phase. Mainly it is based on the occasional experience gained by developing of the industrial software systems.

According to Menkyna *et al.* (2010), the prevailing part of AOP-specific design patterns can be divided into the following categories: pointcut patterns, advice patterns, and intertype declaration patterns.

As far as we know, Hanenberg and Costanza (2002) was one of the first works on AOP-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 and Costanza (2002):

> "... 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 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 a first step towards AOP-specific design patterns and even can be regarded as proto-patterns.

Hanenberg and Schmidmeier (2003) are going one step ahead. In addition to the proposal how to implement some GoF 23 patterns using the native approach, they have proposed the so-called Pointcut method, which "*is used, whenever a certain advice is needed whose execution depends on runtime specific elements*" (Hanenberg and Costanza, 2002). The authors considered the Pointcut method as an AspectJ idiom, rather than as a pattern because it was not presented in the pattern format. According to Hanenberg and Schmidmeier (2003):

> "... 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."

Nevertheless, it seems today that the Pointcut method is expressed in a language independent AOP vocabulary and can be considered as belonging to the advice category of AOP-specific design patterns rather than as an idiom of AspectJ.

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

- **Wormhole**: "transport context information throughout a method call chain without the need for parameters" (Laddad, 2003; Bynens and 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 and Joosen, 2009; belongs to the category of the pointcut patterns);
- **Director (Default Interface Implementation):** *"abstract aspect with multiple roles as nested interfaces"* (Laddad, 2003; Miles, 2004; Bynens and 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 and 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).

Not one of the patterns presented above is 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 are valuable because

Table 1

The classification of OO and AO design problem solutions.

Problems	Solutions		
	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 and 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 and Kiczales, 2002; Laddad, 2003; Miles, 2004; Hanenberg *et al.*, 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 and 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 and Unland, 2003)

they are significant milestones towards the AOP 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 and Joosen, 2009) to be done.

3. Classification of OO and AO Design Problem Solutions

Let us consider the following classification of the ways of solving OO and AO design problems using design patterns (Table 1)[1].

This classification can be illustrated by a simple graphical diagram. Figure 1 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 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 joinpoints in the classes and pointcuts in the aspects, respectively. The connected classes and

[1] We do not consider such problems that are solved by the composition of several patterns.

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

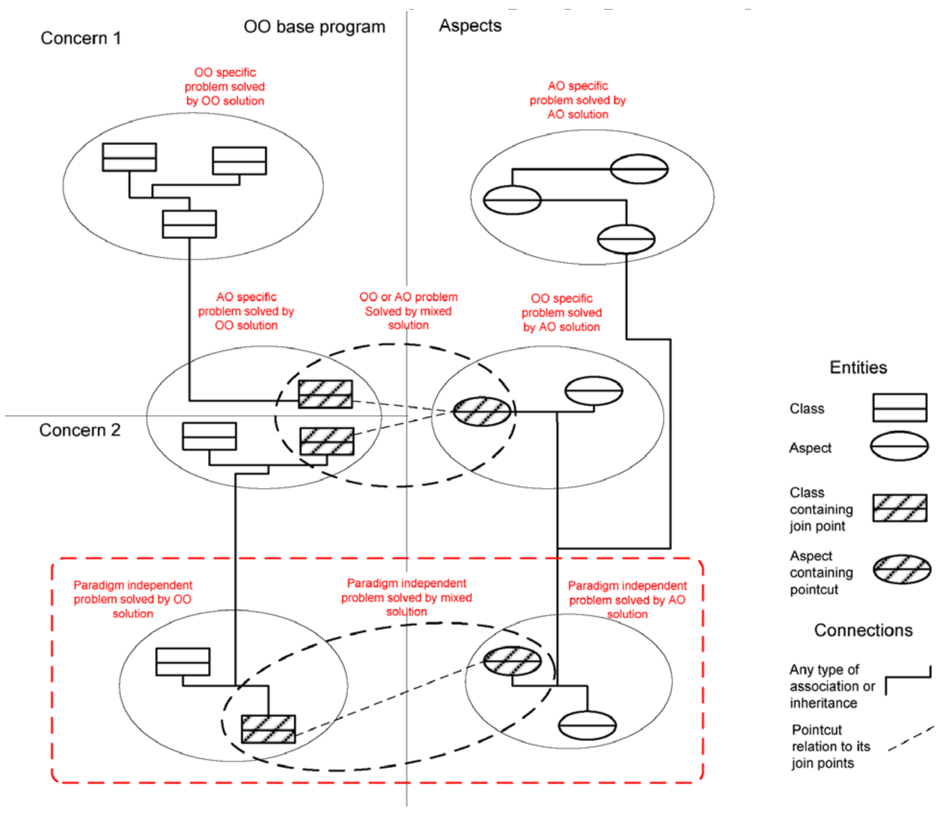

Concern 1
OO base program
Aspects
OO specific problem solved by OO solution
AO specific problem solved by AO solution
AO specific problem solved by OO solution
OO or AO problem Solved by mixed solution
OO specific problem solved by AO solution
Concern 2
Paradigm independent problem solved by OO solution
Paradigm independent problem solved by mixed solution
Paradigm independent problem solved by AO solution
Entities
Class
Aspect
Class containing join point
Aspect containing pointcut
Connections
Any type of association or inheritance
Pointcut relation to its join points

Fig. 1. A graphical diagram illustrating the classification presented in Table 1.

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. 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.

Figure 1 demonstrates all ways that can be used to solve design problems using different implementations of design patterns. Like in Bynens and Joosen (2009), this classification is based on the nature of problems that patterns intend to solve. We consider the 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. We suppose 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, we will demonstrate 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 GoF23 patterns belong to this class. Such aspectizations are composed of pattern-oriented aspects and objects (Hannemann and Kiczales, 2002). A mixed solution of OO specific GoF23 patterns (Prototype, Singleton, and Composite) together with pattern-oriented objects uses pattern-oriented aspects that are bonded with the base OO program (Hannemann and Kiczales, 2002). The Director (Miles, 2004), Container introduction (Hanenberg and 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 and 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 this paper, we deal only with two paradigms: AO paradigm and OO paradigm. In addition, we suppose 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 we know, there are no publications that aim to investigate the class of pure AO solutions solving such problems.

However, Hanenberg and 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 paper is devoted namely to this question. We investigate the GoF 23 patterns, demonstrate that only 20 out of this class of patterns solve paradigm-independent design problems and propose how these patterns can be implemented using AOP constructs only.

4. AO Solutions of Paradigm Independent Design Problems

If some GoF23 pattern 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 AOP 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. Let us briefly consider the proposed methodology, to rewrite paradigm-independent GoF23 design patterns for aspects.

Despite the fact that aspects and classes are different concepts, they have some similarities. 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. So, in the context of this paper, we treat them as singletons. Thus we can use the same or slightly changed structure of GoF23 design patterns to build the AO ones. All we need is to replace OO language constructs by the appropriate AO language – AspectJ in this paper – constructs. It can be done in 3 steps:

- If a GoF23 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 us to access 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 we need at least one class as a placeholder for a joinpoint that initiates the pattern.
- The candidate pattern should be analyzed in order to discover and remove irrelevant data members and methods. Some data members and methods can become irrele-

vant 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 examples of the application of this approach.

4.1. *Investigation of the Applicability of GoF23 Patterns to Design the Aspects*

First of all, let us discuss these GoF23 patterns – Singleton, Prototype, and Composite – that 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 GoF23 patterns can be adopted to solve the aspect design problems. They have been rewritten in AspectJ using only pure AOP constructs. However, the AO implementation of 5 design patterns – Chain of Responsibility, Proxy, Interpreter, Memento, and Visitor – is in some way 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.

Let us now consider, using the above described approach, examples of the AOP implementation of those out of GoF23 design patterns, which can be adopted to solve the aspect design problems... Although we had investigated in detail the implementation of all such patterns, due to the limited space of this paper, we describe the 4 representative examples only: 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, Visitor, Handler, 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.

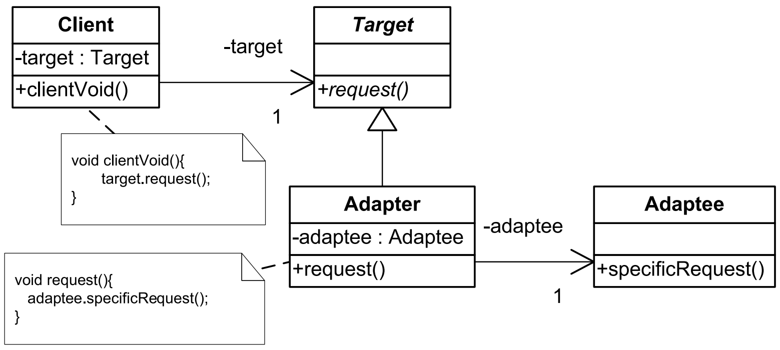


Fig. 2. Adapter design pattern (OO solution).

We use UML class diagrams to model both OO and AO patterns. To represent aspects in UML models we use stereotypes: Aspect, Advice, Pointcut, and Joinpoint. The latter one represents the relation between the pointcut, described in the aspect, and its actual joinpoints in classes. While modelling the AO patterns by UML, we use the traditional UML relations such as inheritance, association, and dependency. For a better understanding of the diagrams, we describe additionally the AO patterns in AspectJ.

Let us consider now GoF 23 Adapter design pattern (Fig. 2). 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.

In order to rewrite the Adapter design pattern for aspects, we apply the proposed

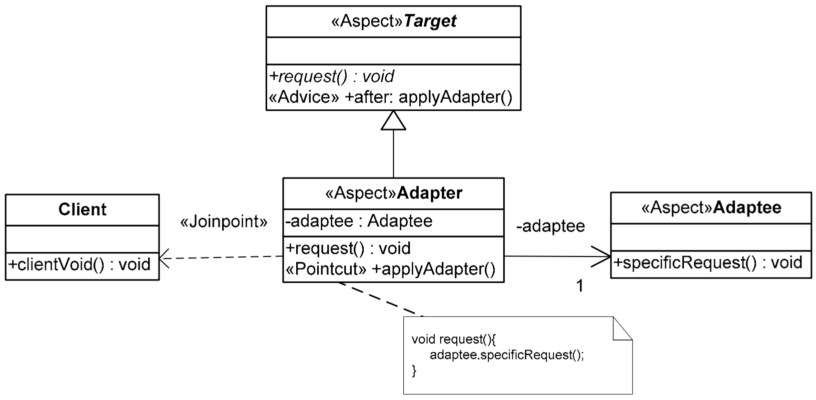


Fig. 3. Adapter design pattern (AO solution).

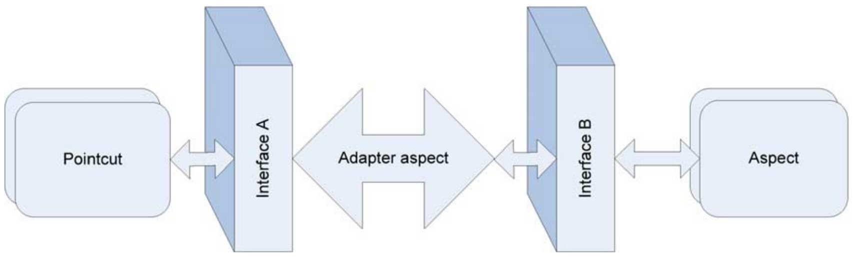


Fig. 4. The idea behind Aspect adapter.

```
public abstract aspect Target {
  void request() ;
   after(): applyAdapter () {
    request();
  }
}

public aspect Adaptee {
  public void specificRequest() {
    System.out.printLn("Executing specific request..")
  }
}

public aspect Adapter extends Target {
  Adaptee adaptee = Adaptee.aspectof();
  void request(){
    System.out.println("Executing inherited request..");
    adaptee.specificRequest();
  }

   pointcut applyAdapter()
  :execution(public static void
      main())&&target(ClientClass);
}
```

Example 2. AspectJ code of the Adapter design pattern.

methodology. In the AO solution (Fig. 3) the classes *Target*, *Adapter* and *Adaptee* are replaced with the aspects *Target*, *Adapter* and *Adaptee*. The class *Client* remains. However, it is not a real object class and serves as a placeholder for a joinpoint 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, we have a solution that consists only of aspects (Fig. 4).

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 *re-*

quest method body and the concrete *applyAdapter* pointcut. The *Adapter* aspect uses the *specificRequest* method defined in the *Adaptee* aspect inside the *request* method.

This example demonstrates how to rewrite the Adapter and other simple object-oriented GoF23 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, we demonstrate below some practical usage of the Adapter AO design pattern.

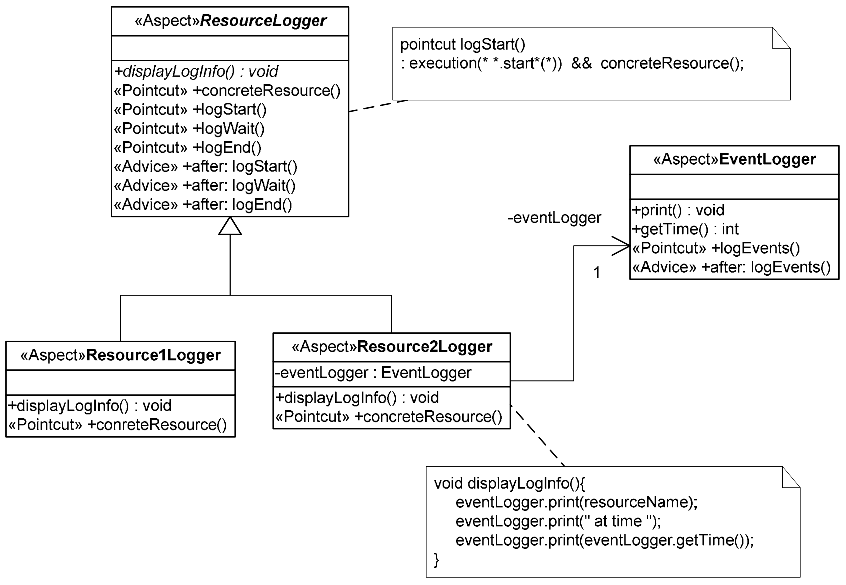
The main intent of *Adapter* is to convert the programming interface of one entity into that of another (Fig. 4). In our case (Example 3), entities are aspects. Let us consider complex Logger concern consisting of several aspects (Fig. 5).

In order to demonstrate a more complex situation, let us consider the GoF23 *Bridge* pattern. 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

We have 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, we can use different aspects for each kind of things to be logged. Thus, we create two aspects responsible for logging events and resources (Fig. 5). 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* we have an abstract operation *displayLogInfo* and an abstract pointcut *concreteResource* that we overwrite in concrete resource loggers. The pointcut *concreteResource* is part of all the other pointcuts and helps us specialize them without rewriting each pointcut. In the *Resource2Logger* we should use operations defined in the *EventLogger*, namely, *print* and *getTime*. In order to adapt these operations to *Resource2Logger*, we apply the Adapter design pattern presented in Fig. 3. 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.

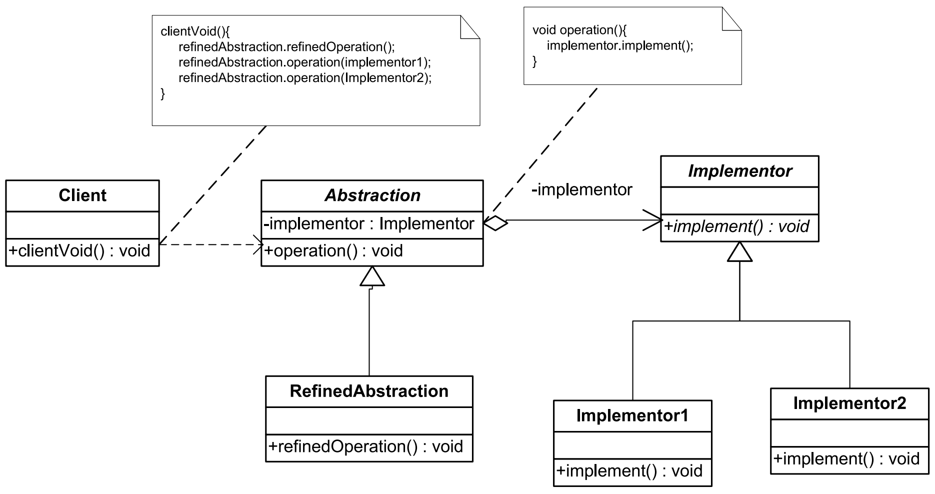
Example 3. AO design pattern Adapter.



```
pointcut logStart()
: execution(* *.start*(*))  &&  concreteResource();
```

```
void displayLogInfo(){
    eventLogger.print(resourceName);
    eventLogger.print(" at time ");
    eventLogger.print(eventLogger.getTime());
}
```

Fig. 5. Application of the AO design pattern Adapter.



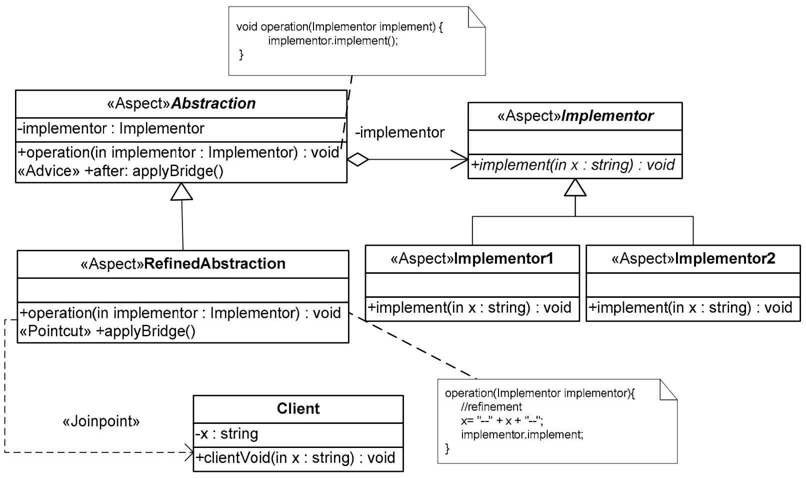
```
clientVoid(){
    refinedAbstraction.refinedOperation();
    refinedAbstraction.operation(implementor1);
    refinedAbstraction.operation(Implementor2);
}
```

```
void operation(){
    implementor.implement();
}
```

Fig. 6. Bridge design pattern (OO solution).

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.*



```
void operation(Implementor implement) {
        implementor.implement();
}
```

```
operation(Implementor implementor){
    //refinement
    x= "--" + x + "--";
    implementor.implement;
}
```

«Aspect»*Abstraction*
-implementor : Implementor
+operation(in implementor : Implementor) : void
«Advice» +after: applyBridge()
-implementor
«Aspect»*Implementor*
+implement(in x : string) : void
«Aspect»RefinedAbstraction
+operation(in implementor : Implementor) : void
«Pointcut» +applyBridge()
«Aspect»Implementor1
+implement(in x : string) : void
«Aspect»Implementor2
+implement(in x : string) : void
«Joinpoint»
Client
-x : string
+clientVoid(in x : string) : void

Fig. 7. Bridge design pattern (AO solution).

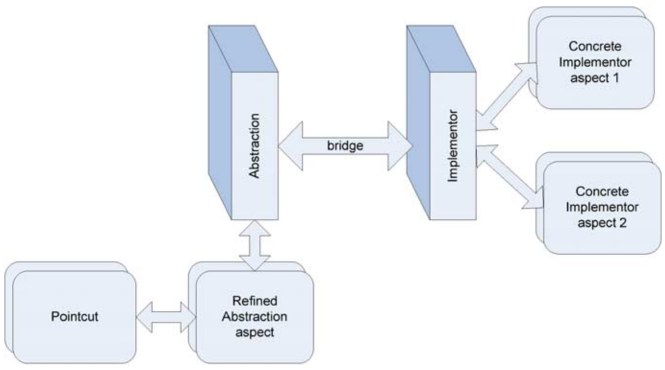

Abstraction
bridge
Implementor
Concrete Implementor aspect 1
Concrete Implementor aspect 2
Pointcut
Refined Abstraction aspect

Fig. 8. The idea behind Aspect Bridge.

Similarly as in the Adapter pattern, in the Bridge pattern (Fig. 7) 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 we have the solution that consists only of aspects (Fig. 8).

Example 4 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.

As far as we are dealing in the AO paradigm with the singletons only, it may seem that AO solutions for the creational design patterns have no sense. Nevertheless, the fact

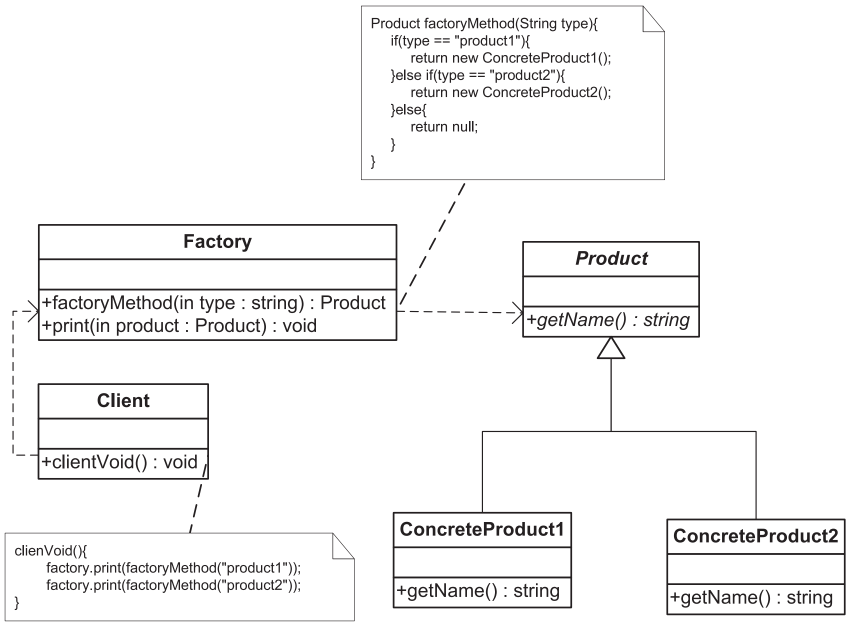
```
public abstract aspect Abstraction {
   String x;

   public void operation(Implementor implementor) {
      implementor.implement(x);
   }

   after(String x): applyBridge(x) {

      this.x = x;
      operation(Implementor1.aspectof());
      operation(Implementor2.aspectof());
   }
}

public aspect RefinedAbstraction extends Abstraction {

   public void operation(Implementor implementor) {

      //refinement
      x = "-"+x+"-";

   implementor.implement(x);
   }

   pointcut applyBridge(String x) :
      call(public void clientVoid(String))&&args(x);

}
```

Example 4. AspectJ code of the Bridge design pattern.

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 we have no factories, we still need to obtain references to aspects for many times and the creational patterns are still very useful for this purpose. We will demonstrate bellow what the AO solutions of creational patterns look like and when we can apply such patterns.

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, we use another variation of the pattern – the parameterized factory method (Fig. 9), 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:

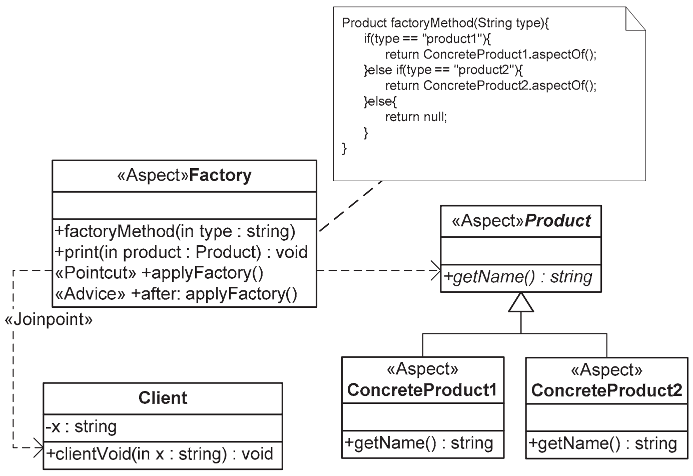
- *Factory,* a class that contains the operation *factoryMethod* which returns the object of type *Product* depending on the requested parameter *type*,



```
Product factoryMethod(String type){
    if(type == "product1"){
        return new ConcreteProduct1();
    }else if(type == "product2"){
        return new ConcreteProduct2();
    }else{
        return null;
    }
}
```

```
clienVoid(){
        factory.print(factoryMethod("product1"));
        factory.print(factoryMethod("product2"));
}
```

Fig. 9. Factory Method design pattern (OO solution).



```
Product factoryMethod(String type){
    if(type == "product1"){
        return ConcreteProduct1.aspectOf();
    }else if(type == "product2"){
        return ConcreteProduct2.aspectOf();
    }else{
        return null;
    }
}
```

«Aspect»Factory
+factoryMethod(in type : string)
+print(in product : Product) : void
«Pointcut» +applyFactory()
«Advice» +after: applyFactory()
«Joinpoint»
Client
-x : string
+clientVoid(in x : string) : void
«Aspect»Product
+getName() : string
«Aspect» ConcreteProduct1
+getName() : string
«Aspect» ConcreteProduct2
+getName() : string

Fig. 10. Factory Method design pattern (AO solution).

- *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.

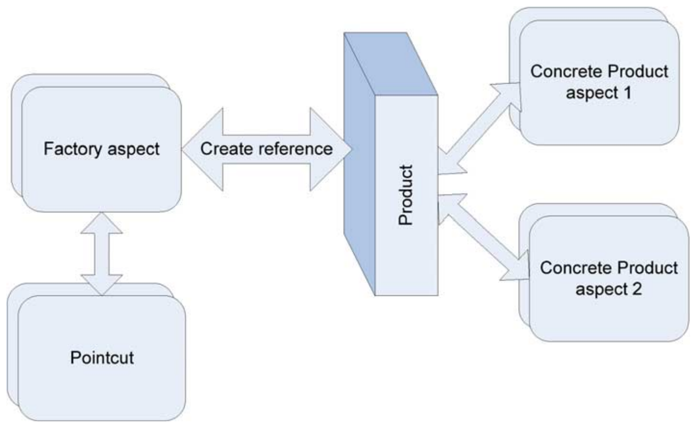
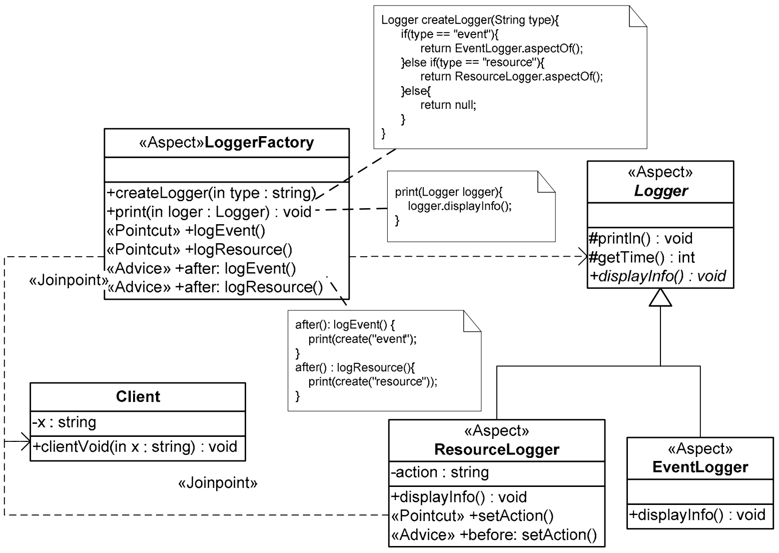


Fig. 11. The idea behind Aspect Factory Method.

```
public aspect Factory {

  static public Product factoryMethod(String type){
     if(type == ″product1″){
      return ConcreteProduct1.aspectOf();
    }else if(type == ″product2″){
      return ConcreteProduct2.aspectOf();
    }else{
      return null;
    }
  }
  private void print(Product product){
    System.out.printf(product.getName()+″\ n″);
  }

  pointcut applyRequest(String x) :
    call(public void clientVoid(String))&&args(x);
  after(String x): applyRequest(x) {
    print(factoryMethod(x));
  }
}
```

Example 5. AspectJ code of the Factory method design pattern.

In the AO solution (Fig. 10) the pattern helps to get a reference to the needed aspect that is defined by the given parameter. We get an analogous result as in the OO version of this design pattern. The difference is that instances of the classes are created each time we execute the main factory method, 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



```
Logger createLogger(String type){
    if(type == "event"){
        return EventLogger.aspectOf();
    }else if(type == "resource"){
        return ResourceLogger.aspectOf();
    }else{
        return null;
    }
}
```

```
print(Logger logger){
    logger.displayInfo();
}
```

```
after(): logEvent() {
    print(create("event");
}
after() : logResource(){
    print(create("resource"));
}
```

Fig. 12. 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 parameterised 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 where we also need concrete loggers 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.

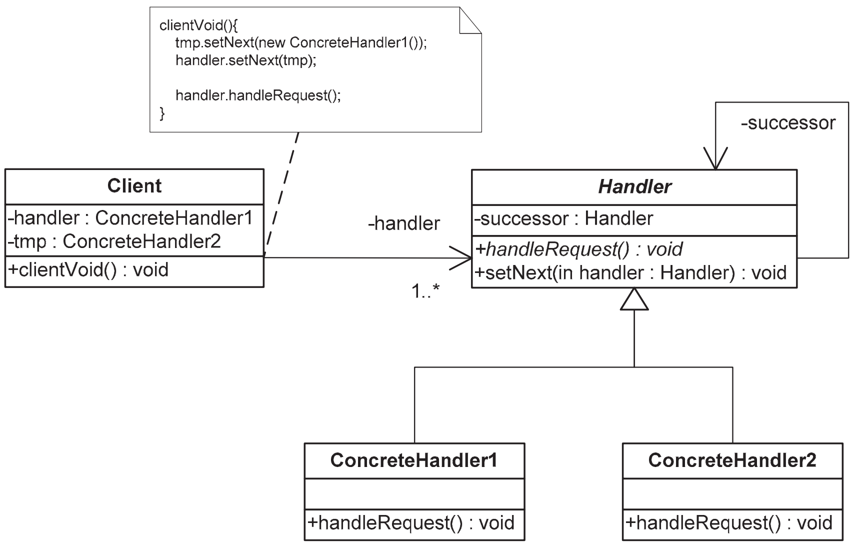
Example 6. AO Factory Method design pattern.

ConcreteProduct1 and *ConcreteProduct2* that extend the abstract aspect *Product* and we have a solution that consists only of aspects (Fig. 11).

The AspectJ code for this solution is presented in Example 5.

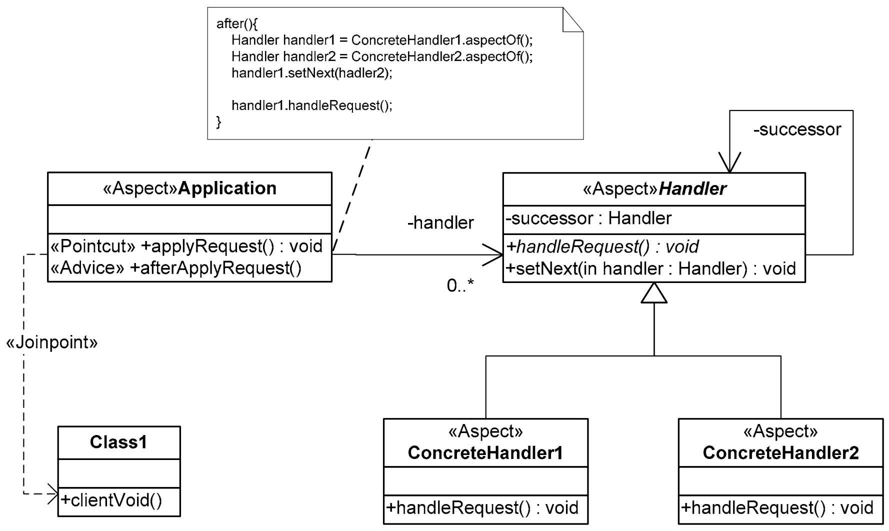
The cardinality of *Product* association in Fig. 10 is set to one, because only one aspect at a moment could be used by *Factory* as defined in the code of the *Factory* aspect (Example 5). This code demonstrates that despite the fact that aspects are singletons, the AO pattern preserves all essential elements of the OO pattern. An example of the application of the AO Factory Method pattern is given in Fig. 12. In this example we deal again with the complex Logger concern consisting of several aspects (Fig. 5).

Finally, let us consider the Chain of Responsibility design pattern as the most representative example of the AO design patterns with the reduced applicability.



```
clientVoid(){
    tmp.setNext(new ConcreteHandler1());
    handler.setNext(tmp);

    handler.handleRequest();
}
```

Fig. 13. Chain of Responsibility design pattern (OO solution).



```
after(){
    Handler handler1 = ConcreteHandler1.aspectOf();
    Handler handler2 = ConcreteHandler2.aspectOf();
    handler1.setNext(hadler2);

    handler1.handleRequest();
}
```

Fig. 14. Chain of Responsibility design pattern (AO solution).

The intent of the Chain of Responsibility 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. 13):

- *Handler,* an abstract class that contains the *handleRequest* operation and defines an interface of *Handler* type objects;

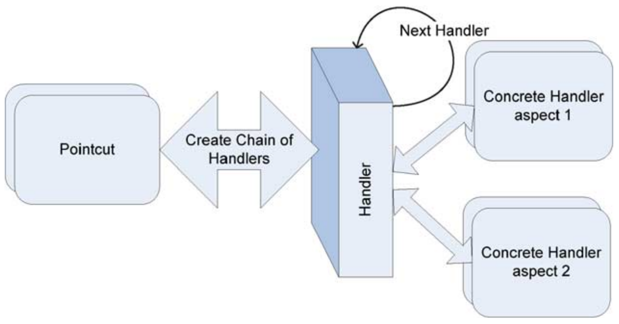
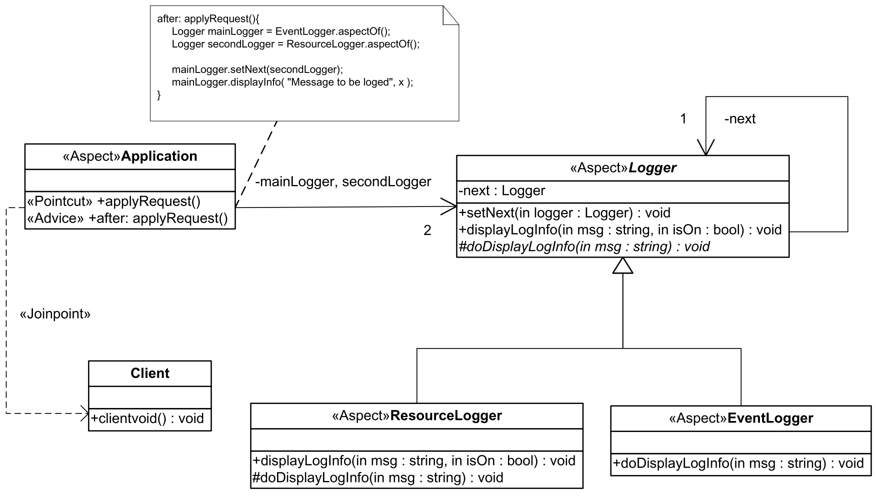


Fig. 15. The idea behind Aspect Chain of Responsibility.

The example bellow (Fig. 16) demonstrates the applicability of the AO Chain of Responsibility pattern. In this example we deal again with a complex Logger concern consisting of several aspects (Fig. 5). The problem is changed slightly to be suitable to apply to the Chain of Responsibility design pattern.
Thus, we still have two different loggers – *ResourceLogger* and *EventLogger,* but we need to log at some of joinpoints using both of them, and only one of them at some other joinpoints. 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*.

Example 7. AO Chain of Responsibility design pattern.



```
after: applyRequest(){
    Logger mainLogger = EventLogger.aspectOf();
    Logger secondLogger = ResourceLogger.aspectOf();

    mainLogger.setNext(secondLogger);
    mainLogger.displayInfo( "Message to be loged", x );
}
```

«Aspect»Application
«Pointcut» +applyRequest()
«Advice» +after: applyRequest()
-mainLogger, secondLogger
2
1
-next
«Aspect»Logger
-next : Logger
+setNext(in logger : Logger) : void
+displayLogInfo(in msg : string, in isOn : bool) : void
#doDisplayLogInfo(in msg : string) : void
«Joinpoint»
Client
+clientvoid() : void
«Aspect»ResourceLogger
+displayLogInfo(in msg : string, in isOn : bool) : void
#doDisplayLogInfo(in msg : string) : void
«Aspect»EventLogger
+doDisplayLogInfo(in msg : string) : void

Fig. 16. Application of the AO Chain of Responsibility design pattern.

- *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*.

In the AO solution (Fig. 14) of the Chain of Responsibility 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. 14), 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, Figure 14 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. 14) falls into an eternal loop. Figure 15 presents the problem solved by the Chain of Responsibility pattern consisting only of aspects.

5. Conclusions and Future Work

The paper investigates the nature of software design patterns and demonstrates that some software system design problems do not depend from a software engineering paradigm that is applied. However, it investigates in detail two paradigms only: aspect-oriented and object-oriented. The paper 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 paper proposes also a technique for redesigning object-oriented patterns into pure aspect-oriented ones and demonstrates application of this technique for the GoF 23 design patterns. To our knowledge, the issues of the development of pure AO design patterns on the basis of the GoF 23 patterns up to time were not investigated. Although originally the GoF 23 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 other GoF 23 patterns arise also in other paradigms including the aspect-oriented one. So, most of these patterns can be useful also in other paradigms. In the AO programming languages such design patterns can be implemented using only AOP constructs. It follows that aspects can be used as collaborative entities, i.e., 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 antipatterns. However, this problem should be investigated in detail. It is the intent of our future research.

References

Aksit, M., Bergmans, L., Vural, S. (1992). An object-oriented language-database integration model: the composition-filters approach. In: Lehrman Madsen, O. (Ed.), *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Utrecht, The Netherlands. *Lecture Notes in Computer Science*, Vol. 615, Springer, Berlin. pp. 72–396.

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

Bernardi, M.L., Di Lucca, G.A. (2005). Improving design pattern quality using aspect orientation. In: *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice* (STEP'05), IEEE Computer Society, Los Alamitos. pp. 206–218.

Bynens, M., 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, New York, ACM, pp. 13–15.

Bynens, M., Lagaisse, B., Joosen, W., Truyen, E. (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, New York, Article No. 2.

Cacho, N., Figueiredo, E., Sant´Anna, C., Garcia, A., Batista, T., 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.

Cunha, C.A., Sobral, J.L., Monteiro, M.P. (2006). Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In: Filman, R.E. (Ed.), *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, AOSD, Bonn, Germany. ACM, New York, pp. 134–145.

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

Dantas, D.S., Walker, D., Washburn, G., Weirich, S. (2008). AspectML: a polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3), 71–130.

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

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

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

Garcia, A., Sant'Anna, C., Figueiredo, E., 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, pp. 14–18. ACM, pp. 3–14.

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

Hachani, O., Bardou, D. (2002). Using aspect-oriented programming for design patterns implementation. In: *Proceedings of 8th International Conference on OOIS*, Position paper at the *Workshop on Reuse in Object-Oriented Information Systems Design*, Montpellier, France.

Hachani, O., Bardou, D. (2003). On aspect-oriented technology and object-oriented design patterns. In: *Proceedings of European Conference on Object Oriented Programming ECOOP*, Position paper at the *Workshop on Analysis of Aspect-Oriended Software*, Darmstadt, Germany.

Hanenberg, S., Costanza, P. (2002). Connecting aspects in AspectJ: strategies vs. atterns. In: Coady, Y (Ed.), *First Workshop on Aspects, Components, and Patterns for Infrastructure Software*, AOSD, Enschede, The Netherlands. TR-2002-12. The Department of Computer Science, University of British Columbia, Vancouver, BC, pp. 40–45.

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

Hanenberg, S., Unland, R., 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, pp. 617–644.

Hannemann, J., 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, New York, pp. 161–173.

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

Hirschfeld, R., Lämmel, R., 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), pp. 25–32.

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

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J, (1997). Aspect oriented programming. In: *Proceedings of European Conference on Object Oriented Programming* (ECOOP), Vol. 1241, pp. 220–242.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G. (2001). Getting started with AspectJ. *CACM*, 44(10), 59–65.

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

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

Lieberherr, K.J., Silva-Lepe, I., Xiao, C. (1994). Adaptive object-oriented programming using graph-based customization. *CACM*, 37(5), 94–101.

Lopes, C.V. (2005). Aspect-oriented programming: a historical perspective (what's in a name?). In: *Aspect-Oriented Software Development*. Addison-Wesley, Reading, pp. 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.

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

Maioriello, J. (2002). *What Are Design Patterns and Do I Need Them?* Online publication, developer.com, QuinStreet Inc. Accessible 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*. Online publication. Accessible at `http://www.objectmentor.com/resources/articles/Principles-and-Patterns`

Menkyna, R., Vranić, V., Polášek, I. (2010). Composition and categorization of aspect-oriented design patterns. In: *Proceedings of 8th International Symposium on Applied Machine Intelligence and Informatics*, SAMI, Herľany, Slovakia, IEEE Press, New York, pp. 129–134.

Meslati, D. (2009). On ASPECTJ and composition filters: a mapping of concepts. *Informatica*, 20(4), 555–578.

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

Noble, J., Schmidmeier, A., Pearce, D.J., Black, A.P. (2007). Patterns of aspect-oriented design. In: *Proceedings of the European Conference on Pattern Languages of Programs* (EuroPLOP), Hillside Publishers, London, pp. 769–796.

Noda, N., 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.

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

Nordberg, M.E. (2001b). Aspect-oriented indirection – beyond object-oriented design patterns. In: *Proceedings of OOPSLA 2001* (Position paper at workshop "Beyond Design: Patterns (mis)used").

Piveta, E.K., 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.

Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., 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, pp. 19–34.

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

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.

Schmidmeier, A., Hanenberg, S., Unland, R. (2003). Implementing known concepts in AspectJ. In: Bachmendo, B., Hanenberg, S., Herrmann, S., Kniesel, G. (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), pp. 65–70.

Tešanović, A, (2004). *What is a Pattern*? Course note, at Linköping University, Sweden. Accessible at `http://www.idi.ntnu.no/emner/dt8100/papers2005/P-a10-tesanovic04.pdf`

Vaira, Ž., Čaplinskas, A. (2009). Compositional aspect-oriented design pattern properties. In: *Proceedings of 50th Conference of Lithuanian Mathematicians Society*. pp. 123–453.

Ž. Vaira is a doctoral student at the Institute of Informatics and Mathematics of Vilnius University. His research interests are aspect-oriented programming and design patterns.

A. Čaplinskas is a professor, principal researcher and the head of the Software Engineering Department at the Institute of Informatics and Mathematics of Vilnius University. His main research interests include software engineering, information system engineering, legislative engineering, and knowledge-based systems.

Nuo konkrečios programų sistemų inžinerijos paradigmos nepriklausančios projektavimo problemos, GoF 23 projektavimo šablonai ir aspektų projektavimas

Žilvinas VAIRA, Albertas ČAPLINSKAS

Straipsnyje siūloma kaip tipinius objektinio projektavimo sprendimus, vadinamuosius GoF 23 projektavimo šablonus, pritaikyti aspektinio projektavimo poreikiams. Analizuojama, kurie iš šių šablonų apskritai yra prasmingi aspektinio projektavimo kontekste ir kaip juos paveikia perėjimas nuo objektinio prie aspektinio projektavimo. Straipsnis remiasi prielaida, jog kai kurie tipiniai sprendimai sprendžia tokias programų projektavimo problemas, kurios nepriklauso nuo konkrečios programų sistemų inžinerijos paradigmos ir kurias tenka spręsti tiek objektinio, tiek ir aspektinio projektavimo kontekste. Jame siūloma, kaip klasifikuoti tipinius projektavimo sprendimus pagal jų sprendžiamų problemų pobūdį ir kaip transformuoti nuo objektinės paradigmos nepriklausomas projektavimo problemas sprendžiančius GoF 23 projektavimo šablonus į aspektinio projektavimo tipinius sprendimus.