Application of pure aspect-oriented design patterns in the development of AO frameworks: A case study

Žilvinas Vaira

Vilnius University, Institute of Mathematics and Informatics, Software Engineering Department, Doctoral student, MS
Vilniaus universiteto Matematikos ir informatikos instituto Programų sistemų inžinerijos skyriaus doktorantas
Akademijos Str. 4, LT-08412 Vilnius
E-mail: zilvinas.vaira@ik.ku.lt

Albertas Čaplinskas

Vilnius University, Institute of Mathematics and Informatics, Principal researcher, Professor, Doctor (HP)
Vilniaus universiteto Matematikos ir informatikos instituto Programų sistemų inžinerijos skyriaus vedėjas, profesorius, daktaras (HP)
Akademijos Str. 4, LT-08412 Vilnius
E-mail: albertas.caplinskas@mii.vu.lt

The paper discusses results of a case study on the advantages applying pure aspect design patterns in the development of aspect-oriented (AO) application frameworks. By an AO application framework we mean a framework which, alongside with traditional object-oriented (OO) framework customization mechanisms, provides also abstract aspects as hot spots. We have tested the hypothesis that pure AO design patterns should promote the ease of designing collaborating abstract aspects representing hot spots. To this end, we studied the process of reworking of the OO simulation application framework into the AO framework. During this study, appropriate qualitative and quantitative data have been collected. The paper presents the generalization of the collected data and concludes that the above hypothesis has been proven.

Introduction

The paper presents results of experimental research on the application of aspect design patterns in the development of aspect-oriented application frameworks. There is still no common consensus as to what the term "aspect-oriented framework" really means. Roughly, all software frameworks, including aspect-oriented ones, may be divided into three categories: application frameworks, domain frameworks and supporting frameworks (Adair, 1995). Aspect-oriented supporting frameworks, for example, JBoss AOP (Fleury, Reverbel, 2003) or Spring AOP (Laddad, 2010), provide the means for implementing crosscutting concerns and/or programming constructs used to specify the crosscutting behaviour of a program. According to Johnson (1988), an application framework is a reusable, "semi-complete" application that can be specialized to produce custom applications. However, later application frameworks have been divided into two categories: frameworks that cover functionality and can be applied to different domains (application frameworks) and frameworks that capture knowledge and expertise in a particular domain (domain frameworks). Sometimes domain frameworks are referred to as enterprise application frameworks (Kaisler, 2005). A domain framework, which produces applications built from a collection of interacting objects, is referred to as an object-oriented domain framework. In this paper, we deal with some category of aspect-oriented domain frameworks, namely white-box frameworks. By the aspect-oriented domain framework we mean a framework which, alongside the traditional object-oriented mechanisms, provides abstract

aspects as hot spots. Such hot spots are specialized by concrete aspects. The applications produced using aspect-oriented domain frameworks are built from a collection of interacting objects which are woven with aspects provided by the framework.

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 GoF 23, have been proposed to ease the designing of object-oriented frameworks (Gamma et al., 1994). A number of ideas (Hannemann, Kiczales, 2002; Noda, Kishi, 2001; Hachani, Bardou, 2003) have been proposed on how to transform GoF 23 patterns into the aspect-oriented ones, however, with the purpose to develop more effective patterns for designing objects. Of course, such patterns are not appropriate for designing aspects. Recently (Vaira, Čaplinskas, 2011) have demonstrated that 20 of GoF23 patterns have also been transformed into pure aspect-oriented patterns (AO GoF 20 patterns) that are purported for designing aspects. By pure aspect-oriented patterns we mean the design patterns that are implemented in the aspect-oriented programming language using only aspect-oriented constructs. The experimental research described in this paper is a qualitative one. It was designed with the aim to validate the following hypothesis:

- under the assumption that the efficiency is measured with regard to the established quantitative parameters such as the code line number, the number of data members and references, the number of involved abstract and specialized entities (classes and/or aspects), the number of hook methods, the number of defined abstract and specialized operations (methods and/or advices), the number of invocations of these operations (calls and/or pointcuts), the framework designs developed using AO GoF 20 design patterns in addition to the GoF 23 patterns are more efficient as compared with its analogues designed using only GoF 23 patterns;
- framework designs developed using AO GoF 20 design patterns allow us to design abstract aspects that facilitate significantly the extension of a framework with new hot spots;
- framework designs developed using AO GoF23 design patterns reduces crosscutting in a framework;
- the possible loss of performance of AO domain frameworks, developed using AO GoF 20 design patterns, compared with their object-oriented analogues, does not exceed 5%.

In addition, the research described in this paper investigates also building techniques of the AO domain white-box framework, under the assumption that they are implemented in AspectJ and Java languages using AO GoF 20 patterns.

The rest of the paper is organized as follows. Section 2 discusses the research methodology, Section 3 describes research settings, Section 4 presents the results, Section 5 surveys related works, and Section 6 concludes the paper.

Research methodology

A case study approach has been used to test the above hypothesis, and the constructive research methodology (Crnkovic, 2010) was applied for this purpose. A case study is an empirical method that aims at investigating some phenomena in their context (Runeson, Höst, 2009). Our research aimed at investigating the impact of the application of AO GoF 23 design patterns on the design of AO domain white-box frameworks. It is a positivist case study (Benbasat et al., 1987) because it measured variables, tested hypotheses and drew inferences from our samples to the whole population of AO domain white-box frameworks. We sought an explanation of the given phenomena, but not in the form of a causal relationship. We investigated both the design results and the design process itself. Different research methodologies can be applied to this end. We selected the constructive research methodology. According to Kari Lukka (Lukka, 2003), constructive research is an experimental

research procedure that can be used to test a hypothesis by developing an innovative construction which implements the assumptions of this hypothesis. Generally, the novel construction should be an abstract notion with a great, actually infinite, number of potential realisations. In our case, it is the AO domain framework. The innovative construction and its development process were considered as test instruments to validate, refine or even to develop an entirely new hypothesis 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 the falsification of a hypothesis only. A case study itself cannot prove any hypothesis and should be linked to some hypothetico-deductive model of explanation. However, the closeness of a case study to real-world situations and its multiple wealth of details argues that this view is correct only in part. In some cases, the results of a case study can be successfully generalised (Flyvbjerg, 2004). This depends upon the case one is speaking of, and how it is chosen. The generalizability of case studies can be increased by the strategic selection of cases (Ragin, 1992). The selected case should be either critical or typical. A critical case is an atypical or extreme case used, in parallel with typical or representative cases, to test the hypothesis in critical situations. From the point of view of our research, a representative example was the framework designed using at least one design pattern from each kind – creational, structural, behavioural – of AO GoF 20 patterns, and a critical case was the one that required application of all AO GoF 20 patterns. For this experimental research, we selected randomly two representative cases. No critical case was investigated; it is the issue of further research.

Although our research, similarly to any other case study, cannot provide statistically significant conclusions, different kinds of evidence, figures and statements are linked together to support strong and relevant conclusions. We use also some quantitative data such as the code line number, the number of data members, the number of involved abstract and specialized entities, the number of hook methods, the number of defined abstract and specialized operations, the number of invocations of these operations, etc. Mainly, we followed the *Guidelines for Conducting and Reporting Case Study Research in Software Engineering* prepared by Per Runeson and Martin Höst (Runeson, Höst, 2009). Quantitative data were collected by measurements and qualitative ones by monitoring, analyzing, comprehending and generalising the framework development process.

There exist two basic ways of how an AO domain framework can be developed: 1) to develop the framework from scratch; 2) to transform a certain existing OO domain framework into the aspect-oriented one. We used the second case which is constrained by the existing design of the OO framework and mainly should replace at least some of the applied object design patterns by the relevant aspect design patterns. It is obvious that only the 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 implementing hot spots in the form of aspects (Monteiro, 2006). The main steps of our research methodology are summarized in Table 1.

Table 1 provides a certain completed cycle. The resulted data are compared at several iterations in order to reject or promote the hypothesis. The qualitative data produced by this research include a brief description of the research steps, UML diagrams of the resulted design patterns,

Table 1. ***The study methods***

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

and a summary of the results confirming the hypothesis. The quantitative data are data of measurements performed for each iteration of the cycle.

The study settings

The OO simulation SimJ framework has been chosen for transformation into the AO domain framework. SimJ is a 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. The SimJ provides five hot spots (simulation, events, resources, entities, entity factory). It is a relatively mature framework which has been improved many times.

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

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

Observations and findings

The OO framework SimJ provides four hot spots and contains only one crosscutting concern, namely logging. The framework is designed in such a way that logging is split into

three specialized parts (for each hot spot) which, using appropriate hooks, can be adapted independently for a particular application. Thus, the logging affects three of four hot spots. The code, related to logging, is scattered over into seven classes. We decided to remove this code and to use it to develop abstract aspects that would implement a new hot spot named Logger. It was necessary to remove this code in such a way that the remaining code would still be correct. We will not discuss the amount of efforts required for reworking, because it is beyond the scope of this paper. 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. In this way, we designed three aspects that implement the default behaviour to all resource logging. Such solution allowed us to customize in the applications some part of this behaviour because, in our case, the AO Template Method pattern allowed for an abstract method implementing a hook. Since the default behaviour of the original OO framework provides only one kind of events, we designed only one additional aspect to implement the default behaviour for event logging. It has no abstract methods and, consequently, does not provide any hooks. For the reasons of efficiency, we decided to use this aspect to implement also the subsidiary logging-related functionality (printing messages, get time values). However, this functionality had to be shared also with the resource logging. As the most reasonable decision to solve this problem, we decided to apply the AO Adapter design pattern. The resulting design is presented in Fig. 1. It provides one additional hot spot (Logger) that can be customized in the applications by overriding the provided hook method.

This design improves the maintainability and unplugability of the logging as compared with the original OO framework because all the logging functionality and the related code are 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 did not allow us to customize the event logging. For this reason, the second design iteration was performed. Since it is reasonable to model the logging behaviour of resources and events by the behaviour of an hierarchy of more specific loggers (Fig. 2), the AO

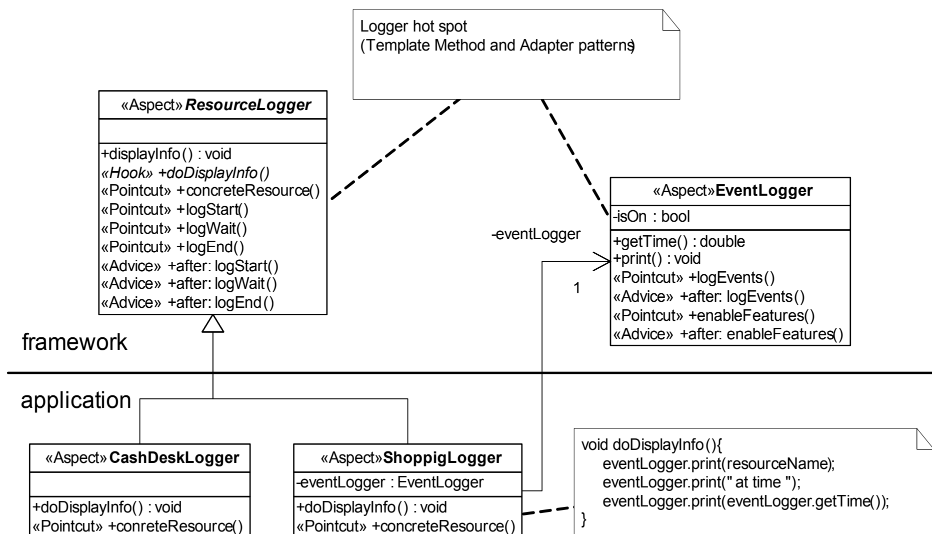

Fig. 1. ***SimJ Logger concern after the first development iteration***

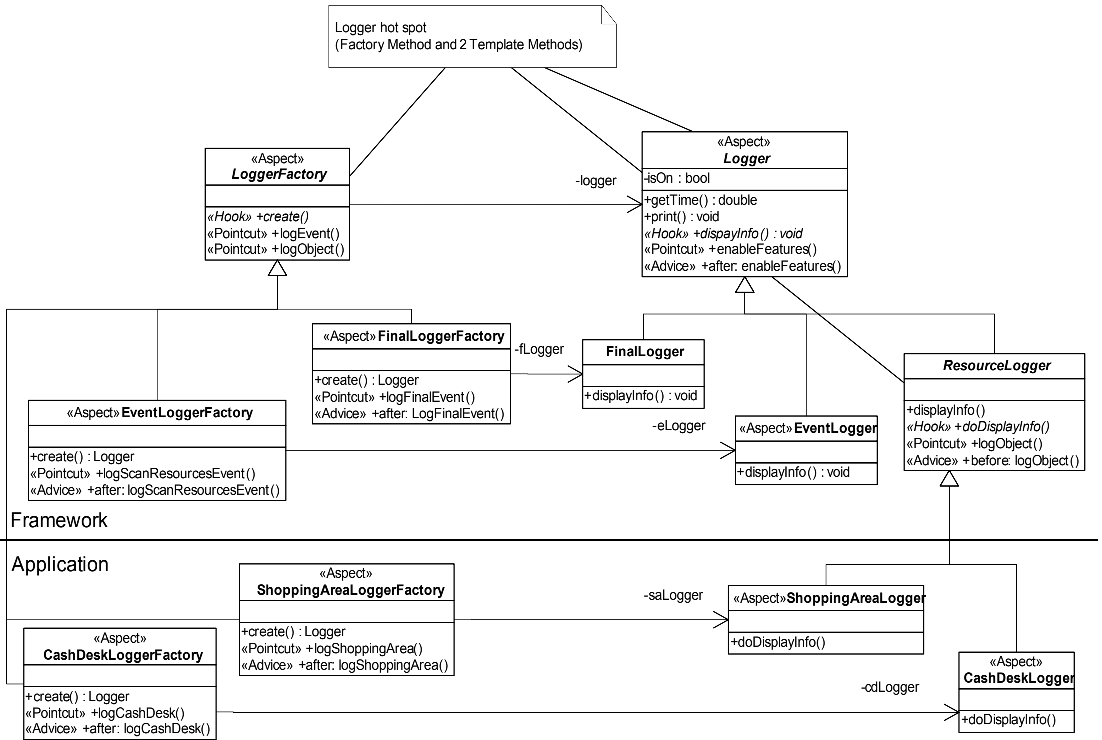

Fig. 2. ***SimJ Logger concern after the second development iteration***

Factory Method design pattern was 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 Logger aspect; therefore, the AO Adapter design pattern is no longer necessary (Fig. 2). On the other hand, the AO Template Method design pattern was applied to design hooks for the Final Logger and the Event logger. So, in the final design, three additional hook methods were designed for the logging hotspot (Fig. 2).

Thus, the final design is an evidence that AO GoF 20 design patterns allow us to design the abstract aspects that facilitate the OO framework extension with new hot spots, and the application of these patterns reduces crosscutting in the framework.

Measurements and data analysis

During both SimJ framework development iterations, some quantitative data on the structure of the code and on the performance of the applications produced using the AO SimJ framework were collected. They are presented in Figs. 3 and 4 by the coresponding bar graphs. Each graph contains three bars: the O bar corresponds to OO implementation, the A1 bar to AO implementation after the first development iteration, and the A2 bar to AO implementation after the second development iteration. The measurements in Fig. 3 are presented as quantities and in Fig. 4a and Fig. 4b as milliseconds. Data on the structure of the code (Fig. 3) demonstrate that the complexity of the code in general decreases. The number of code lines and data members remain almost the same. The first AO development iteration produced less code than did the OO analogue. However, the second

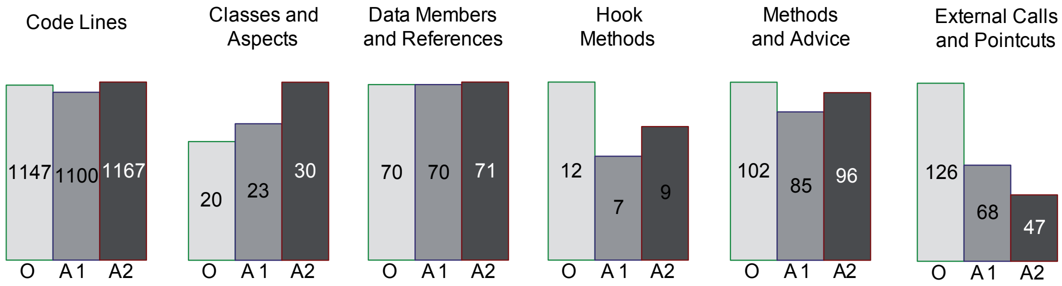

Fig. 3. ***Static quantitative data of measurements (SimJ framework)***

design iteration increased the number of lines, and it became greater than in the OO analogue, but did not exceed 2% and may be considered as acceptable. Besides, the increase of lines was caused by the extended capabilities of logging customization, but not by the application of AO design patterns. The greater number of entities (i.e. classes and aspects) was caused by the finer granularity of the implementation code. It was useful, because the entities became smaller and less complex. During both development iterations, customization was extended by providing one additional AO hot spot. However, the number of hook methods decreased as compared with OO implementation because of the reduced crosscutting of the logging concern. The two additional hook methods were provided by extended customization during the second development iteration A2 than during A1. The number of methods, advice, calls, and pointcuts decreased also in both A1 and A2 cases. The first development iteration produced less methods and advice than did the second one, or the second iteration produced less external calls and pointcuts than did the first one (Fig. 3.).

Tests of the applications produced by the AO SimJ framework revealed some interesting data. After each design iteration, an application was produced, and for each application two tests were performed. In the first test, the application was executed using the logging that aggregated the registered data (Fig. 4a), and in the second test the normal logging functionality was used (Fig. 4b). Each test was performed 50 times in two different modes: 50 separate executions of the application (execution time) and execution of the application 50 times in a countinous cycle (continuous execution time). All executions were performed using the same configuration of the application. Each 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. 4. ***Testing of measurement data (SimJ framework)***

After the first design iteration, the performance of the application, especially in the second mode of execution, was somewhat lost, but it increased again after the second design iteration. This was an unexpected result which cannot be explained on the basis of our observations and requires further investigation.

Hypothesis evaluation

The hypothesis that AO GoF 20 design patterns decrease the complexity of the code was confirmed by both design iterations. The hypothesis that the AO GoF 20 design patterns allow us to design the abstract aspects that facilitate the extension of a framework with new hot spots was also fully confirmed by the design of additional hot spots. The hypothesis that AO GoF20 design patterns reduce crosscutting in the framework was also confirmed because the whole logging implementation code was successfully collected together in logging aspects. The hypothesis that AO GoF 20 design patterns do not cause a significant loss of the performance was confirmed only in part. After the first design iteration, the average of 31% loss of the performance in both execution modes was observed. However, the average performance loss for the second iteration in both execution modes was approximately 0.8% and did not exceed 5%.

Related works

The application of aspects in the design of different frameworks has been studied by several authors; however, the aspects were mostly used to deign frozen spots (i.e. unchangeable parts of a framework). Rausch et al. (2004) used the aspects as a glue code for gluing the framework core and the produced applications. In Santos et al. (2007), Arpaia et al. (2008), the abstract aspects were used to implement hot spots; however, no AO design patterns have been applied for this purpose. More complex design structures that involve some idioms of AspectJ were suggested by Kulesza et al. (2006). These authors proposed how to use extension join points to design hot spots. Hanenberg et al. (2003), Laddad (2003), Miles (2004), Bynens, Joosen (2009) proposed a number of AO design patterns. These patterns can be successfully applied to design AO frameworks. We applied also some of these patterns in our design. However, no experimental data are available on the details of applying these patterns in the framework design. Finally, Vaira, Čaplinskas (2011) proposed how to develop purely AO GoF 20 design patterns. In the present paper, we describe in detail the results of a case study in which we have investigated the application of AO GoF20 patterns for designing abstract aspects in AO domain frameworks.

Conclusions and future work

The case study has demonstrated that AO GoF 20 design patterns can be used to design AO frameworks. During this research, two versions of AO frameworks were designed, and a detailed evaluation of the applied design patterns is presented. The case study has confirmed that AO GoF 20 design patterns decrease the code complexity, eliminate crosscutting, and allow to design additional AO hot spots in frameworks. Performance tests have revealed that in most cases the loss of performance is minimal and fully acceptable. Besides, it depends on the optimization of the design, and the more design refinement steps are performed the better performance can be achieved. The optimization of design depends also on the skills of designers, i.e. on how proper design patterns he/she is able to choose. Of course, it is a kind of art.

In the general case, the AO GoF 20 design patterns and patterns proposed by 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; in particular, pointcut and advice-related design patterns are required.

Our planned research provides for investigation of the application of AO GoF 20 patterns in designing AO domain frameworks from scratch.

LITERATURE

ADAIR, D. (1995). *Building Object-Oriented Frameworks*. AIXpert. Feb. 1995 Appleton, B. 1997. Patterns and software: Essential concepts and terminology [accessed 9 May 2011]. Available from: <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.pdf>.

ARPAIA, P.; BERNARDI, M. L., Di LUCCA, G.; INGLESE V.; 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.; MEAD, M. (1987). The case research strategy in studies of information systems. *MIS Quarterly*, vol. 11, no. 3, p. 369–386.

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, March 2–6, 2009. ACM, p. 13–15.

CRNKOVIC, G. D. (2010). Constructive research and info-computational knowledge generation. Model-based reasoning in science and technology. *Studies in Computational Intelligence*, 2010, vol. 314/2010, p. 359–380.

FLEURY, M.; 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). Vol. 2672 of Lecture Notes in Computer Science. Springer-Verlag, p. 344–373.

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, p. 420–434.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

HACHANI, O.; 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.

HANENBERG, S.; UNLAND, R.; SCHMIDMEIER, A. (2003). AspectJ Idioms for Aspect-Oriented Software Construction. In *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPLoP)*, Irsee, Germany, 25th–29th June, p. 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 Press, p. 161–173.

JOHNSON, R. E.; FOOTE, B. (1988). *Designing Reusable Classes*. Journal of Object-Oriented Programming, June/July, vol. 1(2), p. 22–35 [accessed 4 April 2011] Available from: <http://www.laputan.org/drc/drc.html>.

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

KULESZA, U.; ALVES, V; GARCIA, A.; de LUCENA, C. J. P.; BORBA, P. (2006). Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In *Proceedings of Intl Conference on Software Reuse (ICSR)*, Torino, Italy, p. 231–245.

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.

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, p. 83–101.

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.

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, p. 217–226.

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.

RAUSCH, A.; RUMPE, B.; HOOGENDOORN, L. (2003). Aspect-Oriented Framework Modeling. In *Proceedings of the 4th AOSD Modeling with UML Workshop*, UML Conference 2003, October.

RUNESON, P.; HÖST, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, vol. 14, issue 2, p. 131–164.

SANTOS, A. L.; LOPES, A.; 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, p. 14–24.

VAIRA, Ž.; ČAPLINSKAS, A. (2011). Paradigm-independent design problems, GoF 23 design patterns and aspect design. *Informatica*, 22(2) (accepted for the publishing).

OBJEKTINIO KARKASO PERTVARKYMAS NAUDOJANT ASPEKTINIUS PROJEKTAVIMO ŠABLONUS

Žilvinas Vaira, Albertas Čaplinskas

Santrauka

Straipsnyje pateikiami aspektinių projektavimo šablonų naudojimo aspektiniams dalykiniams karkasams projektuoti eksperimentinio tyrimo rezultatai. Aspektinis dalykinis karkasas – tai toks karkasas, kuriame greta tradicinių objektinio karkaso priemonių naudojami ir abstraktūs aspektai. Atliekant tyrimą siekta išsiaiškinti, kokiu mastu aspektiniai projektavimo šablonai palengvina abstrakčiųjų aspektų ansamblių projektavimą tokiuose karkasuose. Eksperimentas atliktas kaip atvejo analizė. Analizuotas imitacinio modeliavimo uždavinių sprendimo karkaso pertvarkymas iš objektinio į aspektinį. Straipsnyje iškeltos kelios hipotezės apie aspektinių projektavimo šablonų naudojimo rezultatus ir pateikti tas hipotezes patvirtinantys šiame eksperimentiniame tyrime surinkti kokybiniai ir kiekybiniai duomenys.