GDANSK UNIVERSITY OF TECHNOLOGY
Faculty of Electronics,
Telecommunications and Informatics

Adam Przybyłek

Analysis of the impact of aspect-oriented programming on source code quality

PhD Dissertation

Supervisor:
prof. Janusz Górski
Faculty of Electronics,
Telecommunications and Informatics
Gdansk University of Technology

Gdańsk, 2011

Acknowledgment

The work leading to this doctoral thesis has been carried out for four years by me as a research assistant at the Department of Business Informatics, University of Gdańsk. During these four years, a lot of people influenced my dissertation, for which I would like to express thanks.

The head of my department, Stanisław Wrycza, introduced me to the world of being a scientist. I will always be grateful to him for teaching me how to do research and how to work on papers. I learned many invaluable truths from him about academic community. He also supported me in the university. In the face of numerous organizational problems, he always came up with a practical solution. He always believed in my abilities and gave me independence in my research. He also commented and adviced on drafts of this dissertation. However, as soon as my dissertation went beyond his research area, he introduced me to Janusz Górski.

Janusz Górski gave me the opportunity for continuing my studies under his supervision. I owe him many hints, constructive critique and comments that greatly improved the quality of my dissertation. He also helped me to solve some administrative problems.

Michał Przybyłek sparked my interest in AOP. Numerous discussions with him helped me refine my style in scientific thinking, working, and writing. His inspiring criticism of my work and his attachment to detail and perfection significantly improved my research.

Barbara Kitchenham, Leszek Maciaszek and Mieczysław Owoc advised and guided me in research methods. James Noble, David Ungar and the anonymous reviewers of my papers gave me insightful comments. I would also like to thank many scientists, who I had the honor to meet at conferences and discussed my work.

I am also grateful to those researchers who shared the results of their work with me: Alexander Chatzigeorgiou, Eduardo Figueiredo, Phil Greenwood, Stefan Hanenberg, Kevin Hoffman, Konstantinos Kouskouras, Freddy Munoz, and Júlio Taveira.

The Reengineering Forum (REF) industry association gave me a grant from the John Jenkins Memorial Fund to cover the total cost of my participation in CSMR'11. It would not have been possible without the effort and personal engagement of Elliot Chikofsky, Jens Knodel, and Andreas Winter.

Finally, thanks go to the Organizing Committee of TOOLS'11 for giving me an opportunity to attand the conference for free as a student-volunteer.

Abstract

The aim of this dissertation is to investigate the impact of AOP on software modularity, evolvability and reusability in comparison to OOP. In our case study, we compared OO and AO implementations of 11 real-life systems and the 23 GoF design patterns. We also conducted a controlled experiment in which an example program having two implementations (AOP and OOP) was subjected to subsequent increments with the aim to investigate consequences of these increments on program evolvability and reusability. In addition we explored the existing AO implementations of the GoF design patterns according to applying generics and reflective programming. The results of our research demonstrate that AOP has not obvious advantages over OOP as far as modularization is concerned. We also demonstrated that there are limited situations where a reasonable aspect-oriented implementation improves software evolvability and reusability.

Streszczenie

Celem rozprawy jest ocena wpływu paradygmatu aspektowego na: modularność, możliwość dalszego rozwoju oraz ponowne użycie oprogramowania. Jako punkt odniesienia do oceny paradygmatu aspektowego wybrano paradygmat obiektowy. W ramach pracy przeprowadzono studium przypadków, w którym zbadano implementacje obiektowe oraz aspektowe 11 rzeczywistych systemów i 23 wzorców projektowych. Ponadto zrealizowano kontrolowany eksperyment, w ramach którego stworzono prosty system, który następnie został poddany inkrementalnym modyfikacjom polegającym na realizacji nowych wymagań. Na każdym etapie zaimplementowano zarówno wersję obiektową jak i aspektową. Zaproponowano również nowe implementacje wzorców projektowych przy wykorzystaniu paradygmatu aspektowego. Przeprowadzane badania wykazały, że paradygmat aspektowy wbrew powszechnym poglądom nie poprawia modularyzacji oprogramowania. Niemniej, zaobserwowano sytuacje, w których implementacja aspektowa zapewniła lepszą modyfikowalność oraz możliwość ponownego wykorzystania oprogramowania.

Table of Contents

List of abbreviations

AO – Aspect-Oriented

AOSD – Aspect-Oriented Software Development

AOP – Aspect-Oriented Programming

API – Application Programming Interface

CBO – Coupling Between Object classes

CF's – Composition Filters

CK – Chidamber & Kemerer

DML – Data Manipulation Language

GQM – Goal-Question-Metric

GoF – Gang-of-Four

IEEE – Institute of Electrical and Electronics Engineers

IDE – Integrated Development Environment

JDBC – Java DataBase Connectivity

JVM – Java Virtual Machine

LCOM – Lack of Cohesion in Methods

LOC – Lines of Code

MOF – Meta Object Facility

OMG – Object Management Group

OO – Object-Oriented

OOP – Object-Oriented Programming

UML – Unified Modeling Language

SoC – Separation of Concerns

VS – Vocabulary Size

List of figures

List of tables

List of listings

Chapter 1. Introduction

Begin with the end in mind.
Stephen Covey, 1989

The aim of this chapter is to state the research problem, to outline the scope of the research, to present the research methods chosen, and to discuss the related work.

1.1 Overview

The evolution of software development techniques has been driven by the need to achieve a better *separation of concerns* (SoC). A *concern* is a specific requirement or an interest which pertains to the system's development. The term SoC was coined by Dijkstra [1974] and it refers to the ability to decompose and organize system into manageable modules, which have as little knowledge about the other modules of the system as possible Parnas [1972]. In practice, this principle actually corresponds with finding the right decomposition of a problem [De Win et al., 2002]. SoC is closely related to software modularity. The IEEE Standard Glossary of Software Engineering Terminology [1990] defines *modularity* as the degree to which a software system is composed of discrete modules such that a change to one module has minimal impact on other modules. While software engineering gurus vary in their definitions of modularity, they tend to agree on the concepts that lie at its heart; the notion of loose-coupling and high-cohesion [Yourdon & Constantine, 1979; Meyer, 1989; Coad & Yourdon, 1991; Booch, 1994; Fenton & Pfleeger, 1997; MacCormack et al., 2007]. Modularity is considered a fundamental engineering principle since it allows [Baldwin et al., 2000; Brito e Abreu et al., 2002]:

- to develop and test different parts of the same system in parallel by different programmers;
- to break down the problem of understanding a complex system into the independent problems of understanding each module individually;
- to reuse existing parts in different contexts;
- to reduce the propagation of side effects when changes occur.

Concerns can be mapped easily to different modules, if they are functional in nature [Beltagui, 2003]. Such concerns are called core concerns. Kiczales et. al. [1997] found that many systems contain also other kind of concerns, like logging, authentication, error handling, and data persistence, that cannot be represented as first-class entities in the underlying programming language. These are known as *crosscutting concerns*. They usually play a supporting role and capture non-functional requirements or technical-level issues that affect a system as a whole [Przybylek, 2007]. When they are implemented using a traditional language, their code spreads throughout the system. The reason is that traditional languages provide only one dimension along which systems can be decomposed. This limitation is known as the *tyranny of the dominant decomposition* [Tarr et al., 1999] and it states that concerns which do not match to the dominant decomposition must be implemented together with core concerns.

The symptoms of implementing crosscutting concerns in procedural or object-oriented (OO) languages are code scattering and code tangling. *Code tangling* occurs when a module implements multiple concerns. *Code scattering* occurs when similar pieces of code implements the same concern, appear in multiple modules in the program.

Code tangling and scattering run into problems significant enough for practitioners to begin questioning clasical programming paradigms. To achieve more advanced separation of concerns these questioners have proposed a number of approaches such as composition filters, subject-oriented programming, feature-oriented programming and aspect-oriented programming (AOP). The most prominent and recognizable of these is AOP.

AOP introduces a new unit of modularity, an aspect, to implement crosscutting concerns. Although AOP allows programmers to avoid the phenomena of code tangling and scattering it comes with its own set of problems. The distinguishing characteristic of AO languages is that they provide quantification and obliviousness [Filman & Friedman, 2000]. *Quantification* is the idea that one can write unitary and separate statements that have effect in many, non-local places in a program [Filman, 2001]. *Obliviousness* states that the places these quantifications apply do not have to be specifically prepared to receive these enhancements [Filman, 2001]. Quantification and obliviousness may cause problems such as difficulties in modular reasoning [Leavens & Clifton, 2007; Figueiredo et al., 2008]. Furthermore, several new kinds of dependencies are

introduced by the AO constructs to allow for the alteration both of the structure, control and data flow of the modules of the system. These dependencies can make higher the complexity of the code affecting its comprehension [Bernardi & Lucca, 2010]. Hence, AOP by preventing code tangling and scattering improves the comprehensibility of source code in one dimension, and at the same time by introducing quantification, obliviousness, and new kinds of dependencies decreases it in the other dimension. The question is whether the possible gains are worth the confusion it causes.

1.2 Problem statement

Every new technology begins with naive euphoria – the claims of what it can do are exaggerated [Bezdek, 1993]. As a technology grows in strength and moves beyond the embryonic stage, a battle over its acceptance starts. In 2005, Steimann stated the question:

> "*Does aspect orientation really have the substance necessary to found a new software development paradigm, or is it just another term to feed the old buzzword-permutation based research proposal and PhD thesis generator?*"

Paradigms gain their status because they are more successful than their competitors in solving a few problems that the group of practitioners has come to recognize as acute [Kuhn, 1962]. AOP emerged in 1997 [Kiczales et. al., 1997] as a paradigm to implement the concerns that cannot be modularized either in procedural programming or in OOP because of the limited abstractions of the underlying programming languages. Nowadays, with its growing popularity, practitioners are beginning to wonder whether they should start looking into it.

Several studies [Figueiredo et. al., 2008; Filho et. al., 2006; Garcia et. al., 2005; Greenwood et. al., 2007; Sant'Anna et. al., 2003; Soares et. al., 2002] suggest that AOP is successful in modularizing crosscutting concerns. Unfortunately, these studies either are based on intuition and gut feelings, rather than scientific evidence; or wrongly identify modularization with the lexical SoC offered by AOP; or wrongly measure coupling in AO systems. Since we have found indications of the contrary [Przybylek, 2010a, 2010b, 2011b], we argue for the following thesis:

I. **Aspect-oriented programming allows for lexical separation of crosscutting concerns, but it violates the fundamental principles of modular design, such as low coupling, information hiding, and explicit interfaces.**

Since modularity is a low-level quality attribute that influences high-level quality attributes [Fenton & Pfleeger, 1997], we have also tried to assess the extent to which AOP promotes software reuse and systems evolution. This area of research within the AOP community is somewhat restricted by the lack of available AOP-based projects that include adequate maintenance/reuse documentation. Nevertheless, we have observed the superiority of AOP in some narrow scope that is valuable enough to define the second part of our thesis as:

II. **There are limited situations where a reasonable aspect-oriented implementation improves software evolvability and reusability.**

The overall aim of this research is to investigate the impact of AOP on source code quality.

1.3 Justification for the importance and the relevance of the research

The goals of software engineering research include reducing the cost of software development and evolution, reducing the time-to-market, and improving software quality. To a significant degree, outcomes in all of these dimensions depend on design structure [Cai, 2006]. The term structure is used in this dissertation in the same broad sense defined by Ossher [1987]: "*Any system consists of parts such as modules, procedures, classes and methods. The structure of the system is the organization and interactions of those parts*." In particular, developers seek to modularize their systems to better accommodate expected changes, have parts that can be developed and evolved independent from each other, and to ease the understanding of complex designs through abstraction of details hidden within modules [Cai, 2006].

After a decade of research, AOP is still an active topic of discussion in the research community. On the one hand, AOP is glorified and considered as a milestone in programming language development:

- AOP "*is a recent technology for handling crosscutting concerns in a structured and modular manner*" [Hohenstein & Jäger, 2009];
- AOP offers a way to separate concerns and to ensure a good modularization [Guyomarc'h et al., 2005];
- "*AOP is a programming paradigm that increases modularity*" [Hovsepyan et al., 2010];
- "*Given the power of AspectJ to modularize the un-modularizable, I think it's worth using immediately*" [Lesicki, 2002];
- AO software "*is supposed to be easy to maintain, reuse, and evolution*" [Zhao, 2004];
- AOP "*increases understandability and eases the maintenance burden, because modules tend to be more cohesive and less coupled*" [Lemos et. al., 2006].

On the other hand, "disciples of Dijkstra" believe that "*AO programs are ticking time bombs, which, if widely deployed, are bound to cause the software industry irreparable harm*" [Dantas & Walker, 2006]. However, both viewpoints are not backed up by empirical evidence.

In addition, some significant vendors of software like IBM, Motorola, Siemens, and SAP are interested in understanding, evaluating, and applying aspect-oriented techniques. SAP scientists presented a road map to adopting Aspect-Oriented Software Development (AOSD) at SAP for productive use [Pohl et al., 2008]. Siemens developed a large-scale hospital information system (Soarian) that supports seamless access to patient medical records and the definition of workflows for health provider organizations. Aspect were used to implement architecture validation, caching, auditing, and performance monitoring [Rashid et al., 2010].

AOP is also the subject of interest at the most prestigious conferences in software engineering, like OOPSLA, ACM SAC, ICSE, and ECOOP. The Program Committee of the ACM SAC 2010 put forward the question *whether the use of AOP is double-edged*? In the "Call for Papers" for the OOPSLA'08 Workshop on Assessment of Contemporary Modularization Techniques we read:

> A number of new modularization techniques are emerging to cope with the challenges of contemporary software engineering, such as AO Software Development, Feature-Oriented Programming, and the like. The effective assessment of such technologies plays a pivotal role in (i) understanding of their costs and benefits when compared to conventional development techniques, and (ii) their effective transfer to mainstream software development. The main goal of this workshop is to put together researchers and practitioners with different backgrounds in order to discuss open issues on the assessment of contemporary modularization techniques, such as:
>
> - How do new modularization techniques affect working practices and help with software development and evolution? What guidelines can be established from assessment results to improve working practices?
> - What is the impact of using conventional quantitative metrics to assess software modularity? Are they effective enough to assess contemporary modularity techniques? How can we validate assessment mechanisms?
> - What are the potential paths leading to more effective modularization techniques?
> - How can we compare these modularization techniques?

This dissertation provides contributions to answering some of the above questions.

1.4 Research approach

The philosophical stance on which this research is based is *critical-positivist*. Positivists view objective truth as possible, i.e. that there exists some absolute truth about the issues of relevance, even if that truth is elusive, and that the role of research is to come ever closer to it [Seaman, 1999]. Objective knowledge about the real world can be achieved from the empirical knowledge accumulated through perceptual experience [Becker & Niehaves, 2007]. Positivists are reductionist, in that they study things by breaking them into simpler parts. This corresponds to their belief that scientific knowledge is built up incrementally from verifiable observations, and inferences based on them [Easterbrook et al., 2007]. Research methods privileged by positivist are based on the assumption that the measurements of empirical phenomena can be accurate and precise [Cecez-Kecmanovic, 2007]. Positivists prefer methods that start with precise theories from which verifiable hypotheses can be extracted, and tested in isolation. Hence,

positivism is most closely associated with the controlled experiment, however, case studies are also frequently conducted with a positivist stance. [Easterbrook et al., 2007].

The fundamental issue of critical research is that it aims to change the status quo [McGrath, 2005; McAulay et al., 2002]. The critical approach is focused on what is wrong with the world rather than what is right [Walsham, 2005]. It is a different perspective on the analysis that can add up to critical-positivist or critical-interpretivist research [Niehaves & Stahl, 2006]. Critical theorists often use case studies to draw attention to things that need changing [Easterbrook et al., 2007].

The critical-positivist researcher tries to falsify the predictions of the scientific theory. He usually believes that it is more productive to refute theories than to prove them. It is enough to indicate one observation that contradicts the prediction of a theory to falsify it.

1.5 Research methods

The main research method employed in our research is *case study*. A case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context [Yin, 2003]. Case studies offer in-depth understanding of how and why certain phenomena occur, and can reveal the mechanisms by which cause-effect relationships occur. A case study can be applied as a comparative research strategy, comparing the results of using one approach to the results of using another approach [Wohlin et al., 2000]. Case studies can be classified according to the number of cases, as single or multiple cases [Yin, 2003]. A case study is multiple when it involves the examination of more than one similar case. Since a multiple case study does not rely on a sample, the cases investigated do not offer a basis for statistical generalizations, but they are generalizable to theoretical propositions (analytic generalization) [Yin, 2003]. A multiple case design usually offers greater validity [Easterbrook et al., 2007]. Analytic conclusion independently arising from several cases will be more powerful than those coming from a single case alone. The context of these cases are likely to differ to some extent. If common conclusions can still be derived from all the cases, they will have immeasurably expanded the external generalizability of the findings, again compared to those from a single case alone [Yin, 2003].

A case study can involve the examination of more than one unit of analysis. This occures when, within a single case, attention is also given to a subunit or subunits [Yin, 2003]. The unit of analysis defines what the "case" is and it is related to the way the study question is defined [Trindade, 2005]. In software engineering, the unit of analysis might be a project, a particular episode or event, a specific work product, etc. [Easterbrook et al., 2007]. A case study with subunits of analysis is called embedded [Yin, 2003].

We also do experimentation using the quasi-controlled experiment method. A *controlled experiment* is a scientific investigation that takes place in a setting especially created by the researcher [Boudreau et al., 2001]. With this research method, the researcher manipulates one or more independent variables to measure their effect on one or more dependent variables [Basili et al., 1999]. Each combination of values of the independent variables is a treatment. In its simplest form, an experiment has just two treatments representing two levels of a single independent variable (e.g. using OOP vs. using AOP) [Easterbrook et al., 2007]. Experimentation is invaluable in assessing how effective or how promising techniques, paradigms and methodologies are in contrast to other approaches. True experimental research is characterized by manipulation of an independent variable combined with random assignment of participants to groups [Hancock & Algozzine, 2006]. An alternative to true experimental designs are quasi-experimental designs in which experimental rigor so far as manipulation, control, or randomization is not feasible, but the comparison of treatment versus nontreatment conditions is approximated, and the compromises and limitations are stated, understood, and taken into account in all conclusions and interpretations [Mauch & Birch, 2003].

Moreover, we use the *Goal Question Metric* (GQM) approach when we define measurement systems to be used in our empirical studies. GQM is a top-down approach to establish a goal-driven measurement system on three levels (Figure 1.1). It is particularly useful for assessing new software engineering technologies (e.g. what is the impact of the technique X on the productivity of the projects?) [Basili et al., 1994]. The GQM approach was originally developed by Basili & Weiss in the early '80s for evaluating defects and monitoring achievements for a set of projects in the NASA Goddard Space Flight Center environment [Basili & Weiss, 1984; Basili et al., 1994; Solingen & Berghout, 1999]. Today, it is in widespread use for creating and establishing measurement

programs throughout the software industry [Basili et al., 2007]. GQM is typically described as a six-step process where the first three steps are about using business goals to drive the identification of the right metrics and the last three steps are about gathering the measurement data and making effective use of the measurement results to drive decision making and improvements.

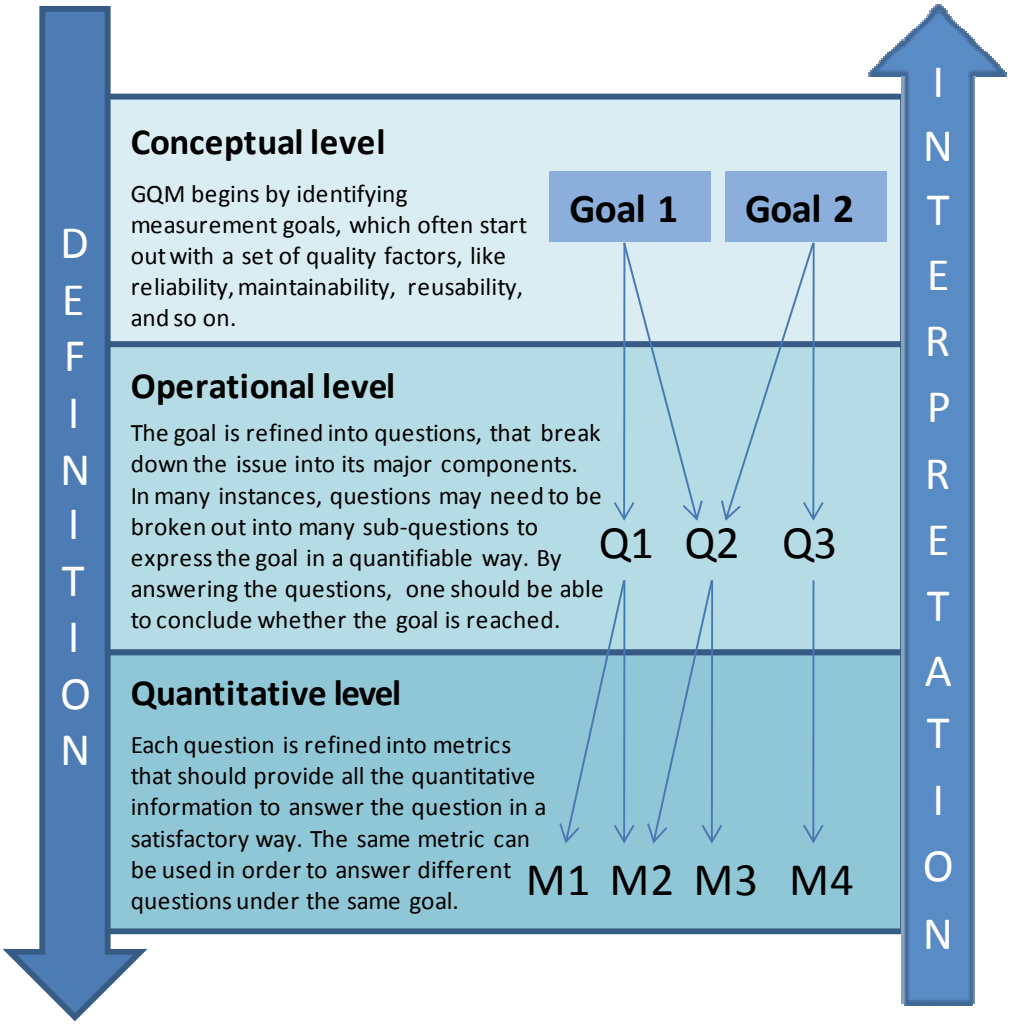


Figure 1.1 The Goal Question Metric approach [Basili & Weiss, 1984; Solingen & Berghout, 1999]

Applying the GQM involves [Basili, 1992; Solingen & Berghout, 1999]:

1. Develop measurement goals for productivity and quality.
2. Generate questions which collectively represent an operational definition of the goal.
3. Specify the measures needed to be collected to answer those questions and track process and product conformance to the goals.
4. Develop mechanisms for data collection.

5. Collect, validate and analyze the data in real time to provide feedback to projects for corrective action.
6. When GQM is implemented to support an organization-wide improvement process, analyze the data in a postmortem fashion to make recommendations for future improvements. The experiences and lessons learned from the study are packaged in the form of policies, procedures and best practices.

In addition, the *action research* method is apply to conduct our supporting research (developing graphical notation for visualizing AspectJ code). In this method, the researcher attempts to solve a real-world problem while simultaneously studying the experience of solving the problem [Davison et al., 2004]. He becomes a part of the research - to be affected by and to affect the research [Milton, 1985]. A precondition for action research is to have a problem owner willing to collaborate to both identify a problem, and engage in an effort to solve it. In some cases, the researcher and the problem owner may be the same person [Easterbrook et al., 2007]. Action research seems to be an ideal research method for the Information Systems field [Avison et al., 2001], especially in those domains where the researcher can be actively involved and benefits for the organization and research community can be expected; where obtained knowledge can be immediately applied and the research process links theory and practice in a cyclical process [Baskerville & Wood-Harper, 1996]. Two key criteria for judging the quality of action research are whether the original problem is authentic (i.e. whether it is a real and important problem that needs solving), and whether there are authentic knowledge outcomes for the participants [Easterbrook et al., 2007].

To conduct action research we follow the *design-science* paradigm. Design-science has its roots in engineering and the sciences of the artificial [Simon, 1996]. It seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts [Hevner et al., 2004]. Such artifacts include - but certainly are not limited to - algorithms (e.g. for information retrieval), human/computer interfaces and system design methodologies or languages [Orlikowski & Iacono, 2001]. IT artifacts are commonly categorized as constructs, models, methods (algorithms and practices), and instantiations (implemented and prototype systems) [Hevner et al., 2004; March & Smith, 1995]. *Constructs* provide the language in which problems and solutions are defined and communicated [Schön, 1983]. The entity-relationship model, for example, is a set

of constructs for representing the type of information that is to be stored in a database. *Models* aid problem and solution understanding. They use constructs to represent a real world situation and to explore the effects of design decisions and changes on the real world. [Simon, 1996]. *Methods* provide guidance on how to solve problems. They can range from formal, mathematical algorithms that explicitly define the search process to informal, textual descriptions of "best practice" approaches [Hevner et al., 2004]. *Instantiations* show that constructs, models, or methods can be implemented in a working system. They are usually in the form of software tools aimed at improving the process of information system development.

Hevner et al. [2004] developed seven guidelines (Table 1.1) for conducting and evaluating good design-science research. Researchers, reviewers, and editors must use their creative skills and judgment to determine when, where, and how to apply each of the guidelines in a specific research project [Hevner et al., 2004].

Table 1.1 Design-Science Research Guidelines [Hevner et al., 2004]

No	Guideline	Description
1	Design as an Artifact	Design-science research must produce a purposeful artifact in the form of a construct, a model, a method, or an instantiation.
2	Problem Relevance	The objective of design-science research is to develop innovative artifacts to important and relevant problems.
3	Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
4	Research Contributions	Effective design-science research must provide clear contributions in the areas of the design artifact, design construction knowledge, and/or design methodologies.
5	Research Rigor	The artifact itself must be rigorously defined, formally represented, coherent, and internally consistent.
6	Design as a Search Process	Design is essentially an iterative search process to discover an effective solution to a problem. Problem solving can be viewed as utilizing available means to reach desired ends while satisfying laws in the problem environment [Simon, 1996]. Means are the set of actions and resources available to construct a solution. Ends represent goals and constraints on the solution. Laws are uncontrollable forces in the environment.
7	Communication of Research	Design-science research must be presented effectively both to a technical audience and to a managerial audience.

1.6 Related work

Work that are mostly related to ours are distributed in four categories: (I) studies that propose coupling metrics for aspects; (II) studies that evaluate the impact of AOP on software modularity; (III) studies that evaluate the impact of AOP on software maintainability and reusability; and (IV) studies that extend UML to support AOM.

Numerous coupling metrics for AO software have been proposed up to now. However, they cannot by used to compare the OO and AO implementations. Zhao [2004], Ceccato & Tonella [2004], Shen & Zhao [2007], and Burrows et al. [2010a; 2010b] proposed fine-grained metrics that separate the coupling contributions of individual AOP mechanisms. Since these metrics measure only a specific kind of coupling, they cannot by used to compare the OO and AO implementations. Our metric quantifies the overall coupling of a given module. The most closely related coupling metric to ours is the one defined by Sant'Anna et al. [2003]. Nevertheless, their metric does not cover all the significant kinds of coupling dependencies in AO software.

There are many studies focusing on a metrics-based comparison among OO and AO modularization. They differ from our research in study settings. Firstly, Garcia et al. [2005], Filho et al. [2006], Hoffman & Eugster [2007], Figueiredo et al. [2008], and Castor et al. [2009] interpret the tally of the metrics' values associated with all the modules for a given implementation, while we interpret the average of the metrics' values.

Secondly, other researchers apply coupling metrics that are invalid to compare between OO and AO implementations. Sant'Anna et al. [2003] and Garcia et al. [2005] do not take into account so-called "semantic dependencies" (see Chapter 5). Other studies can be classified into two groups. In the first group [Filho et al., 2006; Greenwood et al., 2007; Madeyski & Szała, 2007; Figueiredo et al., 2008; Castor et al., 2009], new kinds of coupling introduced by pointcuts are not considered at all. In the second group [Tsang et al., 2004; Hoffman & Eugster, 2007], the coupling introduced by a pointcut is considered only if a module is explicitly named by the pointcut expression.

In addition, Sant'Anna et al. [2003], Garcia et al. [2005], Filho et al. [2006], Greenwood et al. [2007], Figueiredo et al. [2008], and Castor et al. [2009] measure code tangling and code scattering using Concern Diffusion metrics

[Sant'Anna et al., 2003]. They find that AO implementations performed better than their OO equivalents. Since avoiding code tangling and code scattering is the cornerstone of AOP, their observations are predictable and inevitable. In our study we take as given that AOP improves lexical SoC and do not investigate it.

There are also several studies that quantitatively evaluate the impact of AOP on software maintainability and reusability. They differ from our research mainly in the way they measure the quality attributes. Kulesza et al. [2006] evaluate the OO and AO implementations of a Web information system before and after maintenance activities. They apply a suite of metrics for separation of concerns, coupling, cohesion and size. In our opinion, this suite measures software modularity instead of maintainability. Sant'Anna et al. [2003] simulate seven maintenance/reuse scenarios on a multi-agent system. For each scenario, the difficulty of maintainability and reusability is defined in terms of structural changes to the artifacts in the AO and OO implementations, such as number of modules, operations, and lines of code that were added, changed, or copied. Similar metrics suite is used by Figueiredo et al. [2008] to evaluate the stability of software product lines (SPL) that undergoes seven change scenarios. Figueiredo et al. [2008] measure the number of modules, operations, and lines of code that were added, removed or changed during each scenario. In our research, we use one metric to evaluate evolvability and one to evaluate reusability. We consider atomic changes as the indicators of maintenance tasks. The more atomic changes occur between two software versions, the less evolvable the software is. Our reusability metric bases on the ratio of reused LOC to the total number of LOC in a program. Bartsch & Harrison [2008] measure how much time it takes to perform maintenance tasks on an online shopping system. The results appeared to slightly favor the OO implementation over the AO implementation. Their approach to measure maintainability can be viewed as complementary to ours.

There are also empirical studies in AOP rest on qualitative investigation [Hanenberg & Unland, 2001; Koppen & Störzer, 2004; Griswold et al., 2006; Kästner et al., 2007; Munoz et al., 2008; Mortensen, 2009; Taveira et al., 2009; Taveira et al., 2010]. Our dissertation is a continuation of their work and further explores the impact of AOP on software evolvability and reusability. Hannemann & Kiczales [2002] developed AO implementations of the 23 Gang-of-Four (GoF) design patterns. For 12 out of all 23 patterns, they find reusable implementations. We build on their implementations as a starting point. Our research goes a step

further and shows how AO solutions can take advantage of generics and reflective programming. Using these techniques, we provide a highly reusable implementation of the Decorator, Proxy, and Prototype pattern.

Although we think aspects are best modeled with a new set of UML elements, several extensions exist as UML profiles [Evermann, 2007; Fuentes & Sanchez, 2007; Gao et al., 2004; Groher & Baumgarth, 2004; Groher & Schulze, 2003; Mosconi et al., 2008; Stein et al., 2002a; Stein et al., 2002b; Zakaria et al., 2002]. Our research draws inspirations from the work that bases on a heavy-weight extension mechanism [Lions et al., 2002; Hachani, 2003a; Hachani, 2003b; Kande, 2003; Yan et al., 2004]. Our extension is built on top of UML 2.2. This opposes with the proposals that base on the UML 1.x metamodel [Lions et al., 2002; Hachani, 2003a; Hachani, 2003b]. Moreover, differently from Hachani [2003a; 2003b], we do not modify the UML metamodel in any way. Furthermore, in contrast to [Hachani, 2003a; Hachani, 2003b; Yan et al., 2004] our metamodel provides dedicated icons for new elements.

1.7 Dissertation outline

Chapter 2 lays the foundations for understanding the central ideas of this dissertation. It focuses on concepts related to separation of concerns and modularization. This Chapter also gives an introduction to the problem of implementing crosscutting concerns in OO languages. The limitations of OO languages are explained and illustrated by two scenarios of adapting software to new requirements.

Chapter 3 illustrates how aspects can lexically separate the implementation of different concerns. It presents the state-of-the-art in implementing crosscutting concerns. The basic concepts of AOP and Composition Filters are explained and illustrated by two scenarios introduced in the previous Chapter. Section 3.3 provides a discussion on the AO modularization. It also highlights the emerging research efforts in restoring modular reasoning to AOP. An earlier version of this Section appeared in the proceedings of ICSOFT'10 [Przybyłek, 2010c].

Chapter 4 gives the definition of a new modelling language named AoUML that we elaborated to incorporate aspects into class diagram. AoUML is an extension to

the UML metamodel. It is used in the next Chapters to visualize the presented source code. This Chapter is based on our IMCSIT'08 paper [Przybyłek, 2008a].

Chapter 5 introduces metrics that we intend to apply to compare the paradigms with regards to software modularity, evolvability, and reusability. It also explains semantic dependencies in AO software to give a rationale for our coupling metric. The metrics discussed are derived from their OO counterparts. They are used in the next Chapters in our evaluation studies.

Chapter 6 presents a metrics-based comparison among AO and OO software with respect to coupling and cohesion. We evaluate the 23 GoF design patterns and 11 real-world systems. The major findings of the chapter were published in the proceedings of ENASE'10 [Przybyłek, 2010a] and ETAPS'11 [Przybyłek, 2011b].

Chapter 7 is divided into two parts, that were earlier published in the proceedings of BIR'08 [Przybyłek, 2008], ICSOFT'10 [Przybyłek, 2010b], and TOOLS'11 [Przybyłek, 2011c]. The first part of the Chapter presents a quasi-controlled experiment comparing evolvability and reusability between OO and AO implementations in 5 subsequent versions of the producer-consumer program. The second part of the Chapter explores the possibilities for improving implementations of the GoF design patterns using AspectJ with generics and reflective programming.

Chapter 8 summarizes the dissertation.

Chapter 2. Software modularity

I have a small mind and can only comprehend one thing at a time.
Dijkstra, 1972

The aim of this chapter is to review the approaches employed so far to modularize concerns, and to illustrate the need for new decomposition/composition mechanisms.

2.1 Criteria for software modularity

The evolution of programming languages is driven by the perennial quest for better separation of concerns (SoC). Subroutines, which encaptulated a unit of functionality, were invented in the early 1950s and all subsequent program structuring mechanisms such as procedures and classes, have been designed to provide better mechanisms for realising the SoC [Sommerville, 2010]. The term SoC was coined by Dijkstra [1974] and it means "*focusing one's attention upon some aspect*" to study it in isolation for the sake of its own consistency; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. In the context of systems development, this term refers to the ability to decompose and organize the system into manageable modules, which can be developed and maintained in relative isolation.

Dijkstra [1976] and Parnas [1972] suggested, that the best way to achieve SoC is through modularisation. Modularization is the process of decomposing a system into logically cohesive and loosely-coupled modules that hide their implementation from each other and present services to the outside world through a well-defined interface [Parnas, 1972; Yourdon & Constantine, 1979; Booch, 1994]. *Cohesion* is the "*intramodular functional relatedness*" and describes how tightly bound the internal elements of a module are to one another, whereas *coupling* is "*the degree of interdependence between modules*" [Yourdon & Constantine, 1979]. Modularization makes it possible to reason about every module in isolation, such

that when a small change in requirements occurs, it will be possible to go to one place in code to make the necessary modifications [Cline, 1998].

Modularization is closely related to composition and decomposition mechanisms in programming languages. Software composition and the reverse notion of software decomposition are about the partitioning of a software system into smaller parts (decomposition) and the assembly of software systems in terms of these smaller parts (composition) [Ostermann, 2003]. Thus, in practice, modularization corresponds with finding the right decomposition of a problem [De Win et al., 2002].

Herein, the term *module* is used as a generalization of procedure, function, class, interface, and aspect. A module consists of two parts: an interface and a module body (implementation). An interface presents the services provided by a module. It separates information needed by a client from implementation details. It represents a boundary across which control flow and data are passed. A module body is the code that actually realizes the module responsibility. It hides the design decisions and should not be accessible from outside the module. A programmer should be able to understand the responsibility of a module without understanding the module's internal design [Parnas, 1984]. The interface specification should be weaker than the implementation so that an interface allows multiple possible implementations and hence leaves room for evolution that does not invalidate the interface [Ostermann et al., 2011].

An interface as presented above is often termed provided interface. A module can also stipulate a so-called required interface, which is another module's provided interface. A required interface specifies the services that an element needs from some other modules in order to perform its function and fulfill its own obligations.

One of the first mentions of the importance of interfaces appeared in a 1970 textbook on systems development by Gouthier & Pont [1970]: "*At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules.*" Since then, this issue has been repeatedly headlined. Raymond advertizes [Raymond, 2003]: "*The only way to write complex software that won't fall on its face is to build it out of simple modules connected by well-defined interfaces, so that most problems are local*

and you can have some hope of fixing or optimizing a part without breaking the whole."

The interface and implementation parts are also called public and private, respectively. The users of a module need to know only its public part [Riel, 1996]. An interface serves as a contract between a module and its clients. Such contract allows the programmer to change the implementation without interfering with the rest of the program, so long as the public interface remains the same [Riel, 1996]. Parnas [1984] postulates that "*It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behavior of other modules. [...] Except for interface changes, programmers changing the individual modules should not need to communicate.*"

The paradigm that has made a significant contribution to improving software modularity is structured programming. Its origins date back to 1968, when the famous letter "GoTo statement considered harmful" [Dijkstra, 1968] was sent by Dijkstra to the Communications of the ACM. In this letter, Dijkstra calls for the abolishment of GoTo from high-level languages. He states that "*the unbridled use of the GoTo statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress*" (i.e. the state of the program). Next, Dijkstra presents the following program flow structures: sequence, selection, repetition and procedure call. When composing a program using these structures, the contents of the call stack and loop iteration stack are sufficient to determine the state of the program. Hence those contents make up a coordinate system according to which any trace of a program can be represented.

Other issues advocated by structured programming are: splitting a program into subsections with a single point of entry and exit, reducing reliance on global variables and information hiding.

The use of global variables is usually considered bad practice. Wulf & Shaw [1973] in their article "Global variable considered harmful" argue that global variables "*force upon the programmer the need for a detailed global knowledge of the program which is not consistent with his human limitations*". Since any code anywhere in a program can change the value of the variable at any time, understanding the use of the variable may entail understanding a large portion of the program.

Designing a module so that implementation details are hidden from other modules is called information hiding and was proposed by Parnas. In his paper, Parnas [1972] argues that the primary criteria for system modularization should focus on hiding critical design decisions (i.e. difficult design decisions or design decisions which are likely to change). Similar postulates were later put forward in the context of OOP: "*The main idea is to organize things so that when changes to the concept or abstraction occur (as is inevitable), it will be possible to go to one place to make the necessary modifications*" [Cline, 1998]. In the programming community, information hiding has become such an undisputed dogma of modularity that Brooks [1995] even felt that he had to apologize to Parnas for questioning it [Ostermann et al., 2011].

Parnas [1972] also enumerates the benefits expected of modularization: (1) managerial – development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility – it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility – it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood. This comprehensibility is often termed *modular reasoning*. Clifton & Leavens clarify [2003] that a language supports modular reasoning if the actions of a module M written in that language can be understood based solely on the code contained in M along with the signature and behavior of any modules referred to by M. A module M refers to N if M explicitly names N, if M is lexically nested within N, or if N is a standard module in a fixed location (such as Object in Java).

Meyer [1989] summarizes the research on software modularity by enumerating the essential requirements for modular design: (1) decomposability - means that a system can be and is decomposed into a set of cohesive and loosely coupled modules; (2) composability - demands that every module may be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed; (3) understandability - means that each single module is understandable on its own; (4) continuity - describes that a small change in requirements leads to a small change in limited parts of the system and does not affect the architecture; (5) protection - demands that the effect of errors be limited to one little part of a system. Meyer [1989] also postulates five rules which we must observe to ensure modularity: (1)

Direct Mapping - the modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain; (2) Few Interfaces - every module should communicate with as few others as possible; (3) Small Interfaces - if two modules communicate, they should exchange as little information as possible; (4) Explicit Interfaces - whenever two modules A and B communicate, this must be obvious from the text of A or B or both; (5) Information Hiding - the designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.

2.2 From structured to object-oriented programming

The term structured programming was coined to describe a style of programming that merges the ideas proposed in the late 1960s and early 1970s by:

- Dijkstra: SoC, layered architecture, structured control constructs;
- Wirth: stepwise refinement, modular programming;
- Parnas: information hiding, modular programming;
- Hoare: designing data structures;
- Knuth: local variables, literate programming.

In the past, the structured paradigm proved to be successful for tasks, such as controlling petroleum refining facilities and providing worldwide reservation systems. However, as software grew in size, inadequacies of the structured techniques started to become apparent, and the OOP was proposed by Dahl and Nygaard as a better alternative. Since the late 1980s OOP has been the mainstream of software development.

OOP was created from a desire to close correspondence between objects in the real world and their counterparts in software. The object-oriented purism comes from the dogma that everything should be modeled by objects, because human perception of the world is based on objects. An *object* is a software entity that combines both state and behavior. An object's behavior describes what the object can do and is specified by a set of operations. The implementation of an operation is called a *method*. The way that the methods are carried out is entirely the responsibility of the object itself [Schach, 2007] and is hidden from other parts of the program (Larkin & Wilson 1993). An object performs an operation when it

receives a message from a client. A message is a request that specifies which operation is desired. The set of messages to which an object responds is called its message interface [Hopkins & Horan, 1995]. An object's state is described by the values of its attributes (i.e. data) and cannot be directly accessed from the outside. The attributes in each object can be accessed only by its methods. Because of this restriction, an object's state is said to be encapsulated. The advantage of encapsulation is that as long as the external behavior of an object appears to remain the same, the internals of the object can be completely changed [Hunt, 1997]. This means that if any modifications are necessary in the implementation, the client of the object need not be affected.

In OO software development, a system is seen as a set of objects that communicate with each other by sending messages to fulfil the system requirements. The object receiving the message may be able to perform the task entirely on its own (i.e. access the data directly or use its other method as an intermediary). Alternatively, it may ask other objects for information, or pass information to other objects [Hopkins & Horan, 1995].

The most popular model of OOP is a class based model. In this model, an object's implementation is defined by its *class*. The object is said to be an instance of the class from which it was created. A class is a blueprint that specifies the structure and the behaviour of all its instances. Each instance contains the same attributes and methods that are defined in the class, although each instance has its own copy of those attributes.

OO languages offer two primary reuse techniques: inheritance and composition. Software reuse refers to the development of software systems that use previously written modules. *Inheritance* allows for reusing an existing class in the definition of a new class. The new class is called the derived class (also called subclass). The original class from which the new class is being derived is called the base class (also called superclass). All the attributes and methods that belong to the base class automatically become part of the derived class [Cline et al., 1998]. The subclass definition specifies only how it differs from the superclass [Larkin & Wilson, 1993]; it may add new attributes, methods, or redefine (override) methods defined by the superclass.

An object of a derived class can be used in every place that requires a reference to a base class [Cline et al., 1998]. It allows for dispatching a message depending not only on the message name but also on the type of the object that

receives the message. Thus, the methods that matches the incoming message is not determined when the code is created (compile time), but is selected when the message is actually sent (run time) [Hopkins & Horan, 1995]. An object starts searching the methods that matches the incoming message in its class. If the method is found there, then it is bound to the message and executed, and the appropiate response returned. If the appropiate method is not found, then the search is made in the instance's class's immediate superclass. This process repeats up the class hierarchy until either the method is located or there is no further superclass [Hopkins & Horan, 1995]. The possibility that the same message, sent to the same reference, may invoke different methods is called *polymorphism*.

A new class can be composed from existing classes by *composition*. Composition is the process of putting an object inside another object (the composite) [Cline et al., 1998]. A composite can delegate (re-direct) the requests it receives to its enclosing object. Composition models the has-a relationship. It is claimed that composition is more powerful than inheritance, because (1) composition can simulate inheritance, and (2) composition supports the dynamic evolution of systems, whereas inheritance relations are statically defined relations between classes [Bergmans, 1994].

Inheritance is also called "white box" reuse, because internals of a base class are visible to its extensions. In contrast, composition is called "black box" reuse, because the internals of the enclosed object are not visible to the enclosing object (and vice-versa) [Oprisan, 2008]. With composition, an enclosing object can only manipulate its enclosed object through the enclosed object's interface. Because composition introduces looser coupling between classes it is preferable to inheritance.

Developing high quality software requires knowledge usually learned only by experience [Gamma et al., 1995; Albin-amiot & Guéhéneuc, 2001]. Experience acquired in projects that have worked in the past allows a designer to avoid the pitfalls of development [Kuhlemann, 2007]. Over the years, the wisdom about OO software development had been accumulated into what are known as design patterns and then catalogued by Gamma et al. in what is known as the "Gang of Four" book [Gamma et al., 1995].

A design pattern is a general solution that addresses a recurring problem encountered in software development [Hannemann & Kiczales, 2002]. It constitutes a set of guidelines that describe how to accomplish a certain task in a

specific design situation [Pressman, 2005]. A design pattern also identifies classes that play a role in the solution to a problem and describes their collaborations and responsibilities. However, with OO techniques, only the solutions of the patterns are considered reusable. As a consequence the programmer still has to implement the patterns for each application he is constructing [Borella, 2003].

2.3 Tyranny of the dominant decomposition

When solving a simple problem, the entire problem can be tackled at once. For solving a complex problem, the basic principle should be divided into easier to comprehend pieces, so that each piece can be conquered separately [Jalote, 2005]. Programming languages provide mechanisms that allow the programmer to break a system down into modules of behavior or function, and then compose those modules in different ways to produce the overall system [Kiczales et al., 1997]. Although the exact nature of the decomposition unit differs between the structured and OO paradigm, in each case, it feels comfortable to talk about what is encapsulated as a functional unit of the overall system [Kiczales et al., 1997]. Therefore, both decomposition techniques can be generally treated as functional decomposition.

The manner in which a system is physically divided into modules can affect significantly the structural complexity and quality of the resulting system [Parnas, 1972; Yourdon & Constantine, 1979]. Dahl, Dijkstra & Hoare [1972] explain that "*good decomposition means that each module may be programmed independently and revised with no, or reasonably few, implications for the rest of the system.*" Yourdon & Constantine suggest [Yourdon & Constantine, 1979] to decompose a system so that (1) highly interrelated parts of the system should be in the same module; (2) unrelated parts of the system should reside in different modules. According to Yourdon & Constantine [1979] "*What we are striving for is loosely coupled system - that is, a system in which one can study (or debug, or maintain) any one module without having to know very much about any other modules in the system.*" Although the different modules of one system cannot be entirely independent of each other, as they have to cooperate and communicate to solve the

larger problem, the design process should support as much independence as possible [Jalote, 2005].

Implementation and maintenance costs generally will be decreased when each piece of the system corresponds to exactly one small, well-defined piece of the problem, and each relationship between a system's pieces corresponds only to a relationship between pieces of the problem [Yourdon & Constantine, 1979]. Kiczales et al. [1997] found that the abstractions offered by functional decomposition are insufficient to express crosscutting concerns in a modular way. In his PhD dissertation, Ostermann [2003] ilustrates this problem graphically on abstract concern space (Figure 2.1). Each figure represents a particular concern of a software system. There are three options for organizing this space: by size, by shape, or by color. Each of these decompositions is equally reasonable, but they are not hierarchically related [Ostermann, 2003].

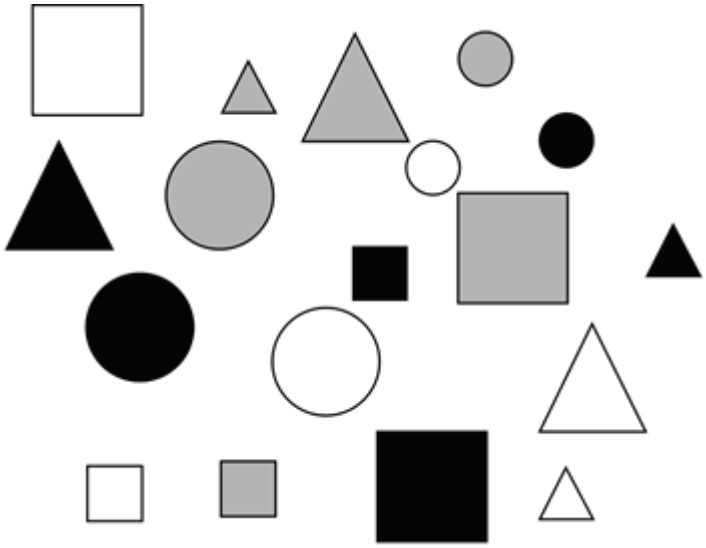


Figure 2.1. Abstract concern space

With a functional decomposition, one fixed classification sequence has to be choosen. In Figure 2.2, the classification sequence is color, shape, size. The problem with such a fixed classification sequence is that only the first element of the list is localized whereas all other concerns are tangled in the resulting hierarchical structure [Mezini & Ostermann, 2004]. Figure 2.2 illustrates this with the concern “circle”, whose definition is scattered around the color-driven decomposition [Ostermann, 2003]. Only the color concern is cleanly separated into white, grey, and black, but even this decomposition is not satisfactory because the color concern is still tangled with other concerns [Mezini & Ostermann, 2004].

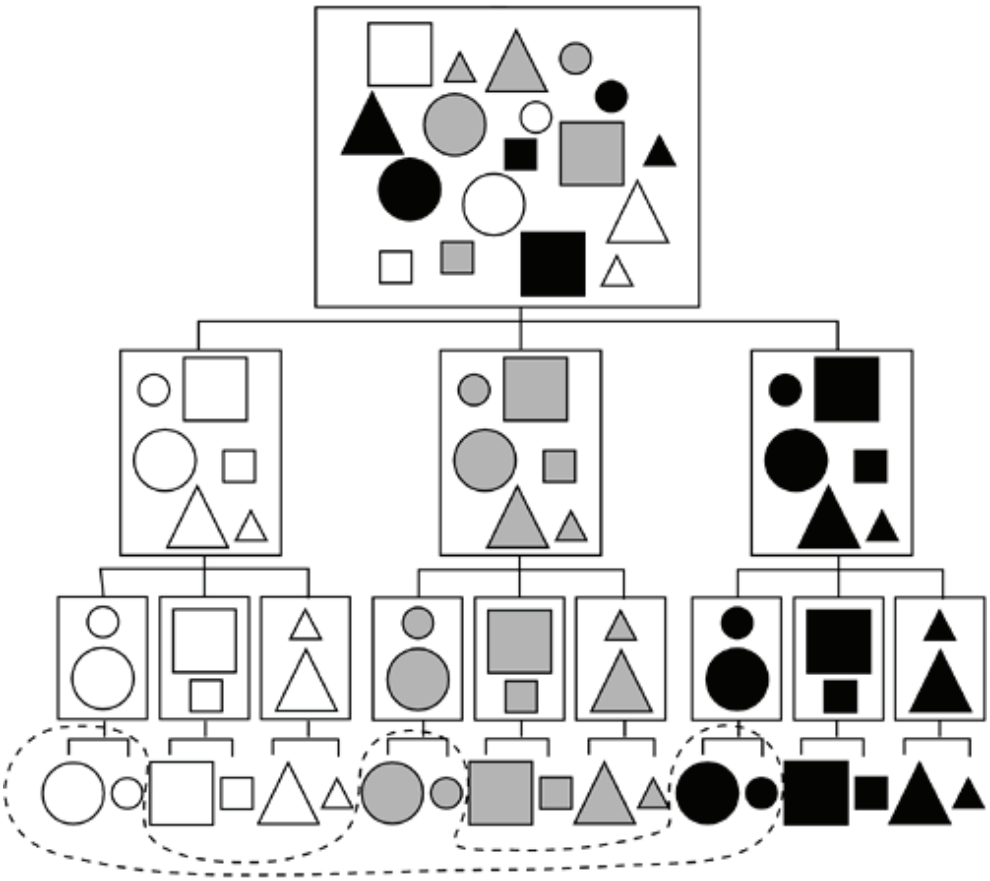


Figure 2.2. Arbitrariness of the decomposition hierarchy

The presented problem is known as the *tyranny of the dominant decomposition* and it means that traditional programming languages generally support only a single "dominant" decomposition at a time. This dominant decomposition satisfies some important needs, but usually at the expense of others [Tarr et al., 1999]. In the result, no matter how well a system is decomposed, the implementation of crosscutting concerns will cut across the chosen decomposition [Mens et al., 2004] causing code tangling and code scattering. *Code tangling* occurs when a module implements multiple concerns. *Code scattering* occurs when similar pieces of the implementation of one concern appear in multiple modules in the program. Tangling and scattering negatively affect source code quality. For example, tangling reduces comprehensibility, as one has to know which statements belong to which concern, and this may not always be obvious [Durr, 2008]. Moreover, whenever a concern needs to be changed, a developer has to localize the code that implements it. This may possibly require him to inspect many different modules, since the code may be scattered across several of them [Bruntink et al., 2004]. Furthermore, tangling reduces maintainability, because updating one

concern may break surrounding code related to other concerns. Tangling also makes it harder to reuse a module, since the module addresses several concerns, and a software designer may wish not to reuse all of them at the same time [Havinga, 2009].

2.4 Impact on maintainability and reusability

Composing systems from existing modules rather than building from scratch has been one of the main goals of the software engineering since its beginning in the 1960s [McIlroy, 1968]. Reusability is the ease with which existing modules can be used in new context [Peters & Pedrycz, 2000]. Using previously written modules as building blocks allows programmers to simplify the construction of software, since the traditional phases of development are replaced with processes of module search and selection [Andrews et al., 2002]. Such approach reduces the development time and costs, downgrades the risk of new projects, and improves the software quality. One of the obstacles to a massive application of software reuse in industrial environments is that creating reusable software modules requires a huge initial investment which is not rapidly amortized.

When development of a software product is complete and it is released to the market, it enters the maintenance phase of its life cycle [Kan, 2002]. Software maintainability is the ease with which a software product can be modified after delivery [IEEE, 1990; Pressman, 2005]. ISO/IEC [1999] defines four categories of maintenance: perfective, adaptive, corrective, and preventive. As software is used, the user usually requests additional features and capabilities [Lewis, 2004]. *Perfective maintenance* extends the software beyond its original requirements. Over time, the original environment (terminal devices, operating system, laws, regulations, business rules, external product characteristics) for which the software was developed is likely to change. *Adaptive maintenance* accommodates the software to its external environment [Pressman, 2005]. It has been estimated that 80% of the software maintenance effort is devoted to software evolution (adaptive and perfective maintenance) [Pigoski, 1997].

Even with the best quality assurance activities, it is likely that the delivered software contains some latent defects that were not detected during testing. *Corrective maintenance* repairs these defects. Computer software deteriorates due

to change, and because of this, *preventive maintenance*, often called software reengineering, must be conducted to enable the software to operate effectively and to make subsequent maintenance easier. In essence, preventive maintenance refers to enhancements to software modularity or understandability. It may also include the study of a system to detect and correct latent faults in the software product before they become effective faults [ISO/IEC, 1999].

Software maintenance has been recognized as the most costly and difficult phase in the software life cycle [Li & Henry, 1995]. Studies and surveys over the years have indicated that software changes typically consume 40% to 80% of overall software development costs [Lientz, 1978; Zelkowitz, 1978; Boehm, 1981; Meyers, 1988; Yourdon, 1992; Hatton, 1998; Glass, 2002; Pressman, 2005]. Hewlett-Packard estimates that 60% to 80% of its R&D personnel are involved in maintaining existing software, and that 40% to 60% of software budget are directly related to maintenance [McKee, 1984; Coleman et al., 1994].

Software modularity, maintainability and reusability are closely related. Much academic work asserts a relationship between the design of a system and the manner in which this system evolves over time [MacCormack et al., 2007]. In particular, modularity creates "options" to adapt a design to meet unforeseen future requirements [Baldwin & Clark, 2000]. Moreover, the more connections between modules, the more dependent they are and the harder it is to reuse them in different contexts. Table 2.1 enumerates work that documented these relationships.

Table 2.1 Impact of coupling and cohesion on reusability and maintainability

	reusability	**maintainability**
coupling	[Hitz & Montazeri, 1995; Bowen et al., 2007]	[Hitz & Montazeri, 1995; Chaumun et al., 2000; Bowen et al., 2007; MacCormack et al., 2007; Breivold et al., 2008]
cohesion	[Bieman & Kang, 1995; Bowen et al., 2007]	[Bowen et al., 2007; Perepletchikov et al., 2007]

2.5 Weaknesses of object-oriented programming – running examples

2.5.1 Example 1 - mathematics software

Consider some mathematics software that is implemented in Java. It consists of many classes and Matrix (Figure 2.3) is one of them. The core concern here is to support mathematical operations. Let's assume that we have a new requirement. We would like to log every method call in the system and how long it takes to execute the method.

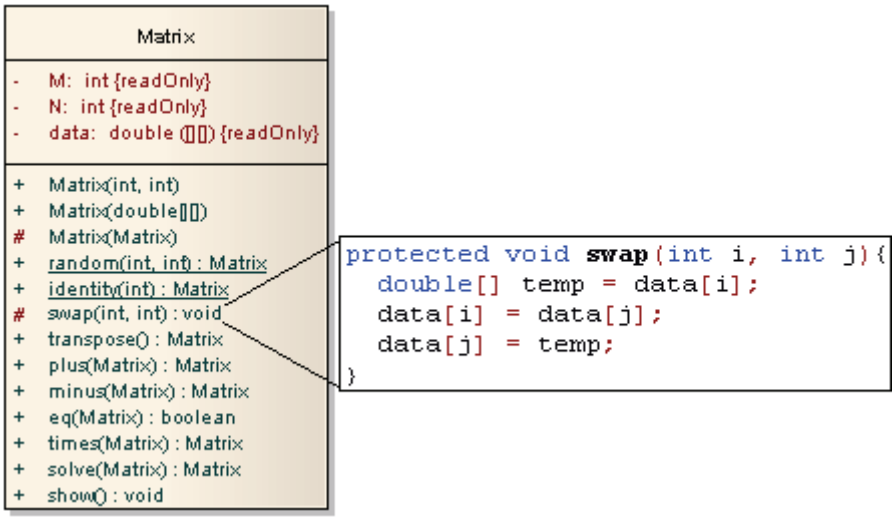


Figure 2.3. The Matrix class

One of the possible ways of OO solution requires embedding the logging code in every method. For example, Listing 2.1 shows how the swap method is instrumented to mesure its execution time.

```
public class Matrix {
  //...
  protected void swap(int i, int j) {
    long start = System.currentTimeMillis();
    double[] temp = data[i];
    data[i] = data[j];
    data[j] = temp;
    long end = System.currentTimeMillis();
    long time = end - start;
    System.out.println("void Matrix.swap(int,int) - "+ time);
  }
  //...
}
```

Listing 2.1 The Matrix::swap() method with time logging

The code associated with the logging concern is shown as shaded code. Such instrumentation is very invasive and breaks the open-closed principle, which states that modules should be open for extension, but closed for modification. Moreover, the log statements are tangled with the operation's core logic and the similar code is scattered across every method in the system.

An alternative approach is to define a new class that extends Matrix and then to wrap each super method call with log statements (Listing 2.2). Moreover, every place in the code where Matrix is instantiated must be replaced by LogMatrix. While this approach limits code tangling, code scattering is still present. Parts of the implementation of logging are replicated in several places. Keep in mind that the application contains hundreds of classes and a new log class is needed for every original class.

```
public class LogMatrix extends Matrix {
  //...
  protected void swap(int i, int j) {
    long start = System.currentTimeMillis();
    super.swap(i,j);
    long end = System.currentTimeMillis();
    long time = end - start;
    System.out.println("void Matrix.swap(int,int) - "+ time);
  }
  public LogMatrix transpose() {
    long start = System.currentTimeMillis();
    LogMatrix m = new LogMatrix(super.transpose());
    long end = System.currentTimeMillis();
    long time = end - start;
    System.out.println("Matrix Matrix.transpose() - "+ time);
    return m;
  }
  //...
}
```

Listing 2.2 Logging in the subclass

2.5.2 Example 2 – Learning Management System

A learning management system (LMS) is deployed at a college. It is suspected that unauthorised persons know lecturers' passwords and have modified data in the system. As a result, it has been decided that all data-changing SQL statements have to be watched. In order to do this, it has been proposed that the existing application be extended by logging every DML statement that modifies the records, together with the date of the incident, the database username, and the login name.

The application uses JDBC to access its database (Figure 2.4). The key elements of JDBC API (in terms of the example presented) are the Connection

interface, the Statement interface and the DriverManager class. DriverManager manages all the details involved in establishing the connection to a specified database. The established connection is returned by DriverManager::getConnection(..) which is a static method. In a typical application scenario, the next step is creation of a Statement object. The Connection::createStatement() method is called upon to do this. The Statement object is associated with an open connection and used to send SQL statements to the database. DML statements such as INSERT, UPDATE, DELETE are usually executed using Statement::executeUpdate(..).

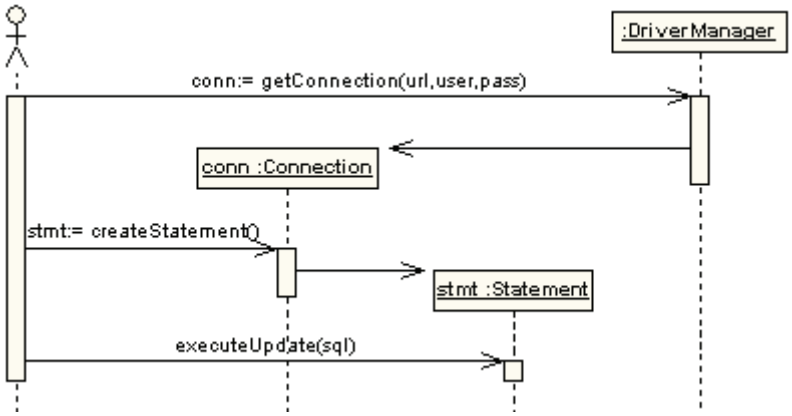


Figure 2.4 A typical usage scenario for accessing a database

An OO solution is based on a delegation model. The LogStatement class is responsible for tracing SQL queries (Listing 2.3). It implements the Statement interface and aggregates the statement object. All messages specified by the Statement interface are delegated to the statement object. Those which are able to modify records (i.e. execute, executeUpdate) are redefined in order to implement tracing. Moreover, to fulfil the new requirement, every object returned by Connection::createStatement() has to be wrapped with LogStatement. LogStatement implements 40 methods. Of these 1 is a constructor, 29 methods are used for forwarding only, 8 methods call the log method and then forrward the original message to Statement object, and 2 methods implement logging. It is apparent that logging is scattered through and tangled with the business logic.

```
public class LogStatement implements Statement {
  private Statement delegate;
  public LogStatement(Statement st) {
    delegate = st;
  }
  private boolean isDML(String sql) {
    String tmp = sql.toUpperCase();
    return tmp.indexOf("UPDATE")>=0 ||
     tmp.indexOf("INSERT")>=0 || tmp.indexOf("DELETE")>=0;
  }
  private void log(String sql) throws SQLException {
    String dbUser =
     delegate.getConnection().getMetaData().getUserName();
    String login = User.getCurrentUser().getLogin();
    System.out.println(
     new Date() + "; " + login + "; " + dbUser + "; " + sql);
  }
  public void cancel() throws SQLException {
    delegate.cancel();
  }
  public boolean execute(String sql) throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql);
  }
  public boolean execute(String sql, int autoGeneratedKeys)
   throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql, autoGeneratedKeys);
  }
  public boolean execute(String sql, int[] columnIndexes)
   throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql, columnIndexes);
  }
  public boolean execute(String sql, String[] columnNames)
   throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql, columnNames);
  }
  public int executeUpdate(String sql) throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.executeUpdate(sql);
  }
  //other methods specified by the Statement interface
}
```

Listing 2.3 LogStatement class definition

2.6 Summary

Modularity is a key concept that programmers wield in their struggle against the complexity of software systems. Although modules have taken many forms over the years from functions and procedures to classes, no form has been capable of

expressing a crosscutting concern in a modular way. The term crosscutting concern refers to an aspect of the system that cannot be cleanly modularized because of the limited abstractions offered by the underlying programming language. In a traditional environment, implementing crosscutting concerns usually results in code scattering and code tangling. The presented examples illustrate that a crosscutting concern cannot be directly implemented by dedicated classes. Instead, its implementation usually spreads over the whole system and cuts across the implementation of all other concerns. Maintaining a crosscutting concern means modifying each fragment of the scattered code realizing that concern. Therefore, increasing the maintenance cost and error proneness [Eaddy et al., 2008; Munoz et al., 2008]. This chapter is partly based on work published in [Przybyłek, 2007] and [Przybyłek, 2009].

Chapter 3. Aspect-oriented programming

If the only tool you have is a hammer, then everything looks like a nail.

The aim of this chapter is to outline the background material necessary to understand the state-of-the-art in implementing crosscutting concerns, and to provoke thoughts about the concepts behind AOP.

3.1 Basic concepts

AOP dates back to 1997, when it grew out of the research work undertaken by Kiczales et al. [1997] at Xerox PARC (Palo Alto Research Center). AOP appeared as a reaction to the problem of dominant decomposition. The aim of AOP is to improve SoC by providing a new unit of decomposition called an *aspect* and new ways of composition. Aspects allow the secondary concerns to be implemented in self-contained modules. The composition of aspects and classes is implemented through new programming mechanisms, such as pointcuts, advices and introductions.

Traditionally, many aspect languages have been implemented as an extension to existing languages. This, in most cases, leads to a conceptual distinction between the "aspect code" and the "base code" [Havinga, 2009]. Once implemented, both base code and aspect code are combined together to produce a final system [Burrows et al., 2010]. The process of actually inserting the aspect code into the code of other modules at the appropriate points is known as *weaving*. Weaving is typically performed at compile-time.

AspectJ was the first AO language and remains the most complete and successful AOP implementation available to date, with AJDT as a production ready IDE. When AspectJ reached a level of stability, it was relocated from Xerox PARC into the arms of the open source community. AspectJ is now maintained as an Eclipse Technology project. Since AspectJ is becoming a de facto standard, we have chosen it as a referral language in this dissertation. AspectJ is implemented as

a weaver that extends the Java compiler. It produces bytecode that runs inside the JVM just like any other java class. Figure 3.1 explains the weaving process.

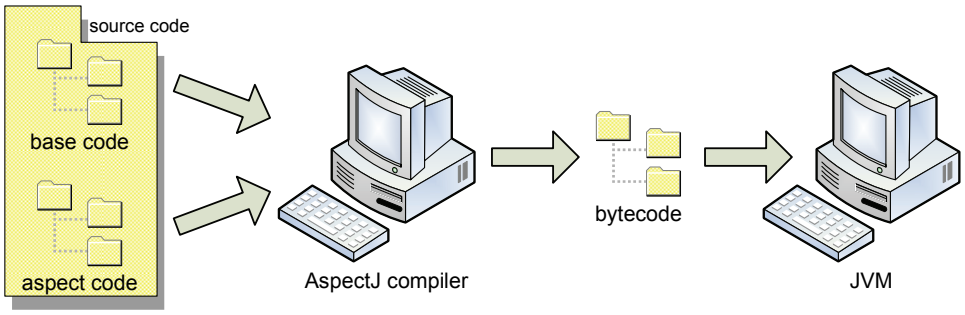


Figure 3.1 Build process with AspectJ

In AspectJ, an aspect can, like a class, realize interfaces, extend other module and declare attributes and operations. In addition, it can declare advices, pointcuts, and inter-type declarations. A *pointcut* is a language construct designed to specify a set of join-points and obtain the context surrounding the join-points as well. A *joinpoint* is an identifiable location in the program flow where the implementation of a crosscutting concern can be plugged in. Typical examples of joinpoints include a throw of an exception, a call to a method, and an object instantiation. An *advice* is a method-like construct used to define an additional behaviour that has to be inserted at all joinpoints picked out by the associated pointcut. An advice is implicitly triggered when specific events (e.g. a method call or a field access) occur during the program execution. The body of an advice is the implementation of a crosscutting concern. An advice is able to access values in the execution context of the pointcut. Depending on the type of advice, whether “before”, “after” or “around”, the body of an advice is executed before, after or in place of the selected joinpoints. An around advice may cancel the captured call, may wrap it or may execute it with the changed context. An inter-type declaration is used to crosscut the static-type structure of classes and their hierarchies. It allows a programmer to add attributes and methods to an established class, from outside the original class definition. Inter-type declarations can also declare that other types implement new interfaces or extend a new class. The reader interested in a comprehensive explanation of AspectJ is referred to [Laddad, 2003; Colyer et al., 2004; Miles, 2004; Gradecki & Lesiecki, 2003].

3.2 Running examples

3.2.1 Example 1 - mathematics software

In the OO implementation, logging was scattered through every method in the system (see Listing 2.2). All we have to do in AOP is to define a single aspect that implements the new requirement (Listing 3.1). The eachMethod() pointcut uses wildcards to capture all method executions regardless of their name, regardless of the name of the class on which the method is defined, and regardless of the return type. The around() advice is activated whenever the eachMethod() pointcut captures something. The logging code is injected around the execution of each method. AspectJ provides the proceed() statement to execute the original method. The thisJoinPoint pseudo-variable is used to expose a context from the join-point that triggers the advice.

```
public aspect TimeLogging {
  pointcut eachMethod(): execution(* *.*(..));
  Object around(): eachMethod() {
    long start = System.currentTimeMillis();
    Object tmp = proceed();
    long end = System.currentTimeMillis();
    long time = end - start;
    // we take the signature of the original method
    Signature sig = thisJoinPointStaticPart.getSignature();
    System.out.println(sig +" - "+ time);
    return tmp;
  }
}
```

Listing 3.1 The TimeLogging aspect

3.2.2 Example 2 – learning management system

The OO implementation tangled the secondary concern with the core logic (see Listing 2.3). A better solution can be obtained by using AOP and implementing a new functionality as an aspect (Listing 3.2). The fundamental difference between the OO and AO solution is lexical separation of the tracing code. The joinpoints at which tracing should be injected are specified as the places where the update methods are called.

```
public aspect DMLmonitoring {

  pointcut DMLexecute(String sql, Statement st): target(st)
   && call( * Statement.execute*(..) ) && args(sql,..);

  after(String sql,Statement st) returning (Object o):
   DMLexecute(sql,st) {
    String login = User.getCurrentUser().getLogin();
    String dbUser = "";
    try {
      dbUser=st.getConnection().getMetaData().getUserName();
    } catch (SQLException e) {};
    String tmp = sql.toUpperCase();
    if( tmp.indexOf("UPDATE")>=0 || tmp.indexOf("INSERT")>=0
     || tmp.indexOf("DELETE")>=0 ) System.out.println(
     new Date()+"; "+login+"; "+dbUser+"; "+sql);
  }
}
```

Listing 3.2. DMLmonitoring aspect definition

3.3 Aspects vs Modularization

3.3.1 AOP promotes unstructured programming

Constantinides et al. [2004] show that AOP has some of the problems associated with the GoTo statement. In particular, it does not allow for creating a coordinate system for the programmer. Since an advice can plug into just about any point of execution of a program, one can never know the previous (or following) statement of any statement [Steimann, 2006]. An advice is even worse than GoTo as the GoTo statement transfers control flow to a visible label, while an advice does not. As a result, just looking at the source code of the base module is not enough to deduce a variable value – an advice might have changed it invisibly for the programmer. Constantinides et al. compare Advice to the ComeFrom statement, which was proposed as a way to avoid GoTo – of course only as a joke [Constantinides et al., 2004].

3.3.2 AOP breaks information hiding

A well designed module hides its implementation details from other modules. Prior to AOP, public interfaces together with private implementations guaranteed that changing a module's implementation would not break other modules as long as the interface would be kept the same. Since AOP this is no longer true. Aspects have the ability to interject functionalities at any joinpoint in the entire program,

effectively bypassing class interfaces. Aldrich [2005] tightens this problem by restricting quantification, in that internal communication events (e.g., private calls within a module) cannot be advised by external clients.

In addition, an aspect can access the private members of any module by using the privileged modifier. In turn, it leads to a globalization of the data contained in modules. Hence, the conclusion drawn by Wulf & Shaw [1973] – that in the presence of global variables a programmer needs "*a detailed global knowledge of the program*" – is therefore also true for the presence of aspects [Steimann, 2006]. Moreover, the ability of aspects to access unrestrictedly the base code can invalidate some important properties of the system by modifying the program flow or leaving protected data structures in an inconsistent state [Munoz et al., 2007].

3.3.3 AOP leaves interfaces implicit

Steimann [2006] tries to apply the idea of provided/required interfaces to AOP. On the one hand, the aspect provides a particular service through which it extends the base module; therefore it should specify the provided interface. However, the matching required interface of the base module remains implicit – the base module does not specify that it needs something. On the other hand, the base module provides a set of program elements, which are required by the aspect to perform its function. Although the aspect depends on these elements, the base module comes without an explicit counterpart interface specification: its provided interface is implicit. Seen either way, the base module specifies no interfaces that could be matched with those of its aspects [Steimann, 2006]. The lack of interfaces makes aspects vulnerable to any changes in the classes to which they apply [Ongkingco et al., 2006]. For the programmer of the base module, this means that everything accessible for aspects should be kept constant. Otherwise, aspects can break down as classes evolve. Ferrari et al. [2010] measured the impact of the obliviousness property on the fault-proneness of the evaluated systems. They found that the lack of awareness between base and aspectual modules tends to lead to incorrect implementations.

The efforts of introducing an explicit interface between aspects and base modules were originated by Gudmundson & Kiczales (G&K) and then continued by Aldrich. Gudmundson & Kiczales [2001] notice that the signature (a name and parameterization) of a pointcut can convey the abstract responsibility captured by the pointcut definition. As such pointcuts provide a basis for a new kind of

interface, which Gudmundson & Kiczales call the pointcut interface. A pointcut interface consists of a collection of named pointcuts and is exported by the base module, which can be a class or a package. The pointcut definition is kept within the module that exports the interface, so anyone looking at the definition would also be looking at the implementation of the base module. By having the exported pointcut, the programmer is aware that the base module may be influenced by aspects. Preserving the pointcut interface guarantees that upgrades to the base module will not disturb the dependent aspects.

Aldrich [2005] introduces a new modularization unit - Open Module - that "*is intended to be open to extension with advice but modular in that the implementation details of a module are hidden*". In this approach, modules (the word "module" here bearing a meaning distinct from common usage) have to export these join points that can be captured by the aspects that are external to the module. Since an advice queries exported pointcuts in order to achieve its function, the pointcuts can be thought of as a provided interface, while its counterpart in the advice header as a required interface. In addition, all calls to interface methods from outside the open module can also be advised. This property is important because many aspects rely only on calls to interface methods, so exporting pointcuts for all of these calls would be cumbersome. The main drawbacks of Open Modules are: (1) Explicitly exposing an interface pointcut means a loss of some obliviousness; (2) The programmer of the base module must anticipate that clients might be interested in the internal event; (3) The programmer has to hide out some implementation details of the designed module to make the module open for advising; (4) When pointcuts are defined within base modules, many join points that have to be advised in the same way cannot be captured by quantified pointcuts, e.g., using wild-card notations. A separate pointcut is required for each base module.

Leavens & Clifton [2007] introduce a required interface in the base module by explicitly naming the aspects that may affect the module behaviour. Then, aspects can only be applied to the modules that reference them. Explicit acceptance of an aspect can be expressed by an annotation.

Hoffman & Eugster [2007] extend AspectJ with explicit join points (EJPs). EJPs introduce a new type of join point, which is explicitly declared by the programmer within aspect, has a unique name and signature. Base code then explicitly references these join points where crosscutting concerns should apply.

The idea of EJPs is to represent cross-cutting concerns via explicit interfaces that act as mediators between aspects and base code.

The placement of interface between the aspect and the base code breaks the obliviousness property of aspects and make the base code aware about the existence of aspects [Munoz et al., 2007]. In addition, as was pointed out by Steimann [2006], both the above solutions not only make advice activation almost indistinguishable from guarded subroutine calling but also they re-introduce the scattering that AOP was to avoid. For instance, with tracing as a crosscutting concern, annotating every method whose execution is to be traced is just as annoying as adding the tracing code on site [Steimann, 2006]. Thus, the use of annotations has potential scaling problems. In addition, this technique is invasive for base modules and unfeasible in case base modules are third party components.

Sullivan et al. [2005] suggest the introduction of design rules that govern how code has to be written to expose specific points in program execution. These rules are documented in so called "crosscutting interface" (XPI) specifications that base code designers "implement" and that aspects may depend upon. Once these interfaces are defined, designers can develop aspect and base code independently [Sullivan et al., 2005]. However, as Steimann [2006] points out, "*specification of the XPI requires an a priori decision what the crosscutting behavior of a system is*". To address this, Sullivan et al. designed their XPIs by asking the question [Sullivan et al., 2005] - "*what constraints on the code would shape it to make it relatively easy to write the aspects at hand, as well as support future aspects?*" Although XPIs do not define a concrete interface, they also violate obliviousness by defining a coordinate coding style between aspects and base code [Munoz et al., 2007]. As mentioned in Chapter 1, obliviousness requires that the underlying system does not make assumptions of any kind about the possible aspects that may be applied [Katz, 2004].

3.3.4 AOP makes modular reasoning difficult

Aspects are most effective when the code they advise is oblivious to their presence [Filman, 2001]. In other words, aspects are effective when programmers of the underlying system does not have to prepare any hooks or annotations [Dantas & Walker, 2006]. However, obliviousness also implies that a base module has no knowledge of which aspects modify it where or when [Steimann, 2006]. It

conflicts with the ability to study the system one module at a time. When studying some module, one needs to consider all aspects that can possibly interfere and change the module's logic [Recebli, 2005]. The need of global analysis is a sign of being unmodular.

A proposal to maintain modular reasoning was put forward by Clifton & Leavens [2002] (C&L) and then continued by Dantas & Walker [2006] and Recebli [2005]. Not all violations of encapsulation by aspects are harmful – otherwise AOP would be useless [Recebli, 2005]. C&L [2002] propose to divide aspects into two categories, assistants and spectators, which provide complementary features. *Assistants* have the full power of AspectJ's aspects, but to maintain modular reasoning it is required that assistants are explicitly accepted by a module (see Section 3.3.2). *Spectators* are constrained to not modify the behavior of the modules that they advise. In concrete terms, a spectator may only mutate the state that it owns and it must not change the control flow to or from an advised method. In addition to mutating the owned state, it seems reasonable to allow spectators to change accessible global state as well, since a Java module cannot rely on that state not changing during an invocation (modulo synchronization mechanisms) [Clifton & Leavens, 2002]. The claim is that spectators are safe to ignore in reasoning about the base-code as they do not influence its specification. Nevertheless, when problems arise, a programmer must examine both the base and relevant aspect code to identify a bug. Moreover, the C&L's approach breaks the obliviousness property in a such way that the base code is aware about specific aspect advising it.

Recebeli [2005] proposes the notion of *aspects purity* to reduce the harm that aspects can do. A pure aspect is one that promises not to alter the behaviour of specified set of modules, only possibly adding something new. Nevertheless, pure aspects only assure harmless when aspects have pure intentions, but giving no assurance with other aspects.

Munoz et al. [2008] propose a framework for specifying the expected interactions between aspect and the base program. Aspects are specified with the invasiveness patterns they realize (Table 3.1), and the base program with assertions allowing or forbidding invasiveness patterns. In AspectJ, aspects crosscut the base program at two levels. At method level advices manipulat the method's behavior and at module level inter-type declarations modify the program structure.

Table 3.1 Classification of invasiveness patterns [Munoz et al., 2008]

classification element	**description**	**invasiveness level**
Augmentation	After crosscutting, the body of the intercepted method is always executed. The advice augments the behavior of the method it crosscuts with new behavior that does not interfere with the original behavior. Examples of this kind of advices are those realizing logging, monitoring, traceability, etc.	method
Replacement	After crosscutting, the body of the intercepted method is never executed. The advice completely replaces the behavior of the method it crosscuts with new behavior. This kind of advices eliminate a part of the base program.	method
Conditional replacement	After crosscutting, the body of the intercepted method is not always executed. The advice conditionally invokes the body of the method and potentially replaces its behavior with new behavior. Examples of this kind of advices are advices realizing transaction, access control, etc.	method
Multiple	After crosscutting, the body of the intercepted method is executed more than once. The advice invokes two or more time the body of the method it crosscuts generating potentially new behavior.	method
Crossing	After crosscutting, the advice invokes the body of a method (or several methods) that it does not intercepts. The advice have a dependency to the class owning the invoked method(s).	method
Write	After crosscutting, the advice writes an object field. This access breaks the protection declared for the field and can modify the behavior of the underlying computation.	method

Read	After crosscutting, the advice reads an object field. This access breaks the protection declared for the field and can potentially expose sensitive data.	method
Argument passing	After crosscutting, the advice modifies the argument values of the method it crosscuts and then invokes the body of the method. The body of the method always executes at least once.	method
Hierarchy	The aspect modifies the declared class hierarchy. For example, the aspect adds a new parent interface to an existing one.	module
Field addition	The aspect adds new fields to an existing class declaration. These fields depending on their protection can be acceded by referencing an object instance of the affected class.	module
Operation addition	The aspect adds new methods to an exiting class declaration. These methods depending on their protection can be acceded by referencing an object instance of the affected class.	module

Munoz et al. [2008] also developed a tool support for their framework. This tool analyzes aspects to infer which invasive pattern they realize. It can also statically check that the aspects conform to the specification of the base program. The violation of the specification is used to alert developers about the risk introduced by unexpected interactions. This assists developers reviewing the harmful code and to reason about its interaction with the base program. Since the Munoz's approach requires a huge amount of work for formalising all the specifications, it seems too complex to be practical and scalable to the real world.

3.3.5 AOP breaks the contract between a base module and its clients

In the presence of aspects, clients of a base module can no longer trust that the provided service meets its specification. Each service can be affected by an advice. Dantas & Walker introduce [Dantas & Walker, 2006] the notion of harmless advice, which is similar to the notation of spectators (see Section 3.3.4). Unlike an

ordinary advice, a harmless advice is not allowed to influence the the underlying computations. Therefore, programmers may ignore *harmless advices* when reasoning about the partial correctness properties of their programs. Although harmless advices are useful for many common crosscutting concerns including: logging, tracing, profiling, invariant checking and debugging, they limit the power of AOP by forbidding aspects to be invasive.

Lagaisse et al. [2004] propose aspect integration contracts (AICs) to specify the permitted interference between an aspect and a base module. AICs specify the permitted interference between an aspect and the base code. AiC are composed of the "aspect requirements" specifications (what aspects require from the base code), "aspect functionalities and effects" specifications (what aspect do with the base code), and the "permitted interference" specification (what aspect can do with the base code). Although AICs seem to be a reasonable limitation of the expressive power of AOP, they require too much of work from the programmers.

3.3.6 AOP escalates coupling

Similar to how OO languages rely on symbolic referencing (e.g. method calls by name), most AO languages in use today rely on referencing more complex structural properties of the program such as naming conventions and package structure [Wampler, 2007]. These structural properties are used by pointcuts to define intended conceptual properties about the base program [Kellens et al., 2006]. It means that pointcuts impose "design rules" that developers of the base program must adhere to in order to prevent unintended join point captures or accidental join point misses [Sullivan et al., 2005]. Design rules introduce tight coupling between the pointcut definition and the base program's structure. If a change occurs in any base module, all aspects need to be reviewed whether they are still working. This phenomenon is called the *fragile pointcut problem*. Furthermore, AO constructs introduce semantic coupling (see Section 5.2.3) that does not exist in OO systems.

Kellens et al. [2006] address the fragile pointcut problem by declaring pointcuts in terms of a conceptual model of the base program, rather than defining them directly in terms of how the base program is structured. As such, they transformed the fragile pointcut problem into the problem of keeping a conceptual model of the program synchronised with that program, when the program evolves.

3.3.7 Impact on maintainability and reusability

AOP has been proven to be effective in lexically separating different concerns of the system [Sant'Anna et al., 2003]. However, the influence of AOP on other quality attributes is still unclear. On the one hand, replacing code that is scattered across many modules by a single aspect can potentially reduce the number of changes during maintenance [Mortensen, 2009]. In addition, modules may be easier to reuse, since they implement single concerns and do not contain tangled code.

On the other hand, constructs such as pointcuts and advices can make the ripple effects in AO systems far more difficult to control than in OO systems. Current AO languages rely on referencing structural properties of the program such as naming conventions and package structure. These structural properties are used by pointcuts to define intended conceptual properties about the program. The obliviousness property of AspectJ implies that the underlying system does not have to prepare any hooks, or in any way depend on the intention to apply an aspect over it [Katz, 2004]. Thus, maintenance changes that conflict with the assumptions made by pointcuts introduce defects [Mortensen, 2009]. This phenomenon is called the pointcut fragility problem [Dijkstra, 1968]. It occurs when a pointcut unintentionally captures or misses a given join point as a consequence of seemingly safe modifications to the base code [Koppen et al., 2004; Mortensen, 2009]. Kästner et al. [2007] reported such silent changes during AO refactoring.

Obliviousness also leads to programs that are unnecessarily hard to understand [Griswold et al., 2006]. Since not all the dependencies between the modules in AO systems are explicit, an AO maintainer has to perform more effort to get a mental model of the source code [Storey et al., 1999]. Creating a good mental model is crucial to understand the structure of a system before attempting to modify it [Mancoridis et al., 1998]. Studies of software maintainers have shown that 30% to 50% of their time is spent in the process of understanding the code that they have to maintain [Fjeldstad & Hamlen, 1983; Standish, 1984; Glass, 2002].

Moreover, incremental modifications and code reuse are not directly supported for the new language features of AspectJ [Hanenberg & Unland, 2001]. In particular, concrete aspects cannot be extended, advice cannot be overridden, and concrete pointcuts cannot be overridden. Hanenberg & Unland proposed four rules of thumb [Hanenberg & Unland, 2001], which allow one to build reusable

and incrementally modifiable aspects. However, increased complexity is the price that has to be paid for it.

3.4 Composition Filters - an alternative approach

Composition Filters (CF's) was defined by Aksit & Tripathi [1988] and originally implemented in the Sina language. The Composition Filters model can be thought of as the conventional OO model in which an object can be surrounded by input and output filters. *Filters* extend the message passing mechanism by manipulating incoming and outgoing messages. Incoming messages have to pass through the input filters until they are dispatched and the outgoing through the output filters until they are sent outside the object [Czarnecki & Eisenecker, 2000; Bergmans & Aksit, 2001]. Dispatching here means either to start searching of a local method, or to delegate the message to another object. The filters together compose the enhanced behaviour of the object, possibly in terms of other objects. The resulting model and its elements are shown in Figure 3.2.

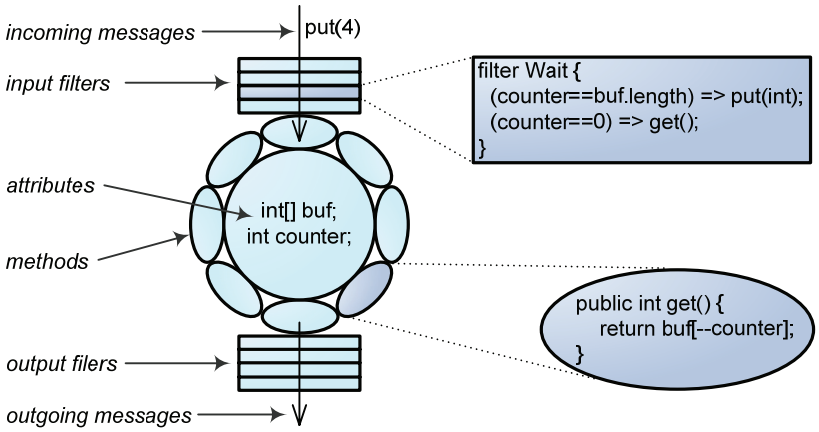


Figure 3.2 The Composition Filters model

A filter definition consists of a *filter type* and *filter guards*. It has the following form:

```
filter filterType {
  condition   =>   selector1,   selector2,   ...,   selectorN;
    //filter guard 1
    ...
    //filter guard 2
}
```

A selector is mainly used for matching messages. In addition it may modify certain parts of messages or indicate the targget object to which the message should be

redirected. When the selector on the left hand matches, no further selectors should be considered. A guard matches the message if (1) the condition evaluates to true, and (2) the message matches one of the selectors. As soon as the first guard is matched, the message is said to be accepted by the filter. A filter rejects a message if none of the filter guards matches the message. The filter type determines the semantics associated with acceptance and rejection of messages [Bergmans, 1994; Bergmans & Aksit, 2001]. In other words, it determines how to handle the messages after the matching process.

The running example of CF's is shown using scenerio 2 (Learning Management System) from Chapter 2.5.2. The presented source code is written in a simplified version of CF's (Listing 3.3).

```
public class LogStatementCF implements Statement {
  private Statement delegate;
  public LogStatementCF(Statement st) { delegate = st; }
  private boolean isDML(String sql) {
    String tmp = sql.toUpperCase();
    return tmp.indexOf("UPDATE")>=0 ||
     tmp.indexOf("INSERT")>=0 || tmp.indexOf("DELETE")>=0;
  }
  private void log(String sql) {
    String login = User.getCurrentUser().getLogin();
    String dbUser = "";
    try {
      dbUser=st.getConnection().getMetaData().getUserName();
    } catch (SQLException e) {};
    System.out.println(
     new Date() + "; " + login + "; " + dbUser + "; " + sql);
  }
  public boolean execute(String sql) throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql);
  }
  //other execute(..) methods
  public int executeUpdate(String sql) throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.executeUpdate(sql);
  }
  //other executeUpdate(..) methods
  filter Dispatch { true => this.*, delegate.* };
}
```

Listing 3.3 The LogStatementCF class using CF's

An arriving message is evaluated according to the Dispatch filter. When a Dispatch filter rejects a message, an exception is raised. In case of acceptance, the message is dispatched to the object that corresponds to the target of the matching selector [Bergmans, 1994]. In our example, an object starts searching the method that

matches the incoming message in its class. If the method is not found, then the search is continued in the Statement class.

The difference between the OO and CF's solution is that the latter does not need to define 29 methods for delegating messages to Statement object, because the delegation is achieved by the Dispatch filter. However, it should be noticed that the CF's implementation is not free from code tangling.

3.5 Summary

The essential problem with OOP is the lack of proper mechanisms to separate the implementation of crosscutting concerns from the implementation of core concerns. This limitation can be overcome by AOP and CF's. Each of these paradigms builds on all the advantages of the OO paradigm and overcomes some OO weaknesses. However, no programming paradigm is without its own set of problems and pitfalls. In Section 3.3 we have explained the current problems that present a major threat against a mainstream adoption of AOP. Aspects make the source code hard to understand, break encapsulation, and increase coupling. On the other hand, CF's extends the OO paradigm in a natural way, but is less powerful than AOP. Comparing to AOP, CF's improves delegation-based reuse and allows one to avoid composition anomalies. We took an overview of CF's, because it is an interesting alternative to AOP. However, since CF's is less powerful than AOP and is still a theoretical concept unsupported by mainstream programming languages, we do not investigate it further in this dissertation. The earlier version of Section 3.3 was originally published in [Przybyłek, 2010c], while other sections of this Chapter are partly based on work published in [Przybyłek, 2007] and [Przybyłek, 2009].

Chapter 4. AoUML: a proposal for aspect-oriented modelling

One picture is worth ten thousand words.

Barnard, 1927

The aim of this chapter is to define a notation which we will use to visualize the source code presented in the next Chapters.

4.1 Introduction

A software design coordinates well with a programming language when the abstraction mechanisms provided at both levels correspond to each other [Piveta & Zancanella, 2003]. Misalignment of design and code results in weak traceability and poor comprehensibility. The wide acceptance of AOP in academia has led to growing interest in aspect-oriented (AO) modelling languages. Since AOP is usually built on top of OOP, it seems natural to adapt UML to aspect-oriented modelling (AOM). Although UML was not designed to provide constructs to describe aspects, its flexible and extensible metamodel enables it to be adapted for domain-specific modelling [OMG, 2009a]. The progression of the AO paradigm, from implementation to design, is very similar to the evolution of the object-oriented and structured paradigms moving from the implementation level to the design level. The movement of the paradigm up the stages of the software lifecycle aid in reducing the semantic gap between each development phase [Gray, 2002].

UML has two ways of extending its language, one is by elaborating a Meta Object Facility (MOF) metamodel and another is by constructing a UML profile. A UML profile is a predefined set of stereotypes, tagged values, constraints, and graphical icons which enable a specific domain to be modelled. It was defined to provide a light-weight extension mechanism [OMG, 2009a]. The term "light-weight" means that the extension does not define new elements in the UML metamodel. The intention of profiles is to give a straightforward mechanism for adapting the UML metamodel with constructs that are specific to a particular domain [OMG, 2009a]. The advantages of choosing the light-weight extension

mechanism are that models can be defined by applying a well-known notation and that this method is generally supported by UML tools. On the other hand, the drawbacks are that, since stereotypes are extensions to the existing elements, certain principles of the original elements must be observed, and consequently expressiveness is limited. Elaborating an MOF metamodel is referred to as heavy-weight extension and is harder than constructing a profile. It also has far less tools support. However, the metamodel constructed can be as expressive as required. Another drawback of the heavy-weight mechanism is the introduction of interdependency between specific versions of UML and its extensions. If UML changes in any way, its extensions may also have to change.

In the last decade, numerous UML's extensions to support AOM have been presented (see [Schauerhuber et al., 2007]). However, none of them has become an acceptable standard. Researchers have usually concentrated on providing UML profiles, while less attention has been given to constructing heavy-weight extensions.

The remainder of this Chapter is organized as follows. In the next Section, the motivation for our proposal is explained. Section 4.3 describes the research methodology. In Section 4.4, a general overview of our extension to the UML metamodel is given. Then, in Section 4.5, the details of each meta-class are introduced. In Section 4.6, in turn, we give two examples of using our proposal. Finally, the last section summarizes our work.

4.2 Motivation for our proposal

The motivation behind our proposal is to integrate the best practices of the existing AO extensions – particularly the following: [Clarke & Banaissad, 2005; Evermann, 2007; Hachani, 2003b; Jacobson & Ng, 2005; Kande, 2003; Kande et al., 2002; Lions et al., 2002; Sapir et al., 2002; Stein et al., 2002a; Stein et al., 2002b] – to define an MOF metamodel that supplements UML with means to visualize AspectJ programs. The existing extensions are not satisfactory for different reasons. Evermann [2007], Fuentes & Sanchez [2007], Gao et al. [2004], Groher & Baumgarth [2004], Groher & Schulze [2003], Mosconi et al. [2008], Stein et al. [2002a], Stein et al. [2002b], Zakaria et al. [2002] stereotype the class element as <<aspect>> and the method element as <<advice>>, although an aspect

is not a class, nor is an advice a method. While such stereotyping was acceptable until UML 1.5, it can no longer be used; the 2.0 release requires semantic compatibility between a stereotyped element and the corresponding base element.

The most valuable heavy-weight extensions were elaborated by Hachani [2003b] and Yan et al. [2004]. The main drawback of both metamodels is the lack of graphical representation for new modelling elements. Moreover, both metamodels contain too much implementation detail and so seem to overwhelm the designer. The Hachani's proposal is specified more strictly and in a more formal fashion but needs updating, because it extends UML 1.4. The other drawback of his proposal is that it modifies the UML metamodel.

Efforts [Aldawud et al., 2003; France et al., 2003; Hachani, 2003a; Reina et al., 2004] to create a generic metamodel which could be fitted to every AO implementation have been unsuccessful, because a metamodel of this kind introduces an impedance mismatch between the design constructs and the language constructs. The conceptual differences between aspect implementations such as AspectJ, JAsCo, Spring, AspectWerkz are significant and cannot be captured effectively in a single metamodel. Moreover, generalization at the design level would be counter-productive at a time when AspectJ is squeezing out other technology at the implementation level.

4.3 Research methodology

In developing our notation, we follow the guidelines developed by Hevner et al. [2004]. Table 4.1 discusses the realization of these guidelines in our work. Not the all guidelines are completely fulfilled in this project. Indeed, Hevner et al. advise against mandatory use of them.

Table 4.1 The AoUML project

No	Guideline	Realization
1	Design as an Artifact	The result of this research is a modelling language named AoUML, which is a construct in Simon's terminology [Simon, 1996]. AoUML extends UML to support AOM. The capture of aspects in the design phase simplifies the AO software development. It helps to better understand and document the design.

2	Problem Relevance	The relevance of modeling techniques in software development is well demonstrated. While AOP has been and remains one of the most visible research streams in the software engineering field, there is still no clear standard for AOM. AoUML bridges the gap between design and implementation in AOSD.
3	Design Evaluation	The utility of the artifact is demonstrated on several examples.
4	Research Contributions	The contribution of this research is the artifact itself, that enriches UML with constructs for modeling aspects. AoUML provides traceability to aspect-oriented code and in consequence allows developers to keep the consistency among design and implementation.
5	Research Rigor	AoUML is defined as a MOF metamodel in a way consistent with UML. Its specification uses the class diagram, and natural language.
6	Design as a Search Process	AoUML is designed by integrating the best features of the existing notations.
7	Communication of Research	Section 4.5 motivates a technical audience, while Section 4.6 primarily focuses on a designer audience.

4.4 Our extension to the UML metamodel

The elaborated extension is described by using a similar style to that of the UML metamodel. As such, the specification uses a combination of notations:

- UML class diagram – to show what elements exist in the extension and how the elements are built up in terms of the standard UML constructs;
- natural language – to describe the semantic of the meta-classes introduced.

The proposed extension introduces a new package, named AoUML, which contains elements to represent the AO concepts, such as: aspect, pointcut, advice, introduction, parent declaration, soft, custom compilation message and crosscutting dependency (Figure 4.1). The proposal reuses elements from the UML 2.2 infrastructure and superstructure specifications by importing the Kernel package. Figure 4.2 shows the dependencies between the UML Infrastructure [OMG, 2009a), the UML Superstructure (OMG, 2009b) and the AoUML package.

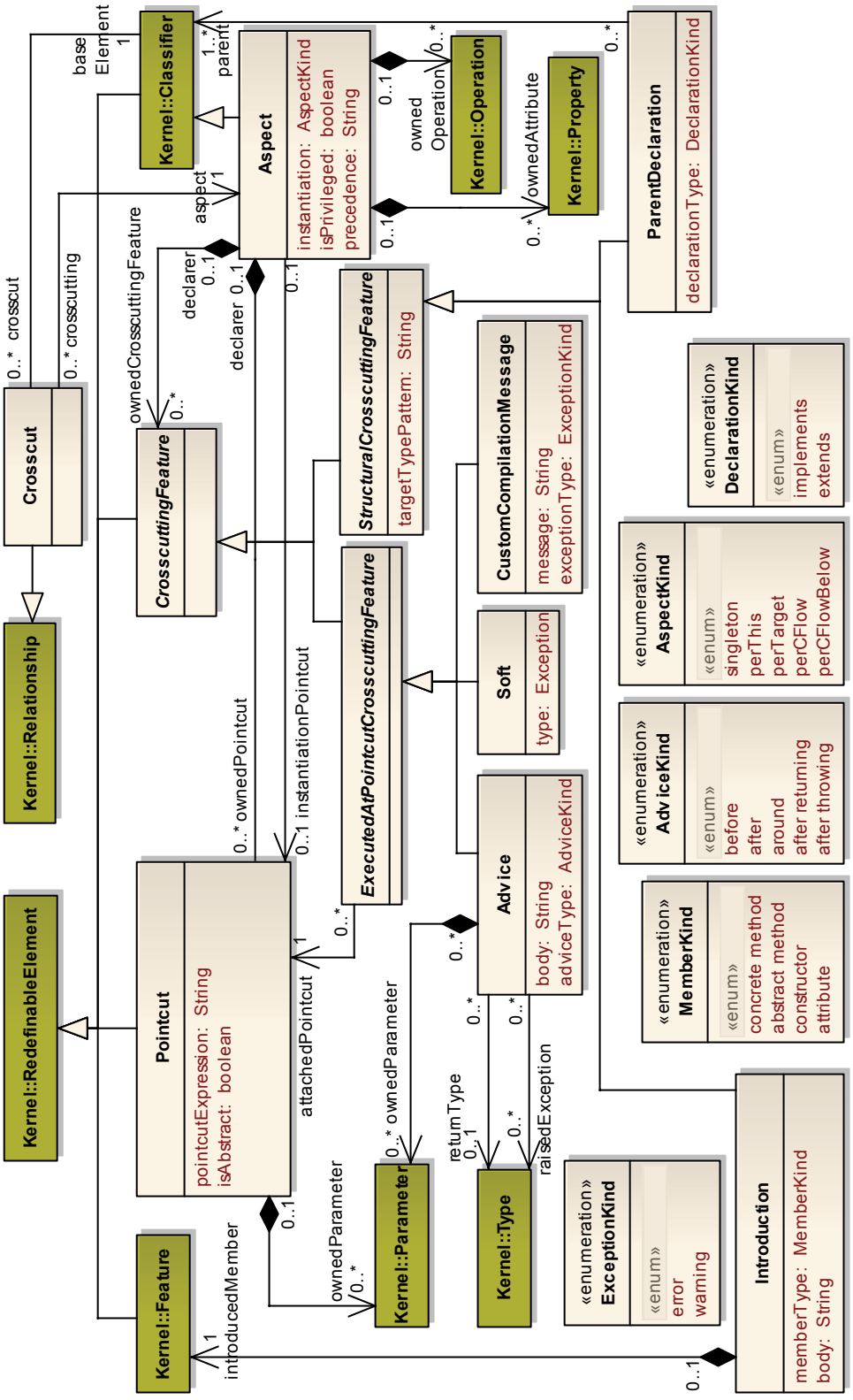


Figure 4.1 The AoUML package.

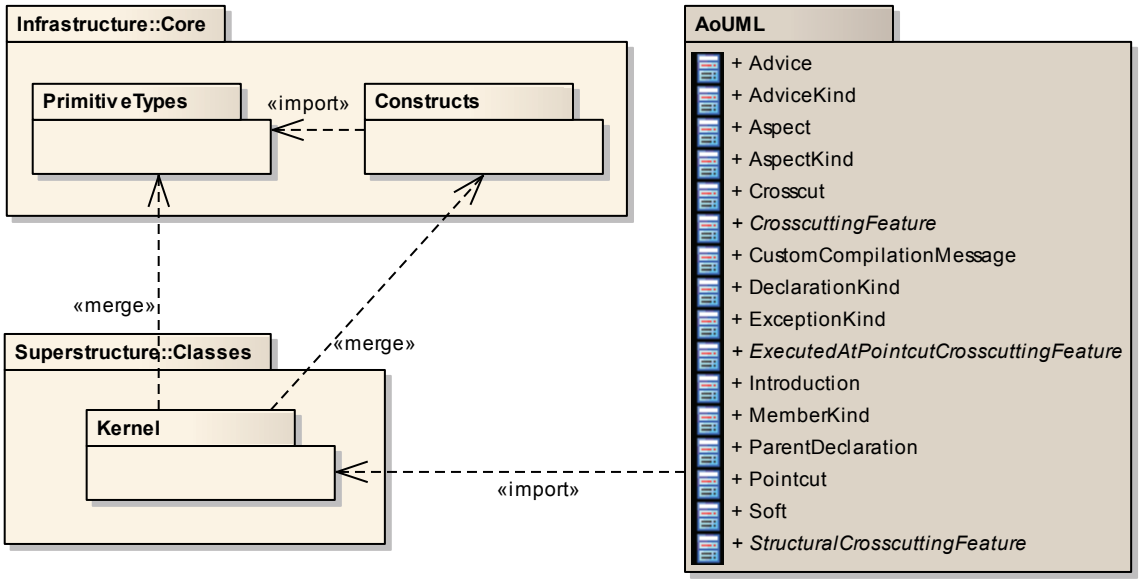


Figure 4.2 Dependencies between packages.

4.5 The AoUML package

4.5.1 Aspect

Semantics

An Aspect is a classifier that encapsulates the behaviour and structure of a crosscutting concern. It can, like a class, realize interfaces, extend classes and declare attributes and operations. In addition, it can extend other aspects and declare advices, introductions and parent declarations.

Attributes:

- isPrivileged – if true, the aspect code is allowed to access private members of the target classifier as a "friend"; the default is false.
- instantiation – specifies how the aspect is instantiated; the default is a singleton.
- precedence – declares a precedence relationship between concrete aspects.

Associations:

- ownedPointcut – a set of pointcuts declared within the aspect.
- instantiationPointcut – the pointcut which is associated with a per-clause instantiation model.

- ownedCrosscuttingFeature – a set of crosscutting features owned by the aspect.
- ownedAttribute – a set of attributes owned by the aspect.
- ownedOperation – a set of operations owned by the aspect.

Notation

The aspect element looks similar to the class element but has additional sections for pointcuts and crosscutting feature declarations. Figure 4.3 provides a graphical representation for aspects.

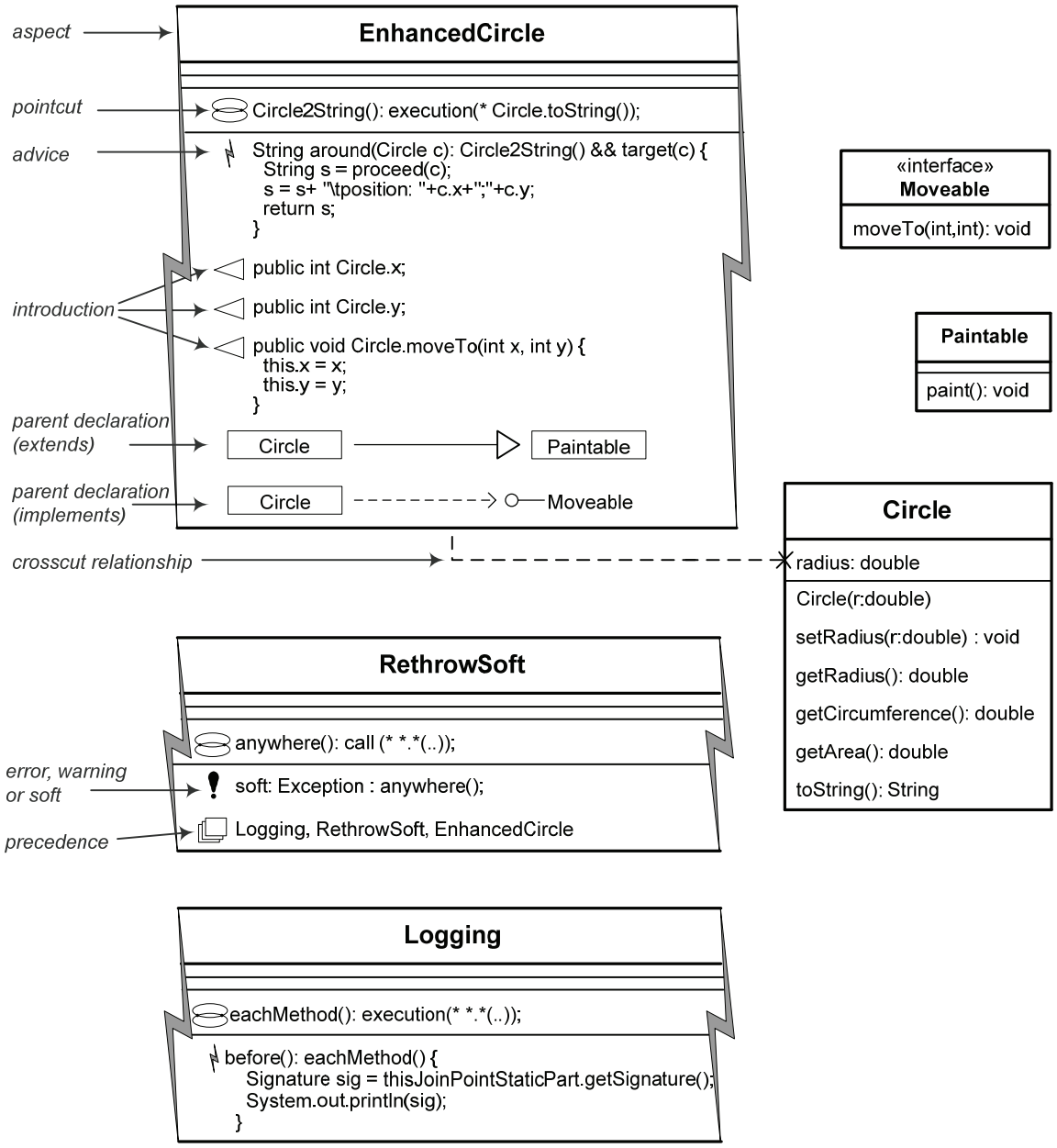


Figure 4.3 Aspect representation.

4.5.2 CrosscuttingFeature

Semantics

A CrosscuttingFeature is an abstract meta-class to generalize „pointcut-determinable" and structural features.

Associations:

- declarer – the aspect that owns this crosscutting feature.

4.5.3 StructuralCrosscuttingFeature

Semantics

A StructuralCrosscuttingFeature affects the structure of the classifier specified by the targetTypePattern expression.

Attributes:

- targetTypePattern – a pattern expression to match classes, interfaces or aspects which are affected by the crosscutting feature.

4.5.4 Introduction

Semantics

An Introduction allows designers to add new attributes or methods to classes, interfaces or aspects.

Attributes:

- memberType – specifies the kind of the inter-type member declaration.

Associations:

- introducedMember – the new member which has to be added to the target classifier.

4.5.5 ParentDeclaration

Semantics

A ParentDeclaration allows designers to add super-types to classes, interfaces or aspects.

Attributes:

- declarationType – specifies the kind of the declaration.

Associations:

- parent – the type implemented or extended by the target classifier.

4.5.6 ExecutedAtPointcutCrosscuttingFeature

Semantics

An ExecutedAtPointcutCrosscuttingFeature is woven with the base code at the places specified by the attached pointcut.

Associations:

- attachedPointcut – refers to the pointcut that defines a set of join-points at which the feature affects the base code.

4.5.7 Advice

Semantics

An advice affects the execution behavior of the base program by inserting its body at each join-point picked out by the attached pointcut. In addition, it has access to values in the execution context of the pointcut.

Attributes:

- adviceType – specifies when the advice's body is executed relative to the join-points picked out.
- body – the code of the advice.

Associations:

- ownedParameter – an ordered list of parameters to expose the execution context.
- raisedException – a set of checked exceptions that may be raised during execution of the advice.
- returnType – specifies the return result of the operation, if present (the "before" and "after" advice cannot return anything).

4.5.8 Pointcut

Semantics

A Pointcut is designed to specify a set of join-points and obtain the context surrounding the join-points as well. Join-points are well-defined places in the program flow where the associated advice must be executed. The purpose of

declaring a pointcut is to share the pointcut expression in many advices or other pointcuts.

Attributes:

- isAbstract - if true, the Pointcut does not provide a complete declaration; the default value is false.
- pointcutExpression – if a pointcut is not abstract, it specifies a set of join-points; it has the same form as in AspectJ.

Associations:

- ownedParameter – an ordered list of parameters specifying what data is passed from runtime context to the associated advice.
- advice – an advice that executes when the program reaches the join points.

Notation

The pointcut signature is as follows:

```
[visibility-modifier] pointcutName([parameters]):
  PointcutExpression
```

4.5.9 CustomCompilationMessage

Semantics

A CustomCompilationMessage specifies that particular join-points should never be reached. If the join-points picked out by the attached pointcut are reached, then either an error or warning will be signaled. It allows enforcing constraints such as coding standards and architectural rules.

Attributes

- message – the string the compiler will print if it encounters a match for the attached pointcut.
- exceptionType – error or warning; the only difference between error and warning is that errors will stop the compilation

4.5.10 Soft

Semantics

A Soft specifies that a particular kind of exception, if thrown at a join point, should bypass Java's usual static exception checking system and instead be thrown as an *org.aspectj.lang.SoftException*, which is subtype of *RuntimeException* and thus does not need to be declared.

Attributes:

- type – refers to the type of exception to soften.

4.5.11 Crosscut

Semantics

A Crosscut is a directed relationship from the aspect that specifies crosscutting concerns to one or more classifier, where the additional structure and/or behaviour will be combined.

Associations:

- baseElement – refers to the classifier that is crosscut.
- aspect – refers to the aspect that affects the classifier.

4.6 Illustrative examples

4.6.1 The Singleton pattern

The aim of Singleton is to ensure that only one instance of a class is created. All requests to create a new object are redirected to that one and only instance. The Singleton pattern ensures that only one instance of a class is created. All requests to create a new object are redirected to that one and only instance. The AO implementation of this pattern (Listing 4.1) was proposed by Hannemann & Kiczales [2002]. The coresponding AoUML diagram is shown at Figure 4.4. The *protectionExclusions* pointcut indicates what classes can access the *Singleton*'s constructor (if any). Its implementation (may be left empty) is provided by concrete sub-aspects. The *around* advice protects the *Singleton*'s constructor (lines 7–11). It creates the unique instance on demand and returns it instead of a new object. The concrete sub-aspect of *SingletonProtocol* defines what classes are *Singleton* (line 13).

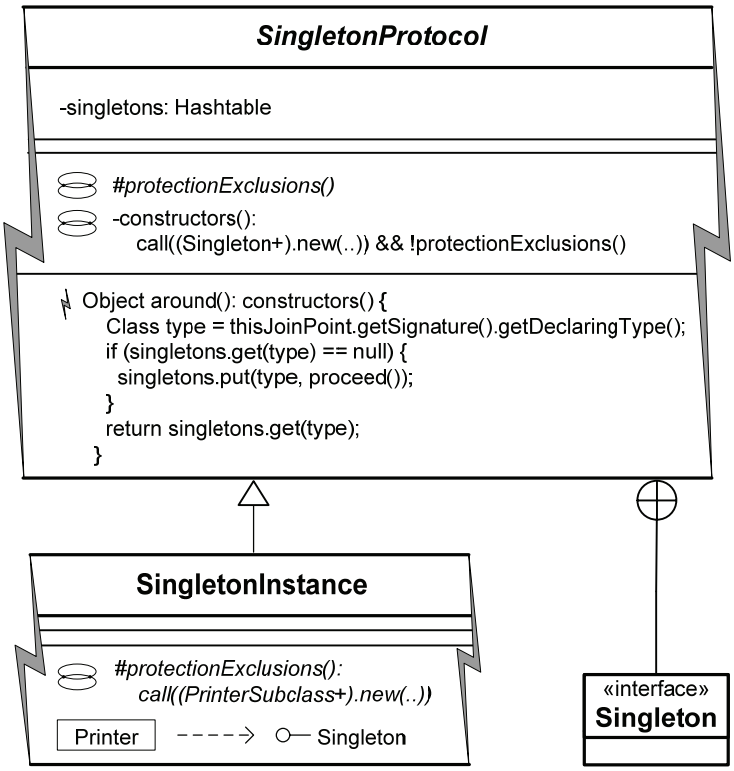


Figure 4.4 The Singlaton pattern.

```
public abstract aspect SingletonProtocol {
  private Hashtable singletons = new Hashtable();     //2
  public interface Singleton {}                       //3
  protected abstract pointcut protectionExclusions();//4
  private pointcut constructors():                    //5
   call((Singleton+).new(..)) && !protectionExclusions();//6
  Object around(): constructors() {                   //7
    Class type =
     thisJoinPoint.getSignature().getDeclaringType();//8
    if (singletons.get(type) == null) {               //9
      singletons.put(type, proceed());                //10
    }
    return singletons.get(type);                      //11
  }
}
public aspect SingletonInstance {
  declare parents: Printer implements Singleton;      //13
  protected pointcut protectionExclusions():          //14
   call((PrinterSubclass+).new(..));                  //15
}
```

Listing 4.1 The AO implementation of the Singleton pattern

4.6.2 The Visitor pattern

The intent of the Visitor pattern is to represent an operation to be performed on the elements of a tree structure. Visitor lets programmers define a new operation without modifying the classes of the elements on which it operates. Following the concept of SoC, the Visitor pattern allows to distinguish between the structure and its processing. Both concerns are implemented by two separate class hierarchies. Without the Visitor pattern, all the methods pertaining to the same kind of functional behavior would be spread over the structure hierarchy. With the Visitor pattern they are encapsulated into a single visitor class, which can be freely added or deleted from the system.

The first AO implementation of the Visitor pattern was presented by Hannemann & Kiczales [2002]. However, their implementation has a few imperfections. (1) The visit methods are distinguished via the name (i.e. visitLeaf, visitNode); a better practice is to use overloading and to distinguish the methods via the type of their parameter. (2) They use *instanceof* operator which is not a good programming practice. (3) They use confusing names of classes and interfaces. Our solution improves the above deficiencies.

H&K [2002] apply the Visitor pattern to operate on a binary tree. Their binary tree either: (a) is a leaf which consists a value, or (b) is a node which consists of a left binary tree, and a right binary tree. Figure 4.5 is the AoUML class diagram of the example.

The *Visitor* interface is implemented by all classes representing operations on the elements of the tree structure. It declares one operation for the type of each element in the structure. Every operation accepts as a parameter an object of the class it deals with. There are two concrete visitors: *SummationVisito*r (Listing 4.3) and *TraversalVisitor*. They provide the context for the algorithm and store the accumulated results as local state. The former collects the sum of all leaf values in the tree, whereas the later displays the tree. Adding a new behavior can be achieved by creating a new class that implements the *Visitor* interface.

Processing the tree starts when the visitor object is applied to the root node, using the *accept* method. This *accept* method invokes a *visit* method (Listing 4.2, Lines 10, 12) of the overgiven visitor using itself as the parameter. If the passed parameter is type of *BinaryTreeLeaf* it is proceeded directly (Listing 4.3, Line 7-9). If it is type of *BinaryTreeNode*, before or after processing it, the visitor object

applies itself to the left and right subtree by invoking their *accept* method and thus the whole tree structure is traversed recursively (Listing 4.3, Line 3-6).

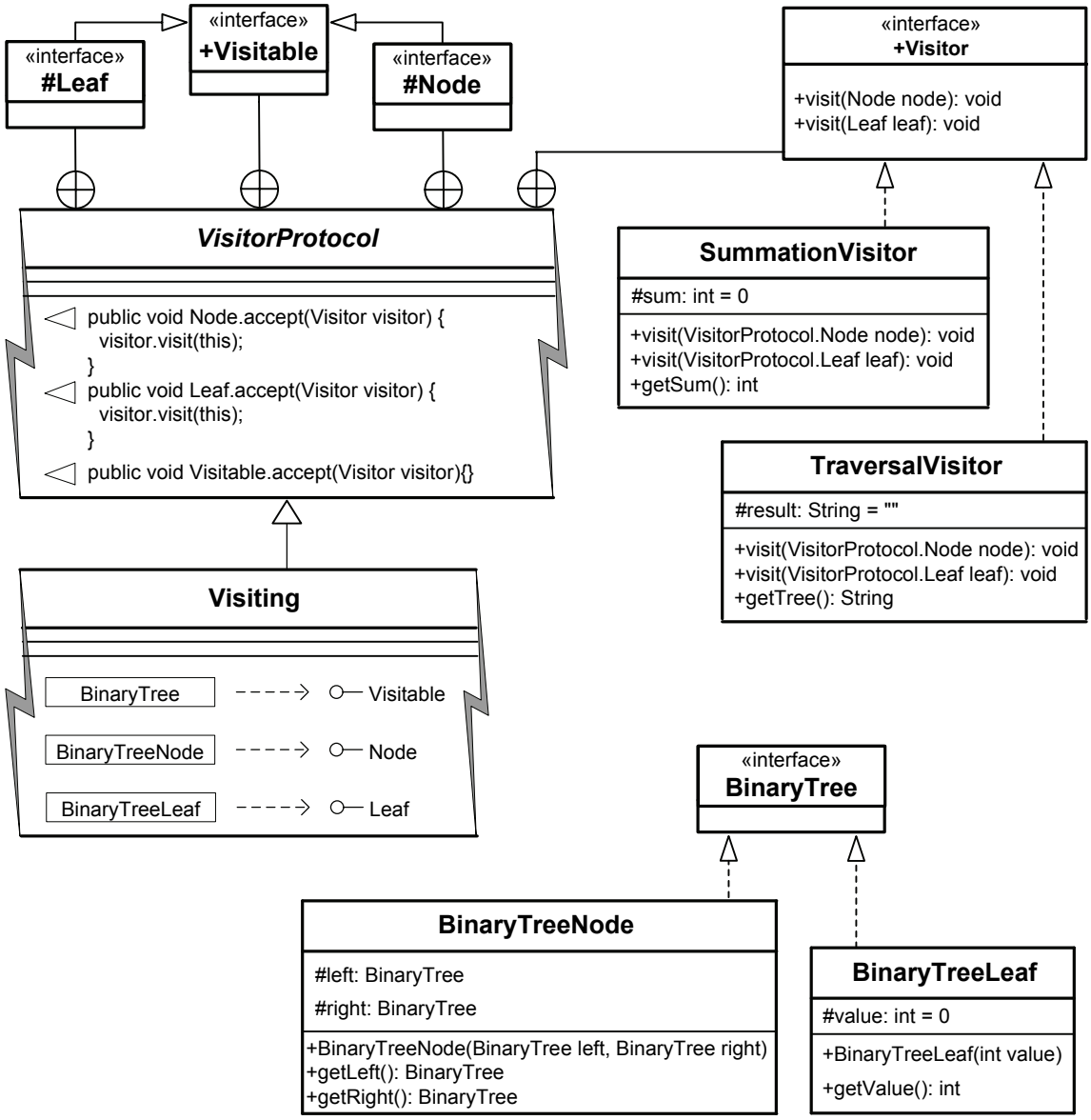


Figure 4.5 The Visitor pattern.

The *Visiting* aspect assigns the application classes, i.e. *BinaryTreeNode* and *BinaryTreeLeaf*, to implement the interfaces *Node* and *Leaf* respectively (Listing 4.2, Lines 15-16). These interfaces implement the *accept* methods via *VisitorProtocol* (Listing 4.2, Lines 9-12). Each *accept* method is used to pass the current tree to a visitor.

```
public abstract aspect VisitorProtocol {
  protected interface Visitable {}                          //2
  protected interface Node extends Visitable {}             //3
  protected interface Leaf extends Visitable {}             //4
  public interface Visitor {                                //5
    public void visit(Node node);                           //6
    public void visit(Leaf leaf);                           //7
  }
  public void Visitable.accept(Visitor visitor) {}          //8
  public void Node.accept(Visitor visitor) {                //9
    visitor.visit(this);                                    //10
  }
  public void Leaf.accept(Visitor visitor) {                //11
    visitor.visit(this);                                    //12
  }
}

public aspect Visiting extends VisitorProtocol {
  declare parents: BinaryTree implements Visitable;         //14
  declare parents: BinaryTreeNode implements Node;          //15
  declare parents: BinaryTreeLeaf implements Leaf;          //16
}
```

Listing 4.2 VisitorProtocol.aj and Visiting.aj

```
public class SummationVisitor implements
 VisitorProtocol.Visitor {                                  //1
  protected int sum = 0;                                    //2
  public void visit(VisitorProtocol.Node node) {            //3
    BinaryTreeNode btnode = (BinaryTreeNode) node;          //4
    btnode.getLeft().accept(this);                          //5
    btnode.getRight().accept(this);                         //6
  }
  public void visit(VisitorProtocol.Leaf leaf) {            //7
    BinaryTreeLeaf btleaf = (BinaryTreeLeaf) leaf;          //8
    sum += btleaf.getValue();                               //9
  }
  public int getSum() { return sum; }
}
```

Listing 4.3 SummationVisitor.java

4.6.3 Summary

The evolution of the AO paradigm is progressing from programming towards the design phase. AoUML enriches UML with constructs for visualizing AspectJ code. Although it takes inspiration from previous work [Evermann, 2007; Hachani, 2003b; Lions et al., 2002; Yan et al., 2004], it is one more step towards closing the gap between development phases. It makes the system model more consistent with the system implementation. In contrast to [Evermann, 2007; Hachani, 2003a;

Hachani, 2003b; Yan et al., 2004] AoUML provides dedicated icons for new elements. Graphical representation improves the understanding of models. Moreover, our proposal allows all aspect-related concepts to be specified in metamodel terms, so that no textual specification or notes are necessary. This means that automatic verification of the created models is simplified. Furthermore, AoUML does not modify the UML metamodel in any way. The earlier version of this Chapter was originally published in [Przybyłek, 2008a].

Chapter 5. Adaptation of object-oriented metrics

> *When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be.*
>
> Kelvin, 1883

The aim of this chapter is to introduce the AO metrics, which are the basis for the empirical studies conducted in the next Chapters.

5.1 Software measurement

To assess with some objectivity the quality of a design, we need to quantify design properties. Software engineers need quantitative assessment techniques to evaluate design alternatives. Measurement is fundamental to any engineering discipline and software engineering is no exception [Balasubramanian, 1996]. Measurement is "*the act or process of assigning a number or category to an entity to describe an attribute of that entity*" [IEEE, 1998] and is conducted by using metrics. IEEE Standard 1061 [1998] defines a software quality metric as "*a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.*" In this dissertation software metrics are used as an objective means to compare the quality of software systems developed using two different paradigms.

5.2 Modularity metrics

5.2.1 Existing OO metrics

Software engineering gurus consider modularity as a key principle when comparing design alternatives [Eick et al., 2001]. For years, they have proposed various programming techniques to improve software modularity. The dogma is that good modularization should exhibit high cohesion and low coupling [Anquetil & Laval, 2011]. This pair of attributes was firstly suggested to measure software modularity by Yourdon & Constantine [1979] as part of their structured design methodology and then it was adapted to the OO paradigm by Coad & Yourdon [1991], Booch [1994], and Meyer [1989]. It was also used by Tsang et al. [2000] to assess modularity in AO software. Furthermore, several empirical studies [Briand et al., 1999; Briand et al., 2001; Hitz & Montazeri, 1995; Ponnambalam, 1997] confirm that improvements in coupling and cohesion are linked to improved modularity.

Despite coupling and cohesion having been concepts in software design for almost 50 years, we still do not have widely-accepted metrics for them. However the most referenced and well-known are **CBO** (Coupling Between Object classes) and **LCOM** (Lack of Cohesion in Methods), defined by Chidamber & Kemerer (CK) in their metrics suite [Chidamber & Kemerer, 1994]. The CK suite is widely used for OO assessment; among other things, it was chosen by the Software Assurance Technology Center at NASA Goddard Space Flight Center. CBO is a count of the number of other modules to which a module is coupled. Two modules are coupled when methods declared in one module use methods or instance variables of the other module [Chidamber & Kemerer, 1994]. LCOM is the degree to which methods within a module are related to one another. It is measured as the number of pairs of methods working on different attributes minus pairs of methods working on at least one shared attribute (zero if negative).

CBO and LCOM complement each other, and because of their dual nature, they are useful only when analyzed together. Attempting to optimize a design with respect to CBO alone would trivially yield to a single giant module with no coupling. However, such an extreme solution can be avoided by considering also the antagonistic attribute LCOM (which would yield inadmissibly high values in the single-module case) [Hitz & Montazeri, 1995].

5.2.2 Existing AO metrics

Since AOP introduces several new kinds of interactions among modules, existing OO measures cannot be directly applied to AO software. The efforts to make the CK metrics suite applicable to AO software were originated by Sant'Anna et al. [2003] and continued by Zhao [2004], Ceccato & Tonella [2004], Shen & Zhao [2007], and Burrows et al. [2010a; 2010b]. The general suggestion is to treat advices as methods and to consider introductions as members of the aspect that defines them. Although this suggestion is enough to adapt LCOM, the adjustment of CBO requires further explanation. Ceccato & Tonella [2004] defined five metrics to measure different kinds of coupling:

- **CMC** (Coupling on Method Call) is a number of modules declaring methods that are possibly called by a given module;
- **CFA** (Coupling on Field Access) is a number of modules declaring fields that are accessed by a given module;
- **CAE** (Coupling on Advice Execution) is a number of aspects containing advices possibly triggered by the execution of operations in a given module;
- **CIM** (Coupling on Intercepted Modules) is a number of modules explicitly named in the pointcuts belonging to a given aspect;
- **CDA** (Crosscutting Degree of an Aspect) is a number of modules affected by the pointcuts and by the introductions in a given aspect.

Zhao [2004] complemented the Ceccato & Tonella's work by specifying the coupling dependencies in a formal way. Full definition of the Zhao's metrics can be found in the original study. One other AOP-specific coupling metric, named **BAC** (Base-Aspect Coupling), was defined by Burrows et al. [2010a]. BAC is a number of join points shadowed from an aspect via advice plus the number of module hierarchy changes from an aspect via intertype declarations, declare soft statements and declare parents statements. Shen & Zhao [2007] and Burrows et al. [2010b] created several fine-grained coupling metrics by splitting the Ceccato & Tonella's metrics into their component elements. These metrics quantify specific coupling properties of AOP.

Since the metrics proposed by Zhao [2004], Ceccato & Tonella [2004], Shen & Zhao [2007], and Burrows et al. [2010a; 2010b] measure only a specific kind of coupling, they may be used to assess the impact of individual AO mechanisms on high-level quality attributes. Nevertheless, they cannot by used to

compare the OO and AO implementations. An AO counterpart to be comparable with CBO must measure multiple kinds of coupling together. The metric that satisfies this requirement is **CBC** (Coupling between Components) defined by Sant'Anna et al. [2003]. It is broader than the original CBO in the sense that it additionally counts modules declared in formal parameters, return types, throws declarations and local variables. CBC considers most of the new kinds of coupling dependencies in AO software: accesses to aspect methods and attributes defined by introduction, and the relationships between aspects and classes or other aspects defined in the pointcuts. However, it is not complete, since it takes into account only syntactic dependencies. Syntactic dependency occurs when there is a direct reference between modules, e.g. aggregation.

5.2.3 Proposed coupling metric – CBO_{AO}

Coupling is a more complex attribute in AO systems, because new programming constructs introduce novel kinds of coupling dependencies. Since some of them occur without explicit references in the code, they are not so easy to realize. Ribeiro et al. [2007] call them *semantic dependencies*. We propose a metric, named CBO_{AO}, that takes into account this subtle coupling.

> CBO_{AO} considers a module X to be coupled to Y if (in parentheses, we provide abbreviations for the dependencies):
>
> - X accesses attributes of Y (A);
> - X calls methods of Y (M);
> - X potentially captures messages to Y (C);
> - Messages to X are potentially captured by Y (C_by);
> - X declares an inter-type declaration for Y (I);
> - X is affected by an inter-type declaration declared in Y (I_by);
> - X uses pointcuts of Y, excluding the case where Y is an ancestor of X (P).

The C_by and I_by dependencies are semantic and are consequences of C and I respectively. Figure 5.1 illustrates coupling dependencies in a simple program.

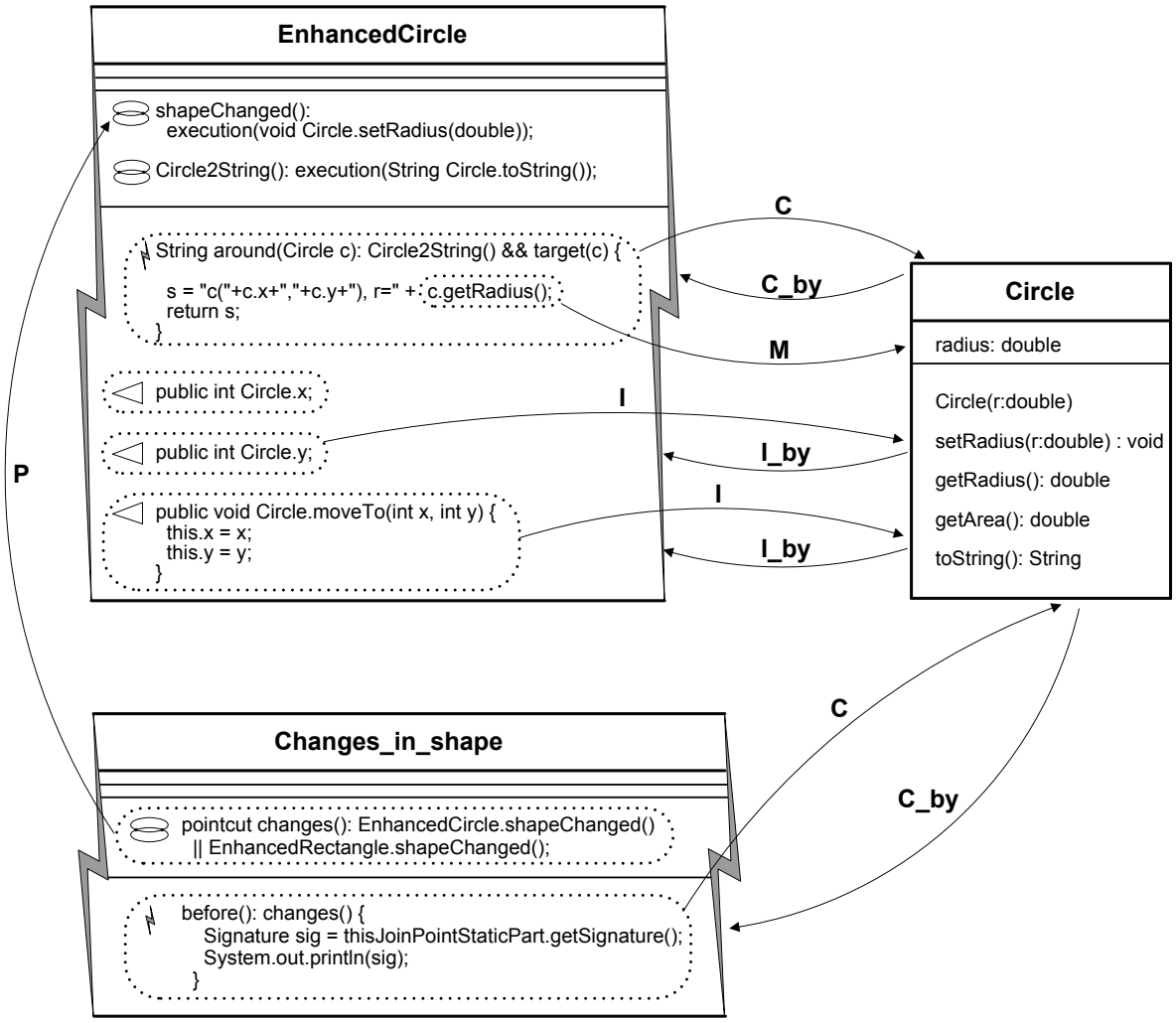


Figure 5.1 Examples of coupling dependencies.

A coupling dependency is represented by a labelled arrow from module X to module Y. The source of the arrow is the construt in X that generates coupling. The target is a module Y to which module X is coupled.

To construct our metric, we extrapolated the original CBO definition according to the question that underlies coupling: "*How much of one module must be known in order to understand another module?*" [Yourdon & Constantine, 1979]. The syntactic dependencies (i.e. A, M, C, I, P) occurring in our metric do not raise any doubts even among proponents of AOP [Sant'Anna et al., 2003; Garcia et al., 2005]. Thus, we only need to demonstrate that for understanding a given module X, we have to analyze Y if the C_by or I_by dependency exists between X and Y.

Let us consider two modules X and Y1 as shown in Listing 5.1. Assume that the inc(5) message has been sent to an instance of X. If we analyze X without considering the C_by dependency from X to Y1, we will deduce (following

program control flow) that the result is 6. However, the result is actually 11, and analyzing Y1 is necessary to compute it correctly.

```
public class X {
  public int inc(int x) {
    return ++x;
  }
}
public aspect Y1 {
  int around(int i): execution( int X.inc(int) ) && args(i) {
    return proceed(2*i);
  }
}
```

Listing 5.1 The C_by dependency

Now, suppose that two new modules Y2 and SubX were added as shown in Listing 5.2. Assume that the same message (inc(5)) has been sent to an instance of SubX. Once again, if we analyze X without considering the I_by dependency from SubX to Y2, we will deduce an incorrect result. The correct is 20.

```
public aspect Y2 {
  public int subX.inc(int x) {
    return x+10;
  }
}
public class SubX extends X {}
```

Listing 5.2 The I_by dependency

5.2.4 Coupling and cohesion at the system level

CBM_{AO} and LCOM are module level metrics. Nevertheless, we intend to compare systems. Thus, we need to lift the values of the module level to the system level. This lifting can be done by aggregating. There are a number of different aggregation functions such as average, sum, max and min. The sum is not useful because the size of an application would affect the measurement results, while modularity is orthogonal to size. The max and min function would make the result based on only one module [Rentrop, 2006]. In this research we use the average aggregation function to lift metric values to the system level.

We present the details of the computation on the Observer pattern [Hannemann & Kiczales, 2002]. Table 5.1 shows the values for both metrics for each module. The correspondence computations are as follows:

- for the OO implementation: $CBO_{AO} = (2+1+2+0+0)/5 = 1$;
  LCOM = (0+24+4+0+0)/5 = 5,6

- for the AO implementation: CBO_{AO} = (2+2+1+0+5+2+3+0+0)/9 = 1,7; LCOM = (0+0+0+10+0+9+0+0+0)/9 = 2,1

a) OO implementation

module name	module kind	CBO_{AO}	LCOM
Main	class	2	0
Point	class	1	24
Screen	class	2	4
ChangeObserver	interface	0	0
ChangeSubject	interface	0	0

b) AO implementation

module name	module kind	CBO_{AO}	LCOM
ColorObserver	aspect	2	0
CoordinateObserver	aspect	2	0
ScreenObserver	aspect	1	0
ObserverProtocol	aspect	0	10
Main	class	5	0
Point	class	2	9
Screen	class	3	0
Observer	interface	0	0
Subject	interface	0	0

Table 5.1 The CBO_{AO} and LCOM values for the Observer pattern

5.3 Evolvability and reusability metrics

Evolvability and reusability are quality characteristics that we cannot measure directly. The amount of reuse is usually measured by comparing the number of reused "items" with the total number of "items" [Frakes, 1993], where items depend on the granularity chosen, e.g. lines of code (LOC), function, or class. Since we are going to measure code reuse, we have chosen the granularity of LOC, yet we count only these reused lines that are part of the modules reused by applying the composition mechanisms of the underlying programming language. Thus, the proposed reuse level metric is defined as:

Reuse Level = LOC_of_reused_modules / total_LOC_in_system

The evolution metric we use is based on previous studies performed by Zhang et al. [2008] and Ryder & Tip [2001]. In their work, the difficulty of evolvability is defined in terms of atomic changes to the modules in a program. At

the core of this approach is the ability to transform source code edits into a list of atomic changes, which captures the semantic differences between two releases of a program. Zhang et al. [2008] presented a catalog of atomic changes for AspectJ programs. For the purpose of our study, we have slightly modified their catalog. Firstly, we consider deleting a non-empty element as an atomic change. Secondly, we use the term "module" as a generalization of class, interface, and aspect.

Our evolution metric breaks source code edits into a list of the following atomic changes:

- add an empty module,
- delete a module,
- add a field,
- delete a field,
- add an empty method,
- delete a method,
- change body of method,
- add an empty advice,
- delete an advice,
- change an advice body,
- add a new pointcut,
- change a pointcut body,
- delete a pointcut,
- introduce a new field,
- delete an introduced field,
- change an introduced field initializer,
- introduce a new method,
- delete an introduced method,
- change an introduced method body,
- add a hierarchy declaration,
- delete a hierarchy declaration,
- add an aspect precedence,
- delete an aspect precedence,
- add a soften exception declaration,
- delete a soften exception declaration.

5.4 Summary

The advent of a new paradigm requires software engineers to define new metrics to measure the quality of programs in this paradigm. In this chapter we reviewed the existing AO metrics. We found that the existing metrics are invalid for evaluating coupling in AO systems, since they do not take into account semantic dependencies between the system modules. Next, we presented all the ways by which modules

can be coupled to each other within AO systems. A new coupling metric (CBO_{AO}) was defined on the base of these coupling dependencies. CBO_{AO} was earlier presented and discussed in scientific forums at ENASE'10 [Przybyłek, 2010a] and ETAPS'11 [Przybyłek, 2011b]. We also proposed metrics for assessing software evolvability and reusability. These metrics were originally introduced in [Przybyłek, 2011c].

Chapter 6. Impact of aspect-oriented programming on software modularity

Measure all that is measurable and attempt to make measurable that which is not yet so.

Galileo

The aim of this chapter is to perform a metrics-based comparison among AO and OO software with respect to modularity.

6.1 Research methodology

The aim of this research is to compare AO and OO systems with respect to software modularity from the viewpoint of the developer. The research method employed is Multiple Embedded Case Study. The units of analysis are the 23 Gang-of-Four (GoF) design patterns [Gamma et al., 1995] and 11 real-world systems. Because every individual case involves the examination of two subunits of analysis (OO and AO implementation), our study is called embedded. The process of conducting our study is illustrated using Activity Diagram in Figure 6.1.

The assessment of both OO and AO implementations bases on the application of metrics that quantify two fundamental modularity attributes, namely coupling and cohesion. In addition, the analysis of real-world systems is supplemented by size metrics. Table 6.1 overviews the employed metrics and associates them with the attributes measured by each one of them. Detailed description of the coupling metric is provided in Chapter 5.2.2. Figure 6.2 illustrates our measurement system.

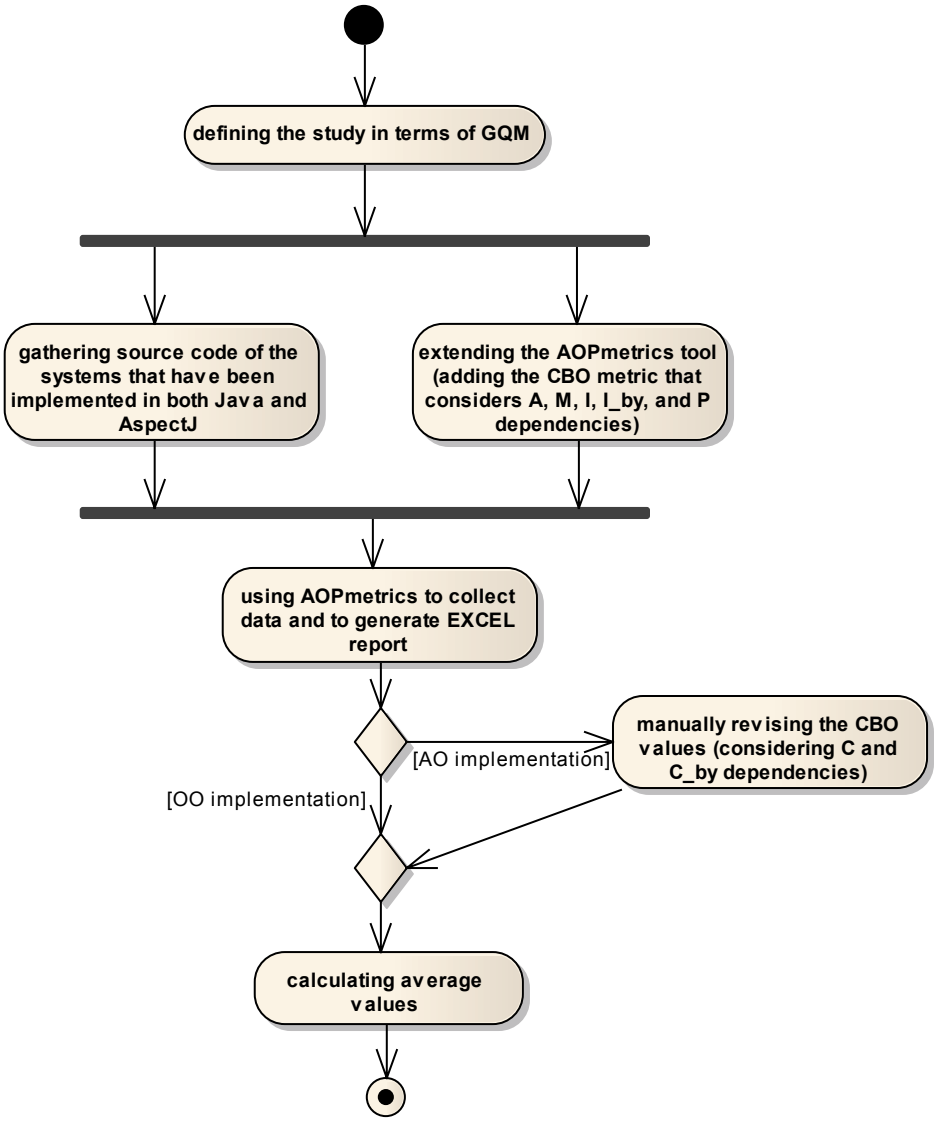


Figure 6.1 Activity Diagram for our study.

Table 6.1 Metric Definitions

Attributes	Metrics	Definitions
Size	Vocabulary Size	Number of modules (classes, interfaces, and aspects) of the system
	Lines of Code	Number of lines in the text of the system's source code
Modularity	Coupling Between Object classes	Number of other modules to which a module is coupled
	Lack of Cohesion in Methods	Number of pairs of methods/advices working on different attributes minus pairs of methods working on at least one shared attribute (zero if negative)

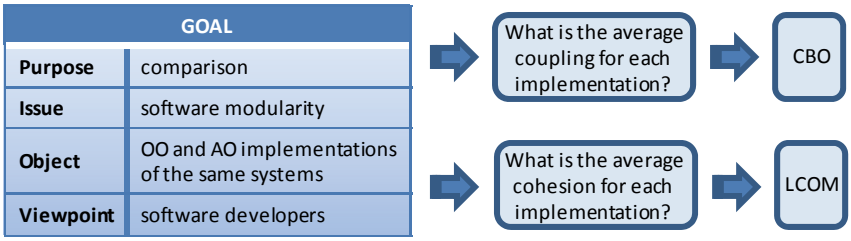


Figure 6.2 GQM diagram of the study.

The data were collected for each module (class, interface or aspect) of each system using the extended version of the AOPmetrics tool [Stochmiałek, 2006]. We extended AOPmetrics (available at: http://przybylek.wzr.pl/AOP/) to support the CBO_{AO} metric as defined in the previous Chapter, except for capturing C and C_by. This is due to some inherent bugs in AOPmetrics [Przybyłek, 2010]. Hence, the CBO_{AO} measures were recalculated manually using the Cross Reference View provided by the AJDT Eclipse plugin. The results are presented separately for design patterns (academic examples) and real-world systems.

6.2 Selected programs

Our study uses systems from different domains and of varying sizes (Table 6.2). All of the real-world systems were originally implemented in Java and, afterwards, were refactored using AspectJ, so that the code responsible for some crosscutting concerns was moved to aspects. In each case, code refactoring was done by proponents of AOP to present the benefits of AOP over OOP.

Table 6.2 Overview of the selected systems

Name	Description
Telestrada	A traveler information system being developed for a Brazilian national highway administrator. It allows its users to register and visualize information about Brazilian roads.
Pet Store	A demo for the J2EE platform that is representative of existing e-commerce applications.

CVS Core	An Eclipse Plugin that implements the basic functionalities of a CVS client, such as checkin and checkout of a system stored in a remote repository.
EImp	An Eclipse Plugin that supports collaborative software development for distributed teams.
Checkstyle	An Eclipse Plugin to help programmers write Java code that adheres to a coding standard. The plugin does this by inspecting the Java source code and pointing out items that deviate from a defined set of coding rules.
Health Watcher	A web-based information system that was developed by Soares et al. [2002] for the healthcare bureau of the city of Recife, Brazil. The system aims to improve the quality of services provided by the healthcare institution, allowing citizens to register complaints regarding health issues, and the healthcare institution to investigate and take the required actions. It involves a number of recurring concerns and technologies common in day-to-day software development, such as GUI, concurrency, RMI, Servlets and JDBC.
JHotDraw	A framework for technical and structured 2D graphics. Its design relies heavily on some well-known design patterns. JHotDraw's original authors are Gamma & Eggenschwiler.
HyperCast	Software for developing protocols and application programs for application-layer overlay networks. It supports a variety of overlay protocols, delivery semantics and security schemes, and has a monitor and control capability. It was developed at the University of Virginia in cooperation with the Microsoft Corporation.
Prevayler	An object persistence library for Java. It is an implementation of the Prevalent System design pattern, in which business objects are kept live in memory and transactions are journaled for system recovery. Business object must be serializable, i.e., implement the java.io.Serializable interface, and deterministic, i.e., given an input, the object's methods must always return the same output.

Berkeley DB Java Edition	A database system that can be embedded in other applications as a fast transactional storage engine. It stores arbitrary key/data pairs as byte arrays and supports multiple data items for a single key. Berkeley DB provides the underlying storage and retrieval system of several LDAP servers, database systems and many other applications.
HyperSQL Database	A relational database management system. It offers a small and fast database engine which supports both in-memory and disk-based tables. HSQLDB is currently being used as a database and persistence engine in many projects, such as Mathematica and OpenOffice.

In the first five systems (i.e. Telestrada, Pet Store, CVS Core, EImp, Checkstyle), aspects were used to implement exception handling [Filho et al., 2006; Castor et al., 2009; Taveira et al., 2009]. Exception-handling is known to be a global design issue that affects almost all system modules, mostly in an application-specific manner.

For the next system (Health Watcher) [Soares et al., 2002; Greenwood et al., 2007] refactoring went beyond exception handling, including in addition concerns such as data persistence, concurrency and distribution (basic remote access to system services using Java RMI). Both the OO and AO designs of the Health Watcher system were developed with modularity and changeability principles as main driving design criteria.

AJHotDraw (ajhotdraw.sourceforge.net) is an aspect-oriented refactoring of JHotDraw with regard to persistence, design policies contract enforcement and undo command. It was started to experiment with the feasibility of adopting aspect-oriented solutions in existing software and demonstrate the strategies proposed by research of the Software Evolution Research Lab of Delft University of Technology in the Netherlands. The aims, objectives and experience of the AJHotDraw project are summarized by Marin et al. [2007].

Sullivan et al. [2005] encountered two types of development problems when refactoring logging and event notification in HyperCast. First, the tight coupling between aspects and method names prevented the development of aspects in parallel with primary code refactoring, because the aspects could only be

developed after inspecting the core concerns. Second, they found cases where joinpoints were not accessible, because AspectJ supports specifying joinpoints at the method call level and data member level, but not at the if or switch statement level. Next, they re-implemented the base version using AspectJ and crosscutting interfaces (XPI). What distinguishes that particular release, is the lack of introductions used. In our experiment, we evaluate the improved version.

Prevayler was refactored using AspectJ and horizontal decomposition by Godil & Jacobsen [2005]. The horizontal decomposition principles were proposed by Zhang & Jacobsen [2004] to guide the AO refactoring and implementation of complex software systems. The refactored code includes persistence, transaction, query, and replication management [Katz, 2004].

By analyzing the domain, manual, configuration parameters, and source code, Kästner et al. [2007] identified many parts of Berkeley DB that represented increments in program functionality that were candidates to be refactored into features. These features are implicit in the original code. They vary from small caches to entire transaction or persistence subsystems. All identified features represent program functionality, as a user would select or deselect them when customizing a database system. From these features, they chose 38 and manually refactored one feature after another (wwwiti.cs.uni-magdeburg.de/iti_db/berkeley/). They used various OOP-to-AOP refactoring techniques, including Extract Introduction, Extract Beginning and Extract End, Extract Before/After Call, Extract Method, and Extract Pointcut [Kästner, 2007].

Störzer et al. [2006] refactored version 1.8.0 of HSQLDB (sourceforge.net/projects/ajhsqldb/). They started with an accepted catalog of well-known crosscutting concerns and then tried to find classes, methods or fields related to the respective concerns. They used manual semantics-guided code inspection supported by Feature Exploration and analysis tool to find a relevant crosscutting code. They discovered and refactored many standard crosscutting concerns, including Logging, Tracing, Exception Handling, Caching, Pooling and Authentication/Authorization. When becoming familiar with the source code, they also found some application specific aspects, for example trigger firing or checking constraints before certain operations are performed [Störzer, 2007].

We also investigate the 23 GoF patterns (Table 6.3) which intensively involve crosscutting concerns. Design paterns represent common software problems and the solutions to those problems. For each pattern Hannemann &

Kiczales [2002] developed an academic example that makes use of the pattern, and implemented the example in both Java and AspectJ. The AspectJ implementations are thought as illustrations of good AOP style and design [Monteiro & Fernandes, 2005]. The Java implementations correspond to the sample C++ implementations in the GoF book [Gamma et al., 1995].

Table 6.3 Overview of the 23 GoF design patterns [Gamma et al., 1995]

Name	**Intent**
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Chain of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.

Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Singleton	Ensure a class only has one instance, and provide a global point of access to it.
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Interpreter	Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

AbstractFactory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Table 6.4 indicates the websites of the programs that are publicly available. The source code of other programs was obtained from the authors.

Table 6.4 Websites of the analyzed programs

Name	**Source**
Telestrada	http://www.kevinjhoffman.com/icse2008/
Pet Store	http://www.kevinjhoffman.com/icse2008/
Checkstyle	http://eclipse-cs.sourceforge.net
Health Watcher	http://www.comp.lancs.ac.uk/~greenwop/ecoop07/
JHotDraw	http://www.jhotdraw.org, http://ajhotdraw.sourceforge.net
HyperCast	http://www.comm.utoronto.ca/hypercast/
Prevayler	http://www.prevayler.org
Berkeley DB Java Edition	http://oracle.com/technology/products/berkeley-db, http://wwwiti.cs.uni-magdeburg.de/iti_db/berkeley/
HyperSQL Database	http://hsqldb.org, http://sourceforge.net/projects/ajhsqldb/
23 GoF design paterns	http://www.cs.ubc.ca/labs/spl/projects/aodps.html

6.3 Experimental results: 11 real-world systems

Table 6.5 shows the obtained results for both size metrics (vocabulary size and LOC) and both modularity metrics (CBO_{AO} and LCOM). For all the employed metrics, a lower value implies a better result. The fifth and sixth column presents the mean values of the measures, over all modules per system. Rows labeled 'Δ' indicate the percentage difference between the OO and AO implementations relative to each metric. A positive value means that the original version performs better, whereas a negative value indicates that the refactored version exhibits better results.

In the case of both CVS and EIMP, their refactored code is not publicly available, so we based our analysis on the measurements carried out by Castor et al. [2009]. However, since they do not consider all kinds of coupling, we cannot present the exact CBO_{AO} values. We can only say that coupling is greater for the refactored systems.

Contradicting the general intuition that AOP makes programs smaller, the refactored versions are larger with regard to the LOC metric in the six cases. However, the increases are rather small and range between 1% and 4% (except for Checkstyle).

The average coupling between modules is significantly higher in most of the refactored versions. For the refactored versions of Prevayler and Health Watcher, it is more than 30% higher than for the corresponding OO releases. Only for HSQLDB, JHotDraw and HyperCast is the increase rather slight. Checkstyle is the only system whose AO version exhibited a better outcome for coupling, with a reduction of 3%. The higher coupling is the result of introducing new constructs intrinsic for AOP. In a typical scenario during AO refactoring, the coupling generated by explicit method call is replaced by the coupling generated by implicit advice triggering. Moreover, Filho et al. found [Filho et al., 2006] that new coupling was introduced when exception-handler aspects had to capture contextual information from classes.

Although the obtained results were as expected due to the above presented theoretical considerations, they contradict the outcomes achieved in several earlier studies. The advocates of AOP claim that the refactored versions of Telestrada [Filho et al., 2006; Castor et al., 2009], Pet Store [Filho et al., 2006; Castor et al., 2009], CVS Core Plugin [Filho et al., 2006; Castor et al., 2009], EImp Plugin

[Filho et al., 2006; Castor et al., 2009,] Health Watcher [Greenwood et al., 2007; Soares et al., 2002], and Prevayler [Godil & Jacobsen, 2005] exhibit lower coupling. However, they take into account only a subset of the dependencies that generate coupling in AO systems. Hence, the coupling measured with their metrics is underestimated.

Table 6.5 Results for Size, Coupling and Cohesion Metrics

I	II	III	IV	V	VI
		VS	**LOC**	**CBO_{AO}**	**LCOM**
Telestrada	**OO**	233	3424	0,81	1,86
	AO	242(18)	3350	0,95	2,17
	Δ	4%	-2%	18%	16%
PetStore	**OO**	345	17798	2,32	20,63
	AO	382(37)	17914	2,76	20,19
	Δ	11%	1%	19%	-2%
CVS	**OO**	257	18876	5,76	71,31
	AO	261(4)	19423	higher	73,90
	Δ	2%	3%	x	4%
Elmp	**OO**	123	8708	1,84	1,53
	AO	126(3)	9041	higher	1,68
	Δ	2%	4%	x	10%
Checkstyle	**OO**	283	18083	7,61	16,01
	AO	330(23)	20101	7,41	22,67
	Δ	17%	11%	-3%	42%
Health Watcher	**OO**	88	6096	3,19	9,24
	AO	103(12)	5768	4,20	7,63
	Δ	17%	-5%	32%	-17%
JHotDraw	**OO**	398	22724	3,57	75,04
	AO	438(31)	23167	3,66	65,70
	Δ	10%	2%	3%	-12%
Hypercast	**OO**	370	50492	3,31	67,24
	AO	391(7)	51207	3,42	67,00
	Δ	6%	1%	4%	-0,4%
Prevayler	**OO**	167	5043	1,87	9,31
	AO	168(55)	4179	2,56	7,01
	Δ	1%	-17%	37%	-25%
Berkeley DB	**OO**	340	41651	4,38	126,31
	AO	452(107)	38770	4,73	78,21
	Δ	33%	-6,9%	8%	-38%
HSQLDB	**OO**	402	80736	4,11	226,91
	AO	413(25)	76210	4,12	247,30
	Δ	3%	-6%	0,3%	9%

The Lack of Cohesion in Methods is the metric for which the impact of AOP has remained unclear. For the refactored versions of Berkeley DB, Prevayler, Health

Watcher and JHotDraw, the average LCOM is respectively 38%, 25%, 17%, and 12% lower than for the corresponding original versions. On the other hand, the average LCOM grew by 42% in the refactored version of Checkstyle, 16% in Telestrada, 10% in the EImp Plugin and 9% in HSQLDB. A partial explanation for this increase is the large number of methods that were created to expose join points (e.g. try-catch blocks in loops, etc.) that AspectJ can capture [Hoffman & Eugster, 2007]. As discussed in [Castor et al., 2009], these new methods are not part of the implementation of the exception-handling concern but a direct consequence of using aspects to implement this concern. The average LCOM varied (positively or negatively) by less than 4% in the refactored versions of the remaining systems.

It is worth mentioning that most researchers compare aggregate coupling and cohesion between an OO and AO version of the same system. Aggregate coupling (cohesion) for a system is calculated as the sum of coupling (cohesion) taken over all modules. Hence, it can be derived from Table 6.5 as multiplication of the average value by vocabulary size. It should be also noted that the original versions perform better with regard to the aggregate coupling and cohesion, since the measures of vocabulary size grew in all cases, due to the introduction of aspects. Nevertheless, aggregate coupling does not satisfy the second axiom of Fenton & Melton [1990] for coupling measures. That axiom states that system coupling should be independent from the number of modules in the system. If a module is added and shows the same level of pairwise coupling as the already existing modules, then the coupling of the system remains constant.

6.4 Experimental results: the 23 GoF design patterns

The CBO_{AO} and LCOM (LCO in AOPmetrics) values were collected for each of the 128 modules across the OO implementations and 179 modules across the AO implementations. Table 6.6 presents the mean values of the metrics, over all modules per pattern. The lower numbers are better. The sixth and seventh column indicates the superior implementation with regard to the CBO_{AO} and LCOM metric, respectively.

Table 6.6 Modularity metrics computed as arithmetic means.

I	II	III	IV	V	VI	VII
	OOP		AOP		winner	
	CBO_{AO}	LCOM	CBO_{AO}	LCOM	CBO_{AO}	LCOM
Builder	0,75	2	1,80	2,20	OO	OO
Command	0,71	0,14	1,58	2,67	OO	OO
Iterator	0,75	0,25	1,40	1,80	OO	OO
Mediator	0,86	0,14	1,13	0,50	OO	OO
Proxy	1,20	0	1,38	0,13	OO	OO
Chain	1,38	0,25	1,58	1,08	OO	OO
Memento	0,67	0	0,75	0,50	OO	OO
State	1,57	0,14	1,86	0,43	OO	OO
Flyweight	0,80	0	0,86	0,14	OO	OO
FactoryMethod	0,50	0	1,38	0	OO	-
Facade	0,80	0	1,83	0	OO	-
Strategy	0,80	0	1,67	0	OO	-
Bridge	0,71	0	1,38	0	OO	-
Composite	0,75	4	1,42	4	OO	-
TemplateMethod	0,75	0	1	0	OO	-
Decorator	1,17	0	1,25	0	OO	-
Prototype	0,67	0,67	2,33	0	OO	AO
Singleton	0,67	0,33	1,33	0	OO	AO
Observer	1	5,60	1,70	2,11	OO	AO
Interpreter	1,56	0,11	2,40	0	OO	AO
AbstractFactory	0,90	0,1	1,18	0,09	OO	AO
Visitor	1,71	0,71	1,92	0,17	OO	AO
Adapter	1	0	1	0	-	-

There is no pattern whose AO implementation exhibits lower coupling. For 22 patterns, the OO implementations present lower coupling, and in one pattern the values obtained for OOP and AOP are equal. With regard to cohesion, the OO implementations are superior in 9 cases, while the AO ones in 6 cases. 8 patterns exhibit the same cohesion in both implementations.

For a further analysis of the effects of AOP, we break the results for this paradigm in two parts: (I) core concerns, and (II) crosscutting concerns (Table 6.7). Metrics in each part are calculated as arithmetic means taken over: (I) all modules that implement the core concerns for a given pattern (it means all interfaces and classes except the Main class); (II) all aspects that comprise the pattern. Metrics in the first part reflect the modularization of core concerns, while metrics in the second part reflect the modularization of crosscutting concerns. The contribution of each part in the overall coupling and cohesion is shown as a percentage. In order to make a fair comparison between the two paradigms, Main classes were also excluded from the OO implementations.

The details of the computation are presented for the Observer pattern (see Figure 5.2). The AO implementation of Observer encompass 4 modules that implements core concerns (i.e. Point, Screen, Observer, Subject), 4 aspects (i.e. ColorObserver, CoordinateObserver, ScreenObserver, ObserverProtocol), and a Main class. The correspondence computations for the AO implementations are as follows:

- for the core concerns (colums: IV, V, VI):
  - core % = $$\frac{\text{the number of modules that implements core concerns}}{\text{the total number of modules minus one}} = \frac{4}{8} = 0{,}5$$
  - CBO_{AO} = (2+3+0+0)/4 = 1,25;
  - LCOM = (9+0+0+0)/4 = 2,25;
- for the crosscutting concerns (colums: VII, VIII, IX):
  - crosscutting % = 1 – core % = 0,5
  - CBO_{AO} = (2+2+1+0)/4 = 1,25;
  - LCOM = (0+0+0+10)/4 = 2,5;
- overall results (colums: X, XI):
  - CBO_{AO} = 0,5·1,25 + 0,5·1,25 = 1,25;
  - LCOM = 0,5·2,25 + 0,5·2,5 = 2,375;

It is worth noting that in the AO versions most of the "badness" is generally accumulated within aspects. When comparing the CBO_{AO} values for classes and interfaces only, the AO implementations are better in 4 cases and worse in 10 out of 23.

Table 6.7 Modularity metrics – a detailed view.

I	II	III	IV	V	VI	VII	VIII	IX	X	XI
	OOP		AOP							
	CBO_{AO}	LCOM	core			crosscutting			overall	
			%	CBO_{AO}	LCOM	%	CBO_{AO}	LCOM	CBO_{AO}	LCOM
Builder	0	2,67	75%	1,00	3,67	25%	3,00	0	1,50	2,75
Command	0,33	0,17	82%	0,89	0	18%	3,00	16,00	1,27	2,88
Iterator	0,67	0,33	75%	0,67	0,33	25%	3,00	8,00	1,25	2,25
Mediator	0,67	0,17	71%	0,60	0	29%	1,50	0,67	0,86	0,19
Proxy	0,50	0	43%	1,00	0	57%	1,75	0,25	1,43	0,14
Chain	1,14	0,29	82%	0,89	0	18%	3,5	6,5	1,36	1,17
Memento	0,50	0	71%	0,20	0	29%	1,50	2,00	0,58	0,58
Flyweight	0,50	0	67%	0,50	0	33%	1,00	0,50	0,67	0,17
FactoryMethod	0	0	71%	0,80	0	29%	2,00	0	1,15	0
Facade	0,75	0	80%	1,50	0	20%	3,00	0	1,80	0
Strategy	0,25	0	60%	1,00	0	40%	1,50	0	1,20	0
Bridge	0,17	0	86%	0,50	0	14%	4,00	0	0,99	0
Adapter	0,33	0	67%	0,5	0	33%	1,00	0	0,67	0
State	1,67	0,17	83%	1,40	0,6	17%	5,00	0	2,01	0,50
TemplateMethod	0	0	75%	0,33	0	25%	1,00	0	0,50	0
Decorator	0,60	0	33%	2,00	0	67%	1,00	0	1,33	0
Prototype	0	1,00	60%	1,67	0	40%	3,00	0	2,20	0
Singleton	0	0,50	60%	0,67	0	40%	2,00	0	1,20	0
Observer	0,75	7,00	50%	1,25	2,25	50%	1,25	2,50	1,25	2,38
Interpreter	0,88	0,13	89%	1,38	0	11%	6,00	0	1,89	0
AbstractFactory	0,67	0,11	90%	0,89	0,11	10%	2,00	0	1,00	0,10
Visitor	1,33	0,83	82%	1,33	0,22	18%	3,50	0	1,72	0,18
Composite	0	5,33	82%	0,78	0	18%	3,50	14,00	1,27	2,52

The problem with the arithmetic mean is that each of the modules contributes equally to the final result. Intuitively, larger modules are more complex, so they should contribute more. In addition, LCOM is not normalized, which means that the cohesion measures of different modules (as they all have different numbers of methods and attributes) should not be compared. Thus, weighted arithmetic means were also calculated. The individual CBO and LCOM values are weighted by the number of methods defined in the module, plus one. Table 6.8 presents the averages calculated in this way. As it turns out, no pattern changes its group.

Table 6.8 Modularity metrics computed as weighted arithmetic means.

	OOP		AOP		winner	
	CBO	LCOM	CBO	LCOM	CBO	LCOM
Builder	0,53	2,35	1,48	3,67	OO	OO
Command	0,89	0,28	2,06	9,70	OO	OO
Iterator	0,74	0,26	1,48	2,44	OO	OO
Mediator	1,00	0,26	1,15	1,00	OO	OO
Proxy	0,90	0	1,52	0,14	OO	OO
Chain	1,33	0,33	2,14	2,46	OO	OO
Memento	0,64	0	1	0,91	OO	OO
State	1,83	0,17	1,897	0,41	OO	OO
Flyweight	0,67	0	0,94	0,18	OO	OO
Composite	0,46	4,92	2,29	6,15	OO	OO
FactoryMethod	0,33	0	1,43	0	OO	-
Facade	1	0	1,80	0	OO	-
Strategy	0,69	0,00	1,75	0	OO	-
Bridge	0,50	0	1,22	0	OO	-
TemplateMethod	0,40	0	0,75	0	OO	-
Decorator	1,13	0	1,25	0	OO	-
Prototype	0,33	0,83	2,53	0	OO	AO
Singleton	0,91	0,36	1,50	0	OO	AO
Observer	1,12	11,23	2,11	4,89	OO	AO
Interpreter	1,42	0,13	2,40	0	OO	AO
AbstractFactory	1,22	0,17	1,51	0,16	OO	AO
Visitor	1,64	0,92	2,24	0,24	OO	AO
Adapter	1	0	1	0	-	-

6.5 Deeper insight into modularity

An established technique for analysing the dependencies among the modules of a system is Dependency Structure Matrix (DSM). A DSM is a square matrix in which the columns and rows are labelled with modules and a non-empty cell models that the module on the row depends on the module on the column. The type of dependency is represented by the value of the cell (the shortcuts are introduced in Section 5.2.2). The CBO_{AO} metric for a module can be calculated from a DSM by counting non-empty cells in the row. To provide complex insight into modularity, LCOM for each module is also presented. The differences in modularity between OO and AO implementations is shown on the Observer pattern [Hannemann & Kiczales, 2002].

The participants in the Observer pattern are subjects and observers. The subject is an object which changes its state, and the observer is an object whose

own invariants depend on the state of the subject. For example, let's consider that there is a subject – Point and one observer – Screen (Figure 6.3). Whenever the Point object changes its position, the Screen object has to be updated. The intention of the Observer pattern is to define a one-to-many dependency between a subject and multiple observers, so that when the subject changes state, all its observers are notified and updated automatically [Piveta & Zancanella, 2003]. Particular classes can play one or both of the Subject and Observer roles. In the presented example, Screen acts as Subject and Observer at the same time. The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject [Gamma et al., 1995]. When a subject wants to report a state change to its observers, it calls its own notifyObservers method, which in turn calls an update method on all observers in the list [Hannemann & Kiczales, 2002]. Since the notification of observers by the subject spreads across the domain classes, it is a crosscutting concern. The main problem with the OO implementation of this pattern is that it is hard to apply the pattern to an existing design.

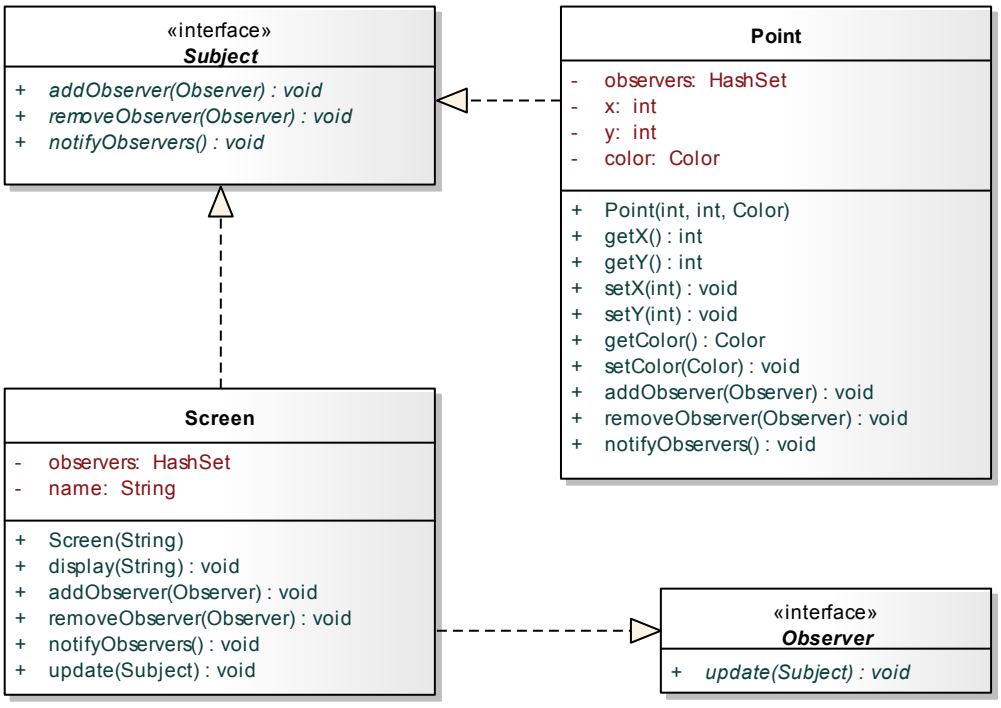


Figure 6.3 The structure of an instance of the Observer pattern in Java.

Hannemann & Kiczales [2002] developed an AO solution as shown at Figure 6.4. The *ObserverProtocol* aspect provides the logic for notifying the observers, when a subject changes its state. The empty interfaces Subject and Observer are marker

interfaces that are used by inheriting subaspects to map the application classes to their roles. E.g. *ColorObserver* assigns the *Observer* interface to the *Screen* class and the *Subject* interface to the *Point* class. The observers for each subject are stored in a global *WeakHashMap* that maps a subject to a list of observers. An *Observer* object becomes registered to receive notifications from a *Subject* object when it is passed to the *addObserver(Subject, Observer)* method. Passing it to the *removeObserver(Subject, Observer)* method ends the *Observer* object's registration to receive notifications.

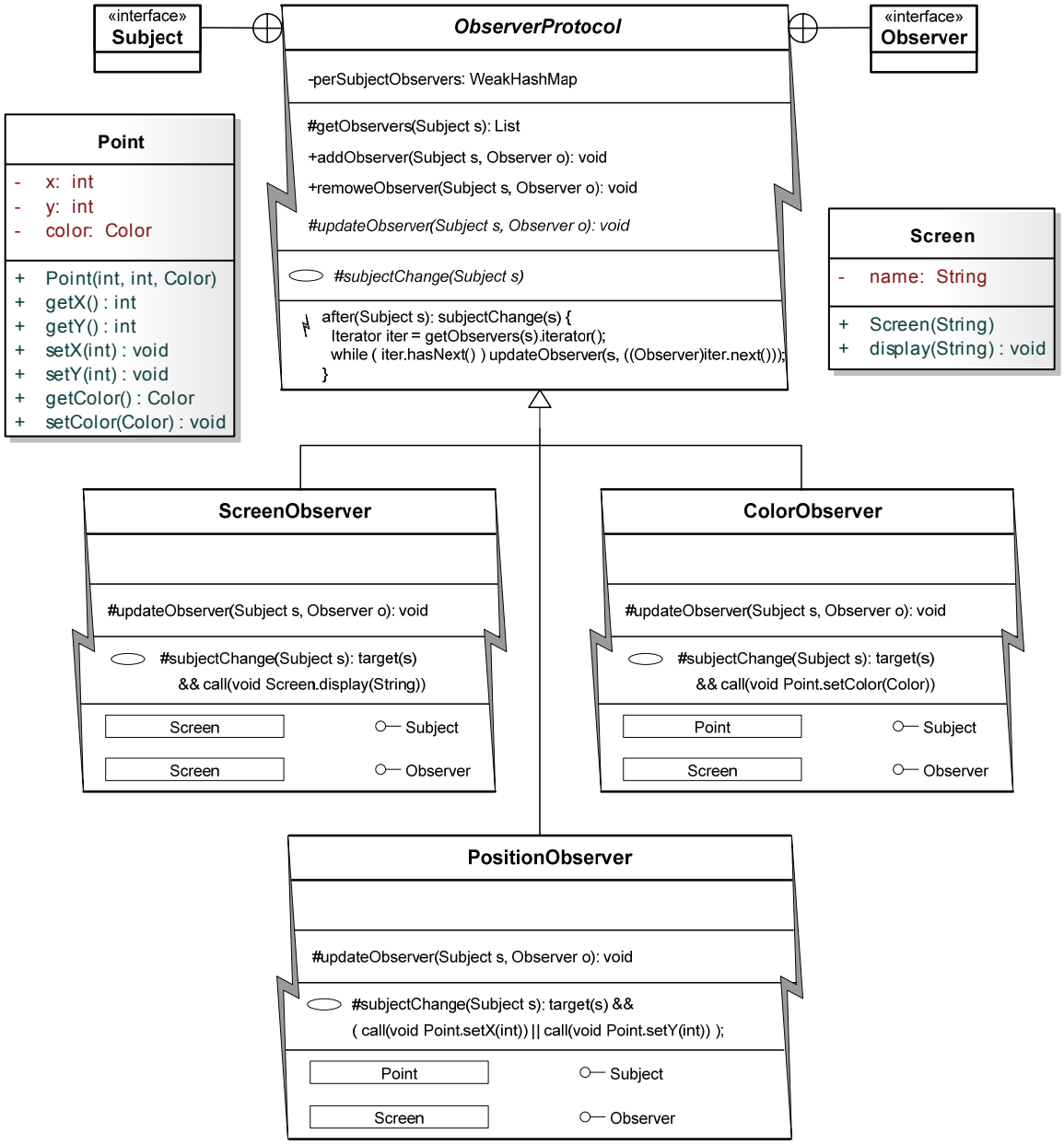


Figure 6.4 The structure of an instance of the Observer pattern in AspectJ.

The *after(Subject)* advice is fired just after reaching the *subjectChange(Subject)* pointcut. This pointcut describes the events that make the *Subject*'s state change and has to be implemented in the concrete subaspect. The after advice gets the *Observers* for the *Subject*, whose state was changed, and then calls the abstract method *updateObserver(Subject, Observer)* on each *Observer*. This method has to be implemented in the concrete subaspect. The *Subject* argument allows the method to know what object originated the notification.

Figure 6.5 shows the dependency matrixes for this pattern. In the OO implementation, the business logic and the pattern context are tangled within the participant classes. As a result, Point and Screen have a poor cohesion. Moreover, code for implementing the pattern is spread across all participants. In the AO implementation, all code pertaining to the relationship between observers and subjects is moved into aspects. Hence, the participant classes are entirely free of the pattern context, and as a consequence they are much more cohesive. In the OO version, a point directly informs its observers by sending a message to them. In the AO version, even though Point does not have any reference to its observers, the coupling has not disappeared. The coupling has changed its form from explicit method call to implicit join-points matching. Whenever a point changes its state, the relevant advice is triggered and the observers are notified. Since not all the dependencies between the modules are explicit, an AO programmer has to perform more efforts to get a mental model of the source code.

a) OO implementation

		1	2	3	4	5	CBO	LCOM
Main	1		M	M			2	0
Point	2				M		1	24
Screen	3				M	M	2	4
ChangeObserver	4						0	0
ChangeSubject	5						0	0

legend
class
interface
aspect

b) AO implementation

		1	2	3	4	5	6	7	8	9	CBO	LCOM
ColorObserver	1						C, I	I, M			2	0
CoordinateObserver	2						C, I	I, M			2	0
ScreenObserver	3							C, I, M			1	0
ObserverProtocol	4										0	10
Main	5	M	M	M			M	M			5	0
Point	6	C_by, I_by	C_by, I_by								2	9
Screen	7	I_by	I_by	C_by, I_by							3	0
Observer	8										0	0
Subject	9										0	0

Figure 6.5 DSMs for the Observer pattern.

6.6 Threats to validity

6.6.1 Construct validity[1]

Construct validity focuses on whether the measures used represent the intent of the study. We identify several limitations within this category. Firstly, we narrow software modularity to cohesion and coupling, despite of many other factors assigned to it. Nevertheless, cohesion and coupling are the concepts that lie at the heart of software modularity and are considered as main factors related to the goodness of modularization [Meyer, 1989; Booch, 1994; Hitz & Montazeri, 1995; Ponnambalam, 1997; Briand et al., 1999b; Briand et al., 2001].

Secondly, we could be criticised for applying metrics that are theoretically flawed. Briand et al. [1998] demonstrate that LCOM is neither normalized nor monotonic. The normalization condition requires that there is the upper limit of the values that the measures can take. Monotonicity states that adding a method which shares an attribute with any other method of the same module, must not increase LCOM. If we drop the very rare case where the methods of a module do not reference any of the attributes, the monotonicity anomaly disappears. The other problem with LCOM is that it does not differentiate modules well [Basili, 1996]. This is partly due to the fact that LCOM is set to zero whenever there are more pairs of methods which use an attribute in common than pairs of methods which do not [Briand et al., 1998]. In addition, the presence of access methods artificially decreases this metric. Access methods typically reference only one attribute, namely the one they provide access to, therefore they increase the number of pairs of methods in the class that do not use attributes in common [Briand et al., 1998]. The CBO metric also indicates inherent weakness. Briand et al. [1999a] illustrate that merging two unconnected modules may affect the overall coupling. Nevertheless, CBO as well as LCOM are widely applied and have been validated in many empirical studies [Basili, 1996; Briand et al., 1999a; Briand et al., 1999b].

Thirdly, the applied metrics address only one possible dimension of cohesion and coupling. Moreover, CBO implicitly assumes that all basic couples are of equal strength [Hitz & Montazeri, 1995]. In addition, it takes a binary approach to coupling between modules: two modules are either coupled or not. Multiple connections to the same module are counted as one [Briand et al., 1999a].

[1] Although all references in this Section to CBO and LCOM apply to the original versions, the considerations concluded are also valid for the extended versions.

In our defence we would point out that the OO community has yet to arrive at a consensus about the appropriate measurement of coupling and cohesion. The interested reader is referred to [Hitz & Montazeri, 1995; Briand et al., 1998; Briand et al., 1999a] where extensive surveys have been presented.

6.6.2 Internal validity

In our case internal validity concerns the question whether any observed effect was caused only by the programming paradigms involved. As in every study of this type, the experience, knowledge, skills, and insights of developers had an influence on the code they produced. Since the programmers of the AO versions contributed to the development of AOP, they might have done their very best to show that the new paradigm is superior. However, since we show that AOP harms software modularity, this weakness supports our conclusions. The causal effect of AOP on software modularity was explained in Section 3.3.

6.6.3 External validity

The investigated cases can be thought of as a population of systems whose implementations are publicly available in both Java and AspectJ. These systems were developed by experienced practitioners from several countries from both academia and industry. The multiple case design strengthens the external generalizability of the findings. Nevertheless, the conclusions obtained from our study are restricted to small- and medium-sized systems. Even so, we believe that much the same results can be expected in large systems. Our experience indicates that in case of large systems, when multiple advices apply to the same join point and when different aspects influence each other, modularity is even harder to achieve. The similar observation was reported by Kästner et al. [2007].

Finally, we could be criticised for generalizing findings from AspectJ to AOP. In our defence, most of the claims about the superiority of the AO modularization have been made in the context of AspectJ. We should also emphasise that AspectJ is the only production-ready general purpose AO language.

6.7 Related work

There are few studies focusing on the quantitative evaluation of the AO modularization. Sant'Anna et al. [2003] conducted a semi-controlled experiment to compare the use of an OO approach (based on design patterns) and an AO approach to implement Portalware (about 60 modules and over 1 KLOC), a multi-agent system. Portalware is a web-based environment that supports the development and management of Internet portals. The collected metrics show that the AO version incorporates modules with higher coupling and lower cohesion. Their coupling metric is broader than the original CBO in the sense that it additionally counts modules declared in formal parameters, return types, throws declarations and local variables. However, it is not complete, since it does not take into account either the semantic dependencies, or the dependency that occurs when an advice refers to a pointcut defined in other, non-ancestor module.

The same suite of metrics was used by Garcia et al. [2005] to compare the AO and OO implementations of the Gang-of-Four design patterns. They performed two studies, one on the original implementations from Hannemann & Kiczales and the other on the implementations with introduced changes. These changes were introduced because the H&K implementations encompassed few participant classes to play pattern roles [Garcia et al., 2005]. Garcia and his team concluded that "the use of aspects helped to improve the coupling and cohesion of some pattern implementations." However, such conclusion may be misleading, according to the metrics they collected. The measures before the application of the changes exhibit that only Composite and Mediator present lower coupling for the AO solutions. The implementations of Adapter and State have the same coupling in both paradigms. In the case of the other patterns, the OO solutions indicate lower coupling. The superiority of OO solutions decreased a little after the changes were introduced. Although the AO implementations of Observer, Chain of responsibility, State and Visitor became better with respect to coupling than their OO counterparts, there are still 16 patterns for which the OO implementations provide superior results. With regard to cohesion, the OO implementations were also superior in most cases. They analyzed the absolute (aggregate) values.

Other studies can be classified into 2 groups. In the first group [Filho et al., 2006; Greenwood et al., 2007; Madeyski & Szała, 2007; Figueiredo et al., 2008; Castor et al., 2009], new kinds of coupling introduced by pointcuts are not

considered at all. In the second group [Tsang et al., 2004; Hoffman & Eugster, 2007], the coupling introduced by a pointcut is considered only if a module is explicitly named by the pointcut expression.

Figueiredo et al. [2008] designed and implemented seven change scenarios for MobileMedia. MobileMedia is a software product line for applications (about 3 KLOC) that manipulate photo, music, and video on mobile devices. The absolute (aggregate) values collected to the coupling and cohesion metrics have favored the OO version for every release. After dividing these values by the number of modules, it turns out that AO versions are superior.

Greenwood et al. [2007] chose the Health Watcher system as the base for their study. Their evaluation focused upon ten releases of the system, which underwent a number of typical maintenance tasks, including: refactorings, functionality increments, extensions of abstract modules and more complex system evolutions. Some of the crosscutting concerns were "aspectized" from the first release, while others were modularized as new HW versions were released. They found that modularity was improved with AOP. The average "coupling" as well as cohesion were enhanced by 17% in the initial version, and by 23% and 21% in the 10th release.

Madeyski & Szała [2007] examined the impact of AOP on software development efficiency and design quality in the context of a web-based manuscript submission and a review system (about 80 modules and 4 KLOC). Three students took part in their study. Two of them developed the system (labeled as OO1 and OO2) using Java, whilst one implemented the system using AspectJ. The observed results show that the AO version is 24% better than the others with regard to average "coupling" and it is 60% (3%) better than OO1 (OO2) with regard to average cohesion.

Filho et al. [2006; Castor et al., 2009] refactored to AOP four systems: Telestrada, Pet Store, CVS, and EImp. The average "coupling" was decreased by 6%, 9%, and 1% for the first three systems and increased by 2% for the last system. Nevertheless, Filho et al. [2006] were aware that their study missed some coupling dependencies introduced by AOP: "a closer examination on the code (...) reveals a subtle kind of coupling that is not captured by the employed metrics." The Telestrada and Pet Store systems were also used by Hoffman & Eugster [2007]. In their study, Hoffman & Eugster calculated two coupling metrics, namely CBM and

CIM. However, since CBM and CIM are not simply additive, the results are difficult to interpret.

Tsang et al. [2004] compared AO vs. OO solutions in the context of real time traffic simulator. They found that aspects improved modularity by reducing "coupling" and cohesion. They considered aspects coupled to classes only if the aspects explicitly named the classes. "For instance, if we have the joinpoint call(* *(..)), then the aspect is not coupled to any classes. However, if we have the joinpoint call(void Test.methodName(..)), then the aspect is coupled to Test." In the conclusion of their work, they recommend the use of wildcards to maximize modularity improvements. Following this reasoning, one could recommend to replace the previous pointcut by call(void Test.methodNam*(..)), where '*' instead of 'e' eliminates "coupling".

Kouskouras et al. [2008] built an emulator of a telecommunications exchange, allowing the user to configure it with commands and to emulate simple calls between subscribers. They developed three different implementation alternatives. The first one follows a naive solution in Java, the second makes use of the Registry pattern and the third applies AspectJ to implement the same pattern. Next, they extended each implementation with several new commands and parameters. They applied the Martin's metrics suite to assess and compare design alternatives. Since they made the source code available for us, we could apply our metrics. The AO implementation is 7% worse than the improved OO implementation with regard to average coupling and it is 1% better with regard to average cohesion.

6.8 Summary

This chapter presented a quantitative study in which we compared OO and AO implementations of 11 real-life systems and the 23 GoF design patterns with respect to modularity. The evaluation was performed using CBO_{AO} that we proposed in the previous Chapter and LCOM that was adapted to AOP by Ceccato & Tonella [2004]. We found that the OO implementations of 10 real-life system exhibited lower coupling. We also found that there was no pattern whose AO implementations exhibited lower coupling, while 22 patterns presented lower coupling in the OO implementations. With the help of Dependency Structure

Matrix we analyzed in detail the coupling dependencies between modules of the Observer pattern. The impact of AOP on cohesion remains unclear.

Chapter 7. Impact of aspect-oriented programming on systems evolution and software reuse

There are two ways of constructing a software design. One is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

Hoare, 1981

The aim of this chapter is twofold. First, to compare AO and OO software with respect to evolvability and reusability. Second, to investigate the possibilities of applying AspectJ with generics and reflective programming to improve implementations of the GoF design patterns.

7.1 Development of a producer-consumer system

7.1.1 Research methodology

The difficulty of performing evolvability and reusability evaluation in AOP is that there are not yet industrial maintenance reports for AO software projects available for analyses. Thus, we have to simulate maintenance tasks in a quasi-controlled experiment. Then, we can measure how much effort is required to evolve the system and how much of the existing code is reused in the consecutive releases. The goal of our experiment is to compare AO and OO implementations of a producer-consumer system that undergoes five functionality increments (Figure 7.1). To measure software evolvability and reusability we use the metrics that we proposed in Chapter 5.

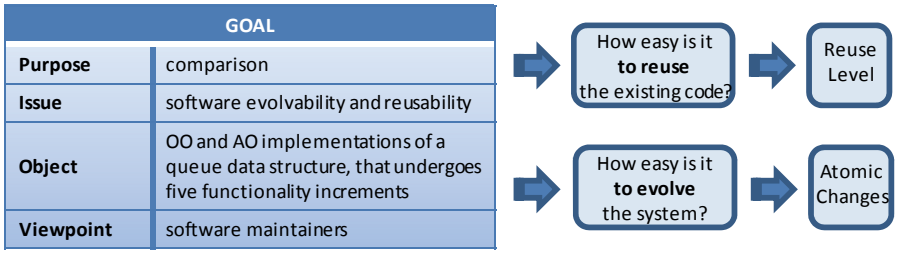


Figure 7.1 GQM diagram of the study.

7.1.2 The producer-consumer system

In a producer-consumer system two processes (or threads), one known as the "producer" and the other called the "consumer", run concurrently and share a fixed-size buffer. The producer generates items and places them in the buffer. The consumer removes items from the buffer and consumes them. However, the producer must not place an item into the buffer if the buffer is full, and the consumer cannot retrieve an item from the buffer if the buffer is empty. Nor may the two processes access the buffer at the same time to avoid race conditions. If the consumer needs to consume an item that the producer has not yet produced, then the consumer must wait until it is notified that the item has been produced. If the buffer is full, the producer will need to wait until the consumer consumes any item.

We assume to have an implementation of a cyclic queue as shown in Figure 7.2. The put(..) method stores one object in the queue and get() removes the oldest one. The nextToRemove attribute indicates the location of the oldest object. The location of a new object can be computed using nextToRemove, numItems (number of items) and buf.length (queue capacity). We also have an implementation of a producer and a consumer.

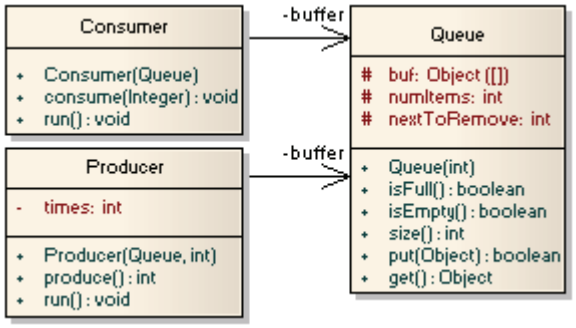


Figure 7.2 An initial implementation.

The experiment encompasses five maintenance scenarios which deal with the implementation of a new requirement. We have selected them because they naturally involve the modification of modules implementing several concerns.

7.1.3 Maintenance scenarios

Stage I: adding a synchronization concern

To use Queue in a consumer-producer system an adaptation to a concurrent environment is required. A thread has to be blocked when it tries to put an element

into a full buffer or when it tries to get an element from an empty queue. In addition, both put(..) and get() methods have to be executed in mutual exclusion. Thus, they have to be wrapped within synchronization code when using Java (Listing 7.1). Since the code supporting the secondary concern (i.e. synchronization) may throw an exception, there is also a technical concern of error handling. The core concern here is associated with adding and removing item from the buffer. The presented implementation tangles the code responsible for the core functionality with the code responsible for handling errors and for cooperating synchronization. Moreover, the implementation of both secondary concerns are scattered through the accessor methods. As a result, the put(Object) and get() methods contain similar fragments of code.

```
public class Buffer extends Queue {
  public Buffer(int n) { super(n); }
  public synchronized boolean put(Object x) {
    while ( isFull() ) try {
      wait();
    } catch (InterruptedException e) {
      System.out.println(e); }
    super.put(x);
    notifyAll();
    return true;
  }
  public synchronized Object get() {
    while ( isEmpty() ) try {
      wait();
    } catch (InterruptedException e) {
      System.out.println(e); }
    Object tmp = super.get();
    notifyAll();
    return tmp;
  }
}
```


Listing 7.1 A new class for Stage I

Lexical separation of concerns can be achieved by using AO constructs (Listing 7.2). The secondary concerns are implemented in ErrorHandler and SynchronizedQueue. SynchronizedQueue::waiting() is a hook method to introduce an explicit extension point. This joinpoint is used by ErrorHandler to wrap wait() invocation. Despite of lexical separation, SynchronizedQueue is explicitly tied to the Queue class, and so cannot be reused in other contexts. Moreover, Queue is oblivious of SynchronizedQueue. This makes it difficult to know what changes to Queue will lead to undesired behavior.

```
public aspect ErrorsHandler {
```

```
  protected pointcut waiting():
   execution(void SynchronizedQueue.waiting());

  void around(): waiting() {
    try {
      proceed();
    } catch (InterruptedException e) {
      System.out.println(e);
    }
  }

  declare soft: InterruptedException:waiting();
}

public aspect SynchronizedQueue pertarget(instantiation()) {

  protected pointcut instantiation(): target(Queue);

  protected pointcut call_get():execution(Object Queue.get());

  protected pointcut call_put(Object x):
   execution( boolean Queue.put(Object) ) && args(x);

  protected void waiting() { wait(); }

  Object around(Queue q): call_get() && target(q){
    synchronized(this) {
      while( q.isEmpty() ) waiting();
      Object tmp = proceed(q);
      notifyAll(); return tmp;
    }
  }

  boolean around(Queue q, Object x):call_put(x) && target(q){

    synchronized(this) {
      while (q.isFull()) waiting();
      proceed(q,x);
      notifyAll(); return true;
    }
  }
}
```

Listing 7.2 New aspects for Stage I

Stage II: adding a timestamp concern

After implementing the buffer a new requirement has occurred – the buffer has to save current time associated with each stored item. Whenever an item is removed, the time how long it was stored should be printed to standard output. A Java programmer may use inheritance and composition as reuse techniques (Listing 7.3). The problem is that three different concerns are tangled within put/get and so these concerns cannot be composed separately. It means that e.g. if a programmer wants a queue with timing he cannot reuse the timing concern from TimeBuffer; he has to reimplement the timing concern in a new class that extends Queue.

```
public class TimeBuffer extends Buffer {
  protected Queue delegateDates;
  public TimeBuffer(int capacity) {
    super(capacity);
    delegateDates = new Queue(capacity);
  }
  public synchronized boolean put(Object x) {
    super.put(x);
    delegateDates.put(
      new Long(System.currentTimeMillis()) );
    return true;
  }
  public synchronized Object get() {
    Object tmp = super.get();
    Long date = (Long) delegateDates.get();
    long curr = System.currentTimeMillis();
    System.out.println(curr - date.longValue());
    return tmp;
  }
}
```

Listing 7.3 The TimeBuffer class

A slightly better solution seems to be using AOP and implementing the timing as an aspect (Listing 7.4). Unless explicitly prevented, an aspect can apply to itself and can therefore change its own behavior. To avoid such situations, the instantiation pointcut is guarded by !cflow(within(Timing)). Moreover, the instantiation pointcut in SynchronizedQueue has to be updated. It must be the same as in Timing. This can be done only destructively, because AspectJ does not allow for extending concrete aspects.

```
public privileged aspect Timing pertarget( instant() ) {
  protected Queue delegateDates;
  protected pointcut instant():
   target(Queue) &&! cflow( within(Timing) );
  protected pointcut init(Queue q):
   execution( Queue.new(..) ) && target(q);
  protected pointcut execution_get():
   execution( Object Queue.get() );
  protected pointcut execution_put():
   execution( boolean Queue.put(Object) );
  after(Queue q): init(q) {
    delegateDates = new Queue(q.buf.length);
  }
  after(): execution_get() {
    Long date = (Long) delegateDates.get();
    System.out.println(
     System.currentTimeMillis() - date.longValue() );
  }
  after(): execution_put() {
    delegateDates.put(new Long(System.currentTimeMillis()));
  }
}
```

Listing 7.4 The Timing aspect.

Stage III: adding a logging concern

The buffer has to log its size after each transaction. The OO mechanisms like inheritance and overridden allow a programmer for reusing TimeBuffer (Figure 7.3). The only problem is that four concerns are tangled within put/get. A module that addresses one concern can generally be used in more contexts than one that combines multiple concerns.

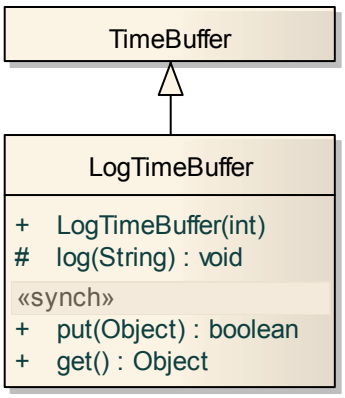


Figure 7.3 A new class for Stage III

The AO solution is also noninvasive and it reuses the modules from the earlier stages. It just requires defining a new aspect (Listing 7.5). When advice

declarations made in different aspects apply to the same join point, then by default the order of their execution is undefined. Thus, the declare precedence statement is used to force timing to happen before logging. The bufferChange pointcut enumerates, by their exact signature, all the methods that need to captured. Such pointcut definition is particularly fragile to accidental join point misses. An evolution of the buffer will require revising the pointcut definition to explicitly add all new accessor methods to it.

```
public aspect Logging {
  declare precedence : Logging, Timing;
  pointcut bufferChange(): !cflow(within(Timing)) &&
    (execution(* Queue.get()) || execution(* Queue.put(..)) );
  after(Queue q): bufferChange() && target(q) {
    System.out.println("buffer size: " + q.size());
  }
}
```

Listing 7.5 The Logging aspect

Stage IV: adding a new getter

The buffer has to provide a method to get “N” next items. There is no efficient solution of this problem neither using Java nor AspectJ. In both cases, the condition for waiting on an item has to be reinforced by a lock flag. A lock flag is set when some thread initiates the “get N” transaction by getting the first item. The flag is unset after getting the last item. In Java (Listing 7.6), not only does the synchronization concern has to be reimplemented but also logging. The reason is that in LogTimeBuffer logging is tangled together with synchronization, so it cannot be reused separately. The duplicate implementation might be a nightmare for maintenance.

```
public class EnhancedLogTimeBuffer extends TimeBuffer {
  protected boolean lock;
  public EnhancedLogTimeBuffer(int capacity) {
    super(capacity);
  }
  protected void lock(boolean b) { lock = b; }
  protected boolean isLock() { return lock; }
  protected void log(String s) {
    System.out.println(s);
  }
  public synchronized boolean put(Object x) {
    super.put(x);
    log("buffer size: "+size());
    return true;
  }
  public synchronized Object get() {
    while ( isEmpty()||isLock() ) try {
      wait();
    } catch (InterruptedException e) {}
    Object tmp = super.get();
    log("buffer size: "+size());
    return tmp;
  }
  public synchronized Object[] get(int n) {
    while ( isEmpty()||isLock() ) try {
      wait();
    } catch (InterruptedException e) {}
    lock(true);
    Object[] tmp = new Object[n];
    for(int i=0; i<n; i++) {
      tmp[i] = super.get();
    }
    lock(false);
    log("buffer size: "+size());
    return tmp;
  }
}
```

Listing 7.6 A new class for Stage IV

In AspectJ, although synchronization is implemented in a separate module, it also cannot be reused in any way because an aspect cannot extend another concrete aspect. Thus, all code corresponding to the synchronization concerns has to be reimplemented (Listing 7.7). A new method to get N items and locking mechanism are introduced to Queue by means of inter-type declaration.

```
public aspect EnhancedSynchronizedQueue pertarget(instant()){
  private boolean Queue.lock = false;
  public void Queue.lock(boolean b) { lock = b; }
  public boolean Queue.isLock() { return lock; }
  public synchronized Object[] Queue.get(int n) {
    while ( isEmpty()||isLock() ) waiting();
    lock(true);
    Object[] tmp = new Object[n];
    for(int i=0; i<n; i++) {
      while ( isEmpty() ) waiting();
      tmp[i] = get();
    }
    lock(false);
    return tmp;
  }
  private void Queue.waiting() { wait(); }
  protected void waiting() { wait(); }
  protected pointcut instant():
   target(Queue) && !cflow( within(Timing) );
  protected pointcut call_get(): call(Object Queue.get()) &&
   !cflow( withincode(* Queue.get(int)) );
  Object around(Queue q):call_get() && target(q) {
    synchronized(this) {
      while(q.isEmpty()||q.isLock()) waiting();
      Object tmp=proceed(q);
      notifyAll(); return tmp;
    }
  }
  declare precedence :
    EnhancedSynchronizedQueue, Logging, Timing;
  //...
}
```

Listing 7.7 A new aspect for Stage IV

In addition, destructive changes in the Logging::bufferChange() pointcut are required (Listing 7.8). Otherwise logs would be reported n times in response to the get(int n) method, instead of just once after completing the transaction. This is due to that get(int n) uses get() for retrieving every single item from the buffer. Furthermore, the ErrorsHandler::waiting() pointcut also needs adjusting to the new decomposition.

```
public aspect Logging {

  pointcut bufferChange():
    !cflow( within(Timing) ) &&
    !cflow( withincode(* Queue.get(int)) ) &&
    ( execution( * Queue.get(..) ) ||
      execution( * Queue.put(..) ) );
  //...
}

public aspect ErrorsHandler {

  protected pointcut waiting():
    execution( void EnhancedSynchronizedQueue.waiting() )
    || execution( void Queue.waiting() );
  //...
}
```

Listing 7.8 Modifications in the pointcuts

Stage V: Removing logging and timestamp

A programmer needs the enhanced buffer from Stage IV, but without the logging and timing concerns. In Java, he once again has to reimplement the get(int) method and much of the synchronization concerns (Listing 7.9). All to do in the AO version is to remove Logging and Timing from the compilation list.

```
public class EnhancedBuffer extends Buffer {

  protected boolean lock;

  public EnhancedBuffer(int capacity) {
    super(capacity);
  }

  protected void lock(boolean b) { lock = b; }

  protected boolean isLock() { return lock; }

  public synchronized Object get() {
    while ( isEmpty()||isLock() ) try {
      wait();
    } catch (InterruptedException e) {}
    return super.get();
  }
  public synchronized Object[] get(int n) {
    while ( isEmpty()||isLock() ) try {
      wait();
    } catch (InterruptedException e) {}
    lock(true);
    Object[] tmp = new Object[n];
    for(int i=0; i<n; i++) {
      tmp[i] = super.get();
    }
    lock(false);
    return tmp;
  }
}
```

Listing 7.9 A new class for Stage V

7.1.4 Empirical results

Table 7.1 presents the number of Atomic Changes and Reuse Level for both releases for every stage. The measures were collected manually. Lower values are better for Atomic Changes but worse for Reuse Level. AOP manifests superiority at Stage III and V, while OOP in the rest of the cases. At Stage III we have implemented a logging concern which is one of the flagship examples of AOP usage. At this Stage, the OO version requires significantly more atomic changes and new lines of code than its AO counterpart. At Stage V, the maintenance tasks are focused on detaching some concerns instead of implementing new ones. The AO solution has turned out to be more pluggable.

Table 7.1 Number of Atomic Changes and Reuse Level per stage.

Stage	Atomic Changes		Reuse Level	
	OOP	AOP	OOP	AOP
I) Adding a synchronization concern	7	19	0,71	0,66
II) Adding a timestamp concern	8	19	0,85	0,67
III) Adding a logging concern	9	6	0,88	0,95
IV) Adding a new getter	9	16	0,73	0,58
V) Removing logging and timestamp	5	3	0,74	1,00

7.1.5 Lessons learned

In an AO system, one cannot tell whether an extension to the base code is safe[2] simply by examining the base code in isolation. All aspects referring to the base code need to be examined as well. In addition, when writing a pointcut definition a programmer needs global knowledge about the structure of the application. E.g. when implementing the Timing aspect, a programmer has to know that the current implementation of the synchronization concern affects each Queue structure, while the timing concern requires a non-blocking Queue.

Moreover, when a system includes multiple aspects, they can begin to affect each other. At Stage III, we have had to explicitly exclude logging the state of the queue that is used by the Timing aspect. Furthermore, we have observed the problem of managing interactions between aspects that are being composed. When advice declarations made in different aspects affect the same join point, it is

[2] in the sense that it does not break the aspect code

important to consider the order in which they execute. Indeed, a wrong execution order can break the program. In our experiment, we have used precedence declarations to force timing to happen before logging and to force both of them to happen within the synchronization block.

In most cases, aspects cannot be made generic, because pointcuts as well as advices encompass information specific to a particular use, such as the classes involved, in the concrete aspect. As a result, aspects are highly dependent on other modules and their reusability is decreased. E.g. at Stage I, the need to explicitly specify the Queue class and the two synchronization conditions means that no part of the SynchronizedQueue aspect can be made generic. In addition, we have confirmed that the reusability of aspects is also hampered in cases where "join points seem to dynamically jump around", depending on the context certain code is called from [Beltagui, 2003]. Moreover, the variety of pointcut designators makes pointcut expressions cumbersome (see EnhancedSynchronizedQueue::call_get()).

Some advocates of AOP believe that appropriate tools can deal with the problems of AOP we encountered. We think that they should reject AOP at all, since some research [Robillard & Weigand-Warr, 2005] "shows" that OOP with a tool support solves the problem of crosscutting concerns:)

7.1.6 Threats to Validity

Construct validity

Construction threats lie in the way we define our metrics. Evolvability and reusability like other quality factors are difficult to measure. Our dependent variables are based on previous studies performed by Zhang et al. [2008], Ryder & Tip [2001] and Frakes [1993]. It is possible that other metrics will be better fitted for the purpose of our study.

Internal validity

Internal validity of our experiment concerns the question whether the effects were caused only by the programming paradigm involved, or by other factors. The experiment has been carried out by the author during his research for the achievement of a Doctor of Philosophy Degree. As the author does not have any interest in favour of one approach or the other, we do not expect it to be a large threat. Nevertheless, other programmers could have chosen the different strategies for implementing secondary concerns.

External validity

Synchronization, logging, and timing present the typical characteristics of crosscutting concerns and as such they are likely to be generalizable to other concerns. Unfortunately, the limited number of maintenance tasks and size of the program make impossible the generalization of our results. However, the academic setting allows us to present the whole programs in detail and to put forward some advantages and limitations of AOP.

7.1.7 Related work

Coady & Kiczales [2003] compared the evolution of two versions (C and AspectC) of four crosscutting concerns in FreeBSD. They refactored the implementations of the following concerns in v2 code: page daemon activation, prefetching for mapped files, quotas for disk usage, and tracing blocked processes in device drivers. These implementations were then rolled forward into their subsequent incarnations in v3 and v4 respectively. In each case they found that, with tool support, the AO implementation better facilitated independent development and localized change. In three cases, configuration changes mapped directly to modifications to pointcuts and makefile options. In one case, redundancy was significantly reduced. Finally, in one case, the implementation of a system-extension aligned with an aspect was itself better modularized.

Bartsch & Harrison [2008] conducted an experiment in which 11 students were asked to carry out maintenance tasks on one of two versions (Java and AspectJ) of an online shopping system. The results did seem to suggest a slight advantage for the subjects using the OO version since in general it took the subjects less time to perform maintenance tasks and it averagely required less line of code to implement a new requirement. However, the results did not show a statistically significant influence of AOP at the 5% level.

Sant'Anna et al. [2003] conducted a quasi-controlled experiment to compare the use of OOP and AOP to implement Portalware (about 60 modules and over 1 KLOC). Portalware is a multi-agent system (MAS) that supports the development and management of Internet portals. The experiment team (3 PhD candidates and 1 M.Sc. student) developed two versions of the Portalware system: an AO version and an OO version. Next, the same team simulated seven maintenance/reuse scenarios that are recurrent in large-scale MAS. For each scenario, the difficulty of maintainability and reusability was defined in terms of structural changes to the artifacts in the AO and OO systems. The total lines of

code, that were added, changed, or copied to perform the maintenance tasks, equaled 540 for the OO approach and 482 for the AO approach.

Kulesza et al. [2006] present a quantitative study that assesses the positive and negative effects of AOP on typical maintenance activities of a Web information system.They compared the AO and OO implementations of a same web-based information system, called HealthWatcher (HW). The main purpose of the HW system is to improve the quality of services provided by the healthcare institution, allowing citizens to register complaints regarding health issues, and the healthcare institution to investigate and take the required actions. In the maintenance phase of their study, they changed both OO and AO architectures of the HW system to address a set of 8 new use cases. The functionalities introduced by these new use cases represent typical operations encountered in the maintenance of information systems. Although they claim that the AO design has exhibited superior reusability through the changes, there is no empirical evidence to support this claim. The collected metrics show only that aspects contributed to: (1) the decrease in the lines of code, number of attributes, and cohesion; (2) the increase in the vocabulary size and lexical separation of crosscutting concerns. They also tried to evaluate coupling using the Sant'Anna's metric [2003], but in Chapter 5.2.1 we have argued why this metric is invalid to compare between OO and AO implementations. An additional interesting observation from Kulesza's study [2006] is that more modules were needed to be modified in the AO version, because it requires changing both the classes along the layers to implement the use case functionality and the aspects implementing the crosscutting issues.

Munoz et al. [2008] showed that aspects offer efficient mechanisms to implement crosscutting concerns, but that aspects can also introduce complex errors in case of evolution. To illustrate these issues, they implemented and then evolved a chat application. They found that it is very hard to reason about the aspects impact on the final application.

Kouskouras et al. [2008] built an emulator of a telecommunications exchange, allowing the user to configure it with commands and to emulate simple calls between subscribers. They developed three different implementation alternatives. The first one follows a simplistic solution applying OOP. The second makes use of the Registry pattern. The third applies AOP to implement the Registry pattern. Next, they investigated the behavior of the designs at a specific extension scenario. The extension scenario involved the addition of several new

commands and parameters. Since they made the source code available for us, we could apply our metrics. The differences in Atomic Changes as well as Reuse Level between various versions of the system were less than 3%. Every version turned out to be very extensible and reusable. Almost all of the modules that were developed in the base release, were reused in the next release. Moreover, the atomic changes were not invasive because they mostly focused on creating new modules instead of modifying the existing ones.

Mortensen et al. [2010] examined the benefits of refactoring three legacy applications developed by Hewlett-Packard. They followed the evolution of the applications across several revisions. The modifications needed to evolve these systems required changes to fewer software items in the refactored systems when compared to the original. The reduction of the average number of modules and files changed between revisions was 4% and 3% respectively.

Taveira et al. conducted two studies to check if AOP promotes greater reuse of exception handling code than a traditional, OO approach. In the first study [Taveira et al., 2009], they assessed the suitability of AOP to reuse exception handling code within applications. They refactored three medium-size applications implemented originally in Java. Aspects were used to implement the exception handlers. Though AOP promoted a large amount of reuse of error handling code, the overall size of the refactored systems did not decrease due to the code overhead imposed by AspectJ. The number of handlers was sensibly lower in the refactored versions but the amount of error handling code was much higher. In the second study [Taveira et al., 2010], they refactored seven medium-size systems to assess the extent to which AOP promotes inter-application reuse of exception handling code. They found out that reusing error handling across applications is not possible in most of the cases and requires some a priori planning. Only extremely simple handlers could be reused across applications.

The experiment closest to ours is the one conducted by Figueiredo et al. [2008] in which they quantitatively and qualitatively assess the positive and negative impacts of AOP on a number of changes applied to MobileMedia. MobileMedia is a software product line for applications with about 3 KLOC that manipulate photo, music, and video on mobile devices. The original release was available in both AspectJ and Java (the Java versions use conditional compilation as the variability mechanism). Then, a group of five post-graduate students was responsible for implementing the successive evolution scenarios of MobileMedia.

Each new release was created by modifying the previous release of the respective version. A total of seven change scenarios were incorporated. The scenarios comprised different types of changes involving mandatory, optional, and alternative features, as well as non-functional concerns. Figueiredo et al. found that AOP usually does not cope with the introduction of mandatory features. The AO solution generally introduced more modules and operations. A direct result of more modules and operations is the increase in LOC. Moreover, depending on the evolution scenario, AspectJ pointcuts were more fragile than conditional compilation. In order to compare their and our results, we have derived the simplest form of Reuse Level and Atomic Changes (Table 7.2) from their measures. Atomic Changes has been limited to counting operations only, while Reuse Level has been calculated as: number_of_reused_LOC / LOC. In general, the measures demonstrate that there is no winner with respect to Reuse Level. The AO solution is significantly better only at Stage VII. With regard to Atomic Changes, the OO implementations are superior for every release.

Table 7.2 Atomic changes and Reuse Level in MobileMedia.

release►		II	III	IV	V	VI	VII	VIII
Reuse Level	**OO**	0,73	0,86	0,96	0,48	0,75	0,01	0,76
	AO	0,62	0,82	0,93	0,55	0,76	0,29	0,74
Atomic Changes	**OO**	120	68	20	111	88	335	149
	AO	150	90	22	134	102	437	175

7.2 Revision of the Gang-of-Four design patterns

7.2.1 Introduction

A critical challenge for software developers is to utilize design experience and reuse existing software when building new systems. Design patterns have proved effective in helping to address these issues. However, the solutions proposed in the original design pattern literature [Gamma et al., 1995] are shaped by techniques as well as language deficiencies from OOP. OO implementations of the design patterns are not themselves reusable software entities. Moreover, applying a certain design pattern typically means embedding them invasively into the program [Kniesel et al., 2004]. With the rise of AOP, new abstractions have occured that suggest it is time for these solutions to be revised.

Hannemann & Kiczales [2002] (H&K) developed AspectJ implementations of the 23 Gang-of-Four (GoF) patterns. However, AspectJ didn't support generics (generic types), when they were doing their research. Generic types let programmers define a type without specifying all the other types it uses. The unspecified types are supplied as parameters at the point of use. Generics were added to AspectJ in 2005. With the advent of this technique, a new support for more reusable implementations have occured. Hence, the solutions that have been presented so far should be revisited and reconsidered. In this research the existing AO implementations are examined according to applying generics and reflective programming.

7.2.2 Research methodology

In developing our solutions, we follow the guidelines developed by Hevner et al. [2004]. Table 7.3 discusses the realization of these guidelines in our work.

Table 7.3 Developing new solutions

No	Guideline	Realization
1	Design as an Artifact	The results of this research are new solutions and implementations of three design patterns. They are methods in Simon's terminology [Simon, 1996].
2	Problem Relevance	The relevance of design patterns in software development is well known [Gamma et al., 1995]. While efforts to use AspectJ to implement GoF patterns have been made, no one has tried to use AspectJ with generics or reflective programming.
3	Design Evaluation	The utility of the artifacts are demonstrated on examples.
4	Research Contributions	The contribution of this research is presented in Section 7.2.5.
5	Research Rigor	The artifacts are formally represented in AspectJ.
6	Design as a Search Process	We reviewed all 23 GoF patterns. It turned out that only Decorator and Proxy took advantages of generics, while applying reflective programming was beneficial for Prototype.
7	Communication of Research	Sections 5.2.3 – 5.2.5 motivates a developer audience.

7.2.3 The Decorator pattern

The intent of the Decorator pattern is to perform additional actions on individual objects [Gamma et al., 1995; Borella, 2003]. The additional actions and the decorated objects are selected at runtime. An alternative for this pattern is inheritance. However, the Decorator pattern has several advantages over subclassing. First, additional actions can be added and removed at runtime and per object. Second, combining independent extensions is easy by composing decorator objects. With subclassing every combination of extensions results in a new subclass and this can cause an explosion of subclasses.

There are many variations of the Decorator pattern, but in this paper the one used is that defined by Borella [2003]. The first AO approach to this pattern was presented by Hannemann & Kiczales [2002]. Their solution has three limitations [Borella, 2003]. Firstly, it does not allow for dynamic attaching and dynamic detaching of decorators. Secondly, it is not possible at runtime to define an order of activation among decorators. Thirdly, after inserting the decorator aspect in the system, all instances of the intended class are decorated.

Other solutions were proposed by Monteiro & Fernandes [2004] and Hachani & Bardou [2002] but these do not add anything new and so need not be considered. A significant contribution to implementing the Decorator pattern was made by Borella [2003]. He uses a per-target instantiation model to decorate only a subset of the instances of a class. His solution enables decorators to be composed dynamically. The order of activation of each decorator is given at runtime.

The Borella solution has two main imperfections. Firstly, the around advice and the wrap method are not generic, and depend on the type of the decorated object. Secondly, the decorator classes could directly implement the Decorator interface. Introducing the Decorator interface via the parent declaration unnecessarily complicates the solution.

Figure 7.4 presents our solution to the Decorator pattern. WrapperProtocol<E> describes the generic structure and behaviour that is determined by the pattern and it does not define any application specific behaviour. This solution reduces the non-reusable parts of the implementation. The around advices are responsible for intercepting objects which should be decorated and passing them to the wrap(E e) method as argument. This method iterates through all the registered decorators and gives them its argument to decorate (Listing 7.10).

After decoration, the argument is returned. A concrete decoration process is implemented in the StarDecorator and DollarDecorator classes.

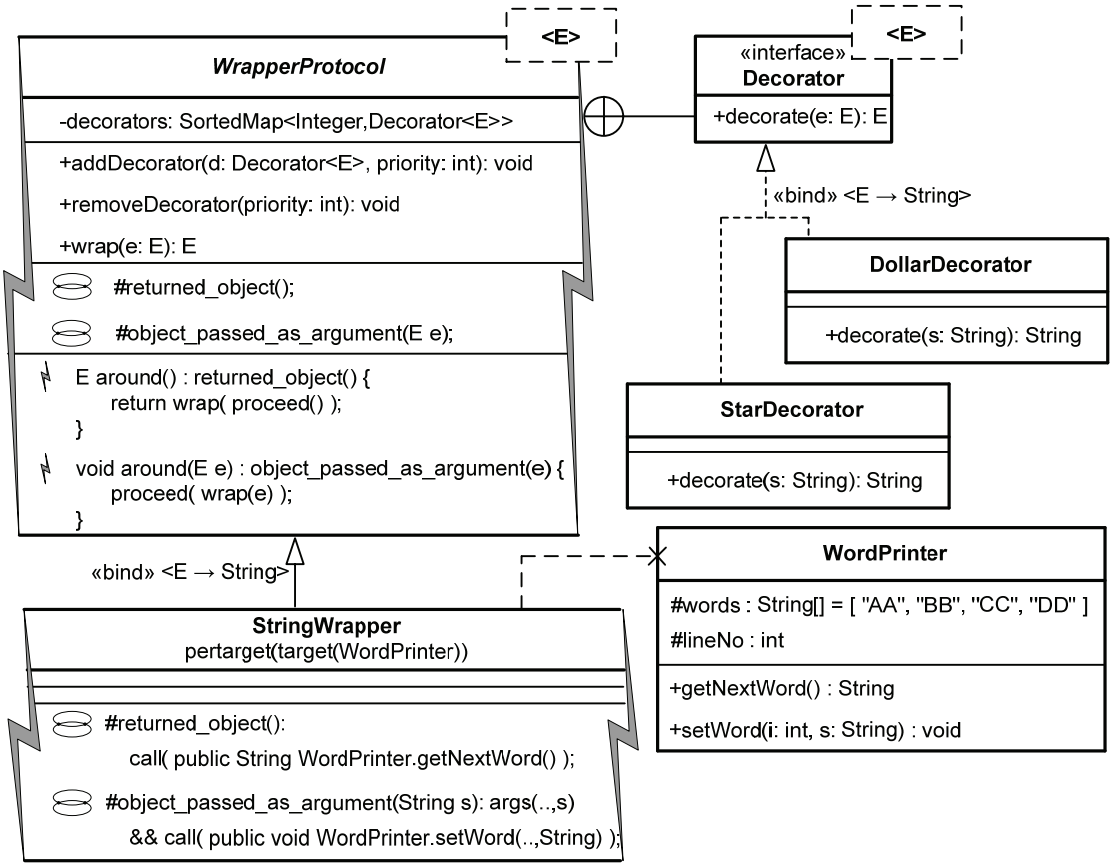


Figure 7.4 The Decorator pattern.

```
public E wrap(E e) {
  for ( Decorator<E> dec: decorators.values() )
    e=dec.decorate(e);
  return e;
}
```

Listing 7.10 The wrap method

The joinpoints at which the object to decorate should be captured are specified by the returned_object and object_passed_as_argument pointcuts. Thus it is possible to decorate the object, which is passed as an argument or returned as a result of a method call. The definitions of both pointcuts are empty, so at least one of them should be overridden in a subaspect.

The concrete subaspect, which knows what type of object is captured and in which context, has to be derived from WrapperProtocol<E> by giving a bound type to the E parameter. One of such subaspects is StringWrapper that binds the E parameter with String. StringWrapper intercepts requests to the getNextWord()

method and performs a decoration on the object returned by this method. An example of the use of StringWrapper is shown in Listing 7.11.

In order to decorate a specific object, the instance of StringWrapper that is associated with this object is retrieved (Line 2). Zero or more decorators are then attached to this instance (Lines 3 and 4). Without any decorator, the getNextWord() method would return "AA". However, the wrapper object has registered (Lines 3 and 4) the StarDecorator and DollarDecorator instances, which wrap the returned object with "***" and "$$$" respectively. As a result, the "*** $$$ AA $$$ ***" string is printed on the screen (Line 6).

```
public class testDecorator {
  public static void main(String[] args) {
    WordPrinter w = new WordPrinter();                //1
    StringWrapper wrapper = StringWrapper.aspectOf(w); //2
    wrapper.addDecorator( new StarDecorator(), 2 );   //3
    wrapper.addDecorator( new DolarDecorator(), 1 ); //4
    w.setWord(0,"XXX");                               //5
    System.out.println( w.getNextWord() );            //6
  }
}
```

Listing 7.11 A use of the Decorator pattern.

7.2.4 The Proxy pattern

The proxy pattern allows the developer to provide a surrogate object in place of the actual object in case access to the real object needs to be delegated or controlled. The following are some of the more common uses for proxies [Grand, 2002]:

- Represent an object that is complex or time consuming to create with a simpler one.
- Create the illusion that an object on a different machine is an ordinary local object.
- Control access to a service-providing object based on a security policy.
- Create the illusion that a service object exists before it actually does.

The structure of this pattern, that uses generics, is shown on Figure 7.5. ProxyProtocol<Subject> is a reusable part of the implementation. Subject is a parameter. The client binds this parameter with the type of the object to be “proxied”. The requestsToSubjectByCaller pointcut intercepts calls to the subject. If a call is proxy protected, the handleProxyProtection method is called instead of the original method. The isProxyProtected method checks whether the request should be handled by the proxy or not. By default it returns true. The

handleProxyProtection method provides an alternative return value if a call is proxy protected. The default implementation returns null. Concrete subaspects are forced to implement the requests pointcut. This pointcut defines which requests need to be handled by the proxy.

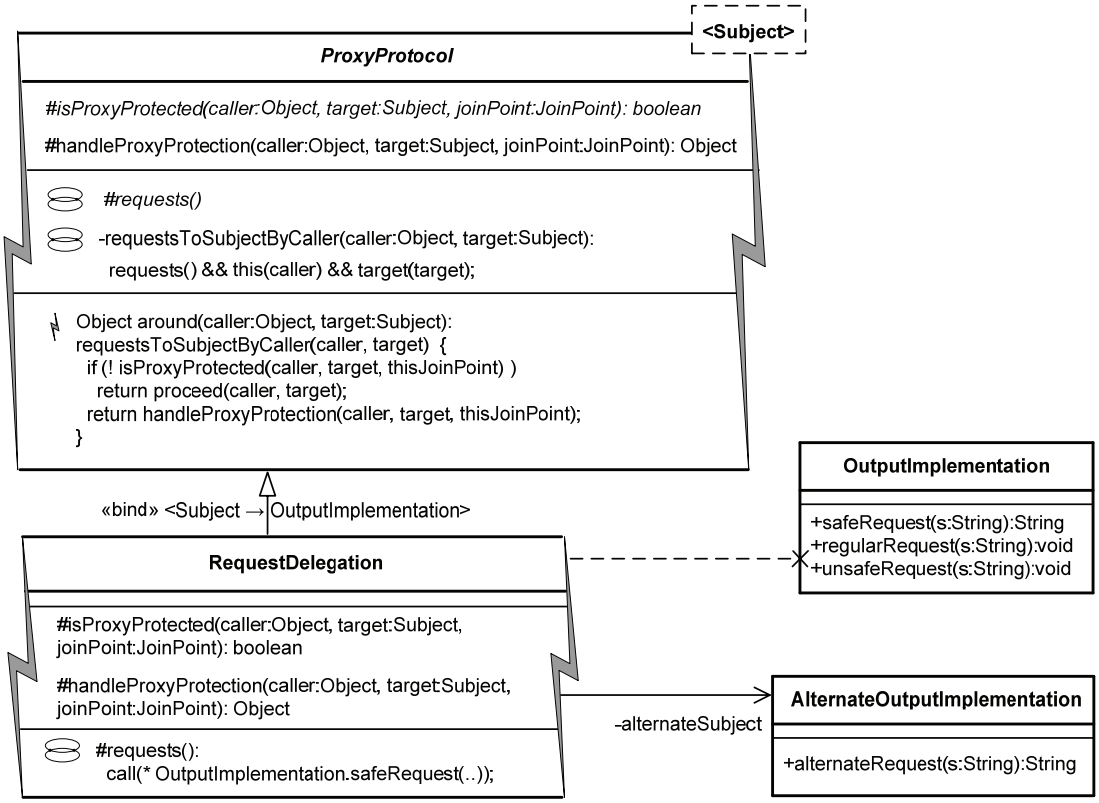


Figure 7.5 The Proxy pattern.

7.2.5 The Prototype pattern

The Prototype pattern allows an object to create a copy of itself without knowing its direct class. This pattern can avoid expensive "creation from scratch". The most important requirement for objects to be used as prototypes is that they have a method, typically called copy, that returns a new object that is a copy of the original object. How to implement the copy operation for the prototypical objects is another important implementation issue. There are two basic strategies for implementing the copy operation [Grand, 2002]:

- Shallow copying means that the attributes of the cloned object contain the same values as the attributes of the original object and that all object references indicate the same objects.
- Deep copying means that the attributes of the cloned object contain the same values as the attributes of the original object, except that attributes

that refer to objects refer to copies of the objects referred to by the original object. In other words, deep copying also copies the objects that the object being cloned refers to. Implementing deep copying can be tricky. You will need to be careful about handling any circular references.

Shallow copying is easier to implement because all classes inherit a clone method from the Object class that does just that. However, unless an object's class implements the Cloneable interface, the clone method will throw an exception and will refuse to work. This work presents the first strategy with some modification (Figure 7.6).

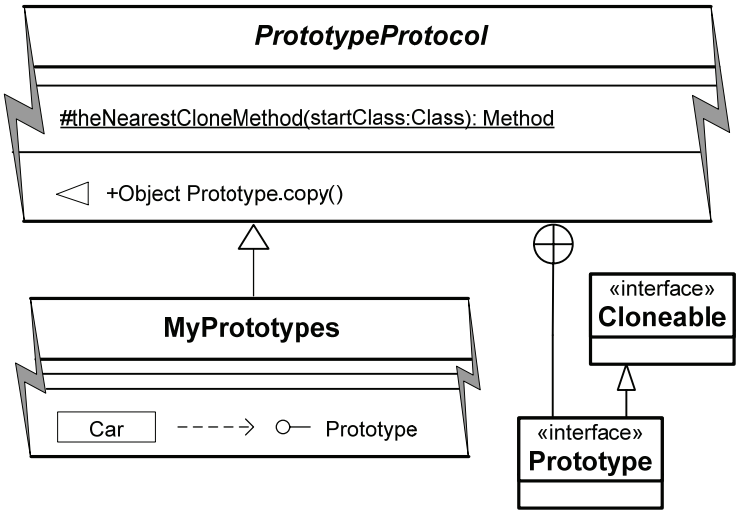


Figure 7.6 The Prototype pattern.

The PrototypeProtocol aspect attaches a default copy() method on all Prototype participants. The implementation of that method is (1) to find the nearest clone() method up the class hierarchy, (2) to invoke it, and (3) to return the result (Listing 7.12). The searching process starts with the runtime class of the object to which the copy() request was sent. If that class does not define the clone method, then the search is made in its superclass. In the worst case, the search repeats until the Object class is reached. MyPrototypes assigns the Prototype interface to Car.

```
privileged public aspect PrototypeProtocol {
  public interface Prototype extends Cloneable{}
  public Object Prototype.copy() {
    Object copy = null;
    Method cloneMethod=null;
    try {
      Class thisClass = ((Object) this).getClass();
      cloneMethod = theNearestCloneMethod(thisClass);
      cloneMethod.setAccessible(true);
      copy = cloneMethod.invoke(this, null);
    } catch(Exception ex) {
      System.out.println(ex);
    }
    return copy;
  }
  protected static Method theNearestCloneMethod(
   Class startClass) {
    Method cloneMethod=null;
    do {
      try {
        cloneMethod =
          startClass.getDeclaredMethod("clone", null);
      } catch(NoSuchMethodException ex) {
        startClass = startClass.getSuperclass();
      }
    } while( cloneMethod == null );
    return cloneMethod;
  }
}
```

Listing 7.12 The PrototypeProtocol aspect

7.2.6 Discussion

This research presents how AO implementations of the Decorator and the Proxy pattern can be improved using generics. Parametrized aspects, which serve as protocols, were created for both patterns. For Decorator not only implementation but also a new solution was developed. The solution is simpler then the one proposed by Borella, whereas the implementation is much more reusable. The only thing a programmer has to implement is two pointcuts. For the Proxy pattern, Hannemann & Kiczales's solution was used, and only the implementation was adapted. Moreover, AOP was combined with reflective programming to provide a default implementation for cloning in the Prototype pattern. All the implemented patterns can be plugged or unplugged without any changes in the existing modules. We also confirm the observation of Bartsch & Harrison [2007] that the question of which implementation to prefer is a trade-off between a higher cognitive complexity and the reusability of the abstract pattern protocol.

7.2.7 Related work

First attempts to reshape design pattern solutions based on AOP were initiated by Hannemann & Kiczales [2002] (H&K) and then continued by Hachani & Bardou [2002], Borella [2003], Monteiro & Fernandes [2004], and Denier, Albin-Amiot & Cointe [2005]. H&K implemented the 23 GoF design patterns in two layers. One abstract layer defined a pattern protocol, i.e. an abstract aspect which localizes all code that belongs to an abstract view of a pattern [Bartsch & Harrison, 2007]. This view defines the basic logic of a pattern and can be reused in different instances of the same pattern [Hannemann & Kiczales, 2002]. The assignment of classes to the roles is done in the aspect that defines a specific pattern instance (concrete layer). For several patterns, H&K found reusable implementations.

7.3 Summary

The first part of the Chapter presented a quasi-laboratory experiment in which we compared the evolution of a simple program that underwent five maintenance scenarios. The superiority of AOP was observed only when detaching secondary concerns and when implementing logging, which is a flagship example of AOP usage. OOP fared better in implementing secondary concerns in three out of four scenarios. Moreover, by reviewing other research, we showed that the claims that AOP improves software maintability and reusability are not backed up by any convincing evidence.

In the second part of the Chapter we presented the results of exploring the existing AO implementations according to applying generics and reflective programming. It was found that Decorator and Proxy are suitable to use with generics, while Prototype is suitable to use with reflection. In each case, the applied programming techniques enhanced reusability of not only the application core part but also the design pattern part.

The scope of our research was too limited to draw any definitive conclusions about the impact of AOP on software evolvability and reusability. We can only say that there are limited situations where a reasonable AO implementation improves software evolvability and reusability.

Chapter 8. Summary

The search for truth is more precious than its possession.
Albert Einstein

8.1 Conclusions

In the scientific community, AOP has been often claimed to improve software modularity compared to OOP. However, this dissertation denies it on empirical (Chapter 6) as well as theoretical (Section 3.3) grounds. We have demonstrated that AOP is in a contrary to the principles of modularity established by Parnas, Dijkstra, Yourdon, Constantine, Meyer and other greats of the past. Thus, we formulated the first part of our thesis statement as:

III. Aspect-oriented programming allows for lexical separation of crosscutting concerns, but it violates the fundamental principles of modular design, such as low coupling, information hiding, and explicit interfaces.

Another promise of AOP is to improve software evolvability and reusability. Since high-level quality attributes are even more important for software developers, we have also made efforts to assess the extent to which AOP promotes software reuse and systems evolution (Chapter 7). Unfortunately, this area of research within the AOP community is somewhat restricted by the lack of available AOP-based projects that include adequate maintenance/reuse documentation. Thus, we have been able to conduct only preliminary research in a laboratory environment. This research consists of two parts. In the first part, we have performed a quasi-controlled experiment in which OOP has generally fared better in implementing crosscutting concerns. In the second part, we have reviewed the GoF design patterns from the perspective of applying AspectJ with generics and reflective programming. We have found more reusable implementation for Decorator, Proxy, and Prototype in comparison to their OO counterparts. Although we have complemented the existing knowledge on AOP we cannot draw any definitely conclusions about the impact of AOP on software evolvability and reusability, due

to the limited scope of our research. Nevertheless, our observations support the second part of our thesis statement:

IV. There are limited situations where a reasonable aspect-oriented implementation improves software evolvability and reusability.

8.2 Contributions

8.2.1 Evaluating the impact of AOP on software modularity

We have discussed the novel kinds of coupling dependencies introduced by AOP and proposed the CBO_{AO} metric. CBO_{AO} can be applied to AO as well as OO software and generates comparable results. At the same time, we have argued that since the existing coupling metrics for AOP do not cover all the possible kinds of coupling dependencies, the results obtained with these metrics are underestimated. To know the real results, we have re-evaluated the existing systems.

We have compared OO and AO implementations of 11 real-life systems and the 23 GoF design patterns. In only one case we have found evidence that the AO implementation results in lower coupling than its OO counterpart. We are the first who experimentally demonstrated this effect. The impact of AOP on cohesion remains unclear.

Since our metrics do not cover all aspects of software modularity, we have also surveyed other criteria and found that AOP is fundamentally at odds with the basic principles on which software engineering has depended for the last 50 years. In Section 3.3 we argue that:

- AOP promotes unstructured programming
- AOP breaks information hiding
- AOP leaves interfaces implicit
- AOP makes modular reasoning difficult
- AOP breaks the contract between a base module and its clients
- AOP escalates coupling

We have also examined the studies that propose to reduce obliviousness as a trade-off for an increase in modularity. We have found that in these approaches AOP

loses its ability to add new features to the code without having to intrusively modify the code.

8.2.2 Exploring the possibilities of AOP in the context of software reuse and evolvability

We have performed some initial experiments on software reuse and evolution. Firstly, we have compared OOP with AOP on a small program that undergoes five functionality increments. The superiority of AOP has been observed only when detaching crosscutting concerns and when implementing logging, which is a flagship example of AOP usage. OOP has fared better in implementing crosscutting concerns in three out of four scenarios. The lessons learned from this study are as follows. In a AO system, one cannot tell whether an extension to the base code is safe simply by examining the base program in isolation. All pointcuts referring to the base program need to be examined as well. In addition, when writting a pointcut definition a programer needs a global knowledge about the structure of the application. This is due to the fact that pointcuts try to define intended conceptual properties about the base program, based on structural properties of the program. As a consequence it may lead to the pointcut fragility problem that causes ripple effects during system evolution. In most cases, aspects cannot be made generic, because pointcuts as well as advices encompass information specific to a particular use, such as the classes involved, in the concrete aspect. As a result, aspects are highly dependent on other modules and their reusability is decreased. Futhermore, we have confirmed that the reusability of aspects is also hampered in cases where “join points seem to dynamically jump around”, depending on the context certain code is called from.

Secondly, we have explored the existing AO implementations of the GoF patterns according to applying generics and reflective programming. We have found that Decorator and Proxy are suitable to use with generics, while Prototype is suitable to use with reflection. In each case, the applied programming techniques enhanced reusability of not only the application core part but also the design pattern part. Moreover our solutions make an application independent of the design patterns which can be plugged or unplugged depending on the presence or absence of aspects. The impact of the AO solutions on software maintainability is not obvious. On the one hand, localizing pattern related code might be a step towards increasing the maintainability of design patterns [Bartsch & Harrison, 2007]. On

the other hand, a higher cognitive complexity of the AO implementations decreases comprehensibility of design patterns.

8.2.3 Elaborating an extension to the UML metamodel for Aspect Oriented Modeling

We have presented an approach to integrate aspect orientation with the current state-of-the-art in modelling languages. The elaborated metamodel (AoUML) enriches UML with constructs for visualizing aspects. Thus we have improved the traceability from design to implementation by reducing the semantic gap between these development phases. We have useed AoUML in other parts of our dissertation.

8.3 Evaluation of the results

The work presented in this dissertation has been published in the proceedings of nine international conferences and one book's chapter. Each of the publication is summarised below:

- [Przybyłek, 2007] gives an introduction to the problem of implementing crosscutting concerns in OO languages. The limitations of OO languages are explained and illustrated by 3 scenarios of adapting software to new requirements. Then, the paper presents the state-of-the-art in implementing crosscutting concerns. The basic concepts of AOP and CF's are explored and applied to the scenarios to avoid code scattering and code tangling. Finally, the paper suggests programming guidelines that assist programmers in deciding when to use what paradigm.
- [Przybyłek, 2009] is an extension of [Przybyłek, 2007]. It surveys SoC and modularization techniques from the days of structured programming to post-OO paradigms of today's academic research. It also explains the „tyranny of the dominant decomposition" problem and the limitations of mainstream languages associated with it. Then, it outlines new programming constructs offered by AOP and CFs to avoid these limitations. Finally, the paper discusses the lessons learned from developing software in the post-OO paradigms. It suggests areas of future work related to post-OO paradigms.

- [Przybyłek, 2008a] focuses on the design phase. It presents a new modelling language named AoUML that we elaborated to incorporate aspects into class diagram. AoUML is an extension to the UML metamodel. Its specification is described by using a similar style to that of the UML metamodel. First, an overview class diagram is introduced to show the constructs that was included in the extension and how these constructs are built up in terms of the standard UML constructs. Then, the semantics and syntax are described in detail using natural language. The practical applicability of AoUML is demonstrated by visualizing three GoF design patterns.
- [Przybyłek, 2010c] discusses whether AOP makes software more modular. First, we briefly review the literature of software engineering related to modularity and SoC techniques. Next, we show that AOP is in conflict with the well-established principles of modular design that Parnas, Dijkstra and other greats of the past laid out and on which software engineering has depended for the last 50 years.
- [Przybyłek, 2010a] presents a systematic case study in which we compare OO and AO implementations of the 23 GoF design patterns with respect to modularity. We advocate for measuring modularity with the help of coupling and cohesion. The evaluation is performed applying the CBO and LCOM metrics from the CK suite, which we adapt to measure AO software. We argue that the existing metrics are invalid for evaluating overall coupling in AO systems, since they do not take into account semantic dependencies between the system modules. We also describe some bugs in the tool that is widely used to evaluate AO software. We have fixed and extended this tool to collect our metrics. Finally, we present the results of our experiment. We found that there is no pattern whose AO implementations exhibits lower coupling, while 22 patterns present lower coupling in the OO implementations. With the help of Dependency Structure Matrix we analyze in detail the coupling dependencies between modules of the Observer pattern. With regard to cohesion the OO implementations are superior in 9 cases, while the AO ones in 6 cases. 8 patterns exhibit the same cohesion in both implementations.
- [Przybyłek, 2011b] expands our earlier work on software modularity. We compare two versions (Java and AspectJ) of 10 real-life systems. The

obtained results confirm our previous findings. We found that there is no evidence that AOP promotes better modularity of software than OOP. The OO implementation of every system exhibits lower coupling. With regard to cohesion the OO implementations are superior in 4 cases, while the AO ones in 6 cases. We also further explain semantic dependencies in AO software to give a rationale for our coupling metric.

- [Przybyłek, 2008b] examines whether the existing implementations of the Decorator pattern can be improved using AspectJ and generics. In the first part of the paper we review the solutions to the Decorator pattern presented by Hannemann & Kiczales and Borella. It turns out that their solutions have some limitations and imperfections. In the second part of the paper we develop a reusable implementation of the Decorator pattern that can be easily (un)plugged into code.
- [Przybyłek, 2010b] is a continuation of [Przybyłek, 2008b] and further explores the possibilities for improving implementations of the GoF design patterns. It was found that not only Decorator, but also Proxy can take advantage by using generics. In addition, reflective programming was employed for Prototype to provide a default implementation for cloning. In each case, the obtained implementation is highly reusable.
- [Przybyłek, 2011c] describes a quasi-laboratory experiment in which we compare the evolvability and reusability of OO and AO implementations of a classical producer-consumer system that undergoes five maintenance scenarios. We have found that in general the OO implementations have exhibited superior results through the maintenance tasks. Nevertheless, more industrial data needs to be investigated before more definitive conclusions can be drawn about the impact of AOP on software evolvability and reusability.
- [Przybyłek, 2011a] presents the proposal of our dissertation.

Table 8.1 shows how the publications are mapped into the chapters of the dissertation.

Table 8.1 Mapping from the publications to the chapters

publication	chapters
[Przybyłek, 2007]	2, 3
[Przybyłek, 2008a]	4
[Przybyłek, 2008b]	7.2
[Przybyłek, 2009]	2, 3
[Przybyłek, 2010a]	6
[Przybyłek, 2010b]	7.2
[Przybyłek, 2010c]	3.3
[Przybyłek, 2011a]	1, 8
[Przybyłek, 2011b]	6
[Przybyłek, 2011c]	7.1

8.4 Epilog

In any community of scientists, there are some individuals who are bolder than most. Occasionally they generate a rival to the established paradigm [Kuhn, 1962]. That was in 1997, when Kiczales et al. [1997] presented AOP. As most promising new paradigms, AOP started with hype (see Figure 8.1). Hype is a natural handmaiden to overpromise, and most technologies build rapidly to a peak of hype. Following this, there is almost always an overreaction to ideas that are not fully developed [Bezdek, 1993]. AOP is currently at this stage (have already been some criticisms published [Tourwe et al., 2003; Constantinides et al., 2004; Steimann, 2006]) and to go further it needs more critical reviews by disciples of Dijkstra. The criticism leads to a crash of sorts, followed by a period of wallowing in the depths of cynicism. Many new technologies evolve to this point, and then fade away. The ones that survive do so because someone finds a good use (= true user benefit) for the basic ideas [Bezdek, 1993].

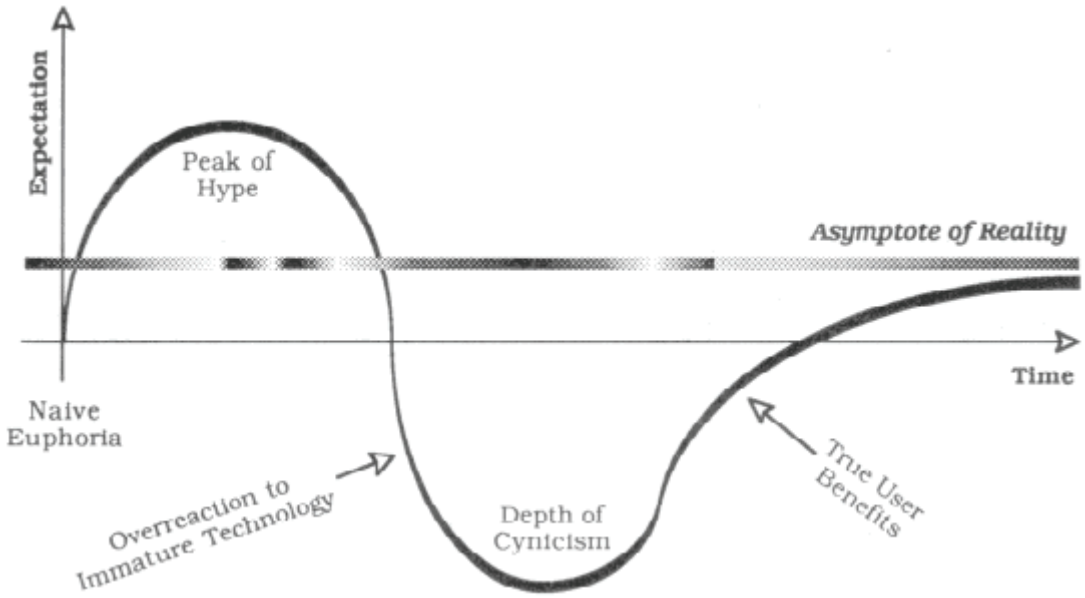


Figure 8.1 Evolution of new technology [Bezdek, 1993]

It is likely that OOP will not be the last programming paradigm developed in the history of software engineering. Something will supersede OOP, just as OOP has superseded procedural programming. Whether AOP will become a subsidiary technique to implement just a few crosscutting concerns like logging, tracing, debugging, etc. or its anomalies will be resolvable and it will become a mainstream paradigm is impossible to predict. One thing is certain, the potential transfer of AOP to the mainstrem of the software development dependents on our ability to discover the AOP's pitfalls. Thus, the dissertation is one more step towards closing AOP to the "asymptote of reality".

References

[Aksit & Tripathi, 1988] Aksit, M., Tripathi, A.: Data Abstraction Mechanisms in Sina. ACM Sigplan Notices, vol. 23(11), pp. 267-275, 1988

[Albin-amiot & Guéhéneuc, 2001] Albin-amiot, H., Guéhéneuc, Y.: Design Patterns: A Round-trip. In: 15th European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary, 2001

[Aldawud et al., 2003] Aldawud, O., Elrad, T., Bader, A.: UML Profile for Aspect-Oriented Software Development. In: 3rd Workshop on Aspect-Oriented Modeling with UML at AOSD'03, Boston, MA, 2003

[Aldrich, 2005] Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05), Glasgow, UK, 2005

[Andrews et al., 2002] Andrews, A., Ghosh, S., Man Choi, E.: A Model for Understanding Software Components. In: IEEE International Conference on Software Maintenance (ICSM'02), Montreal, Canada, 2002

[Anquetil & Laval, 2011] Anquetil, N., Laval, J.: Legacy Software Restructuring: Analyzing a Concrete Case. In: 15th European Conference on Software Maintenance and Reengineering (CSMR'11), Oldenburg, 2011

[Avison et al., 2001] Avison, D.E., Baskerville, R., Myers, M.: Controlling action research projects. Information Technology & People, 14 (1), pp. 28-45, 2001

[Balasubramanian, 1996] Balasubramanian, N.V.: Object oriented metrics. In: 3rd Asia-Pacific Software Engineering Conference (APSEC'96), Seoul, South Korea, 1996

[Baldwin & Clark, 2000] Baldwin, C.Y., Clark, K.B.: Design Rules, vol. 1, The Power of Modularity. MIT Press, Cambridge, 2000

[Bartsch & Harrison, 2008] Bartsch, M., Harrison, R.: An exploratory study of the effect of aspect-oriented programming on maintainability. Software Quality Journal, vol. 16(1), 23-44, 2008

[Bartsch & Harrison, 2007] Bartsch, M., Harrison, R.: Design Patterns with Aspects: A Case Study. In: Writing Group at EuroPLoP'07, Irsee Monastery, Germany, 2007

[Basili et al., 1996] Basili, V.R., Briand, L.C., Melo, W.L.: A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering vol. 22(10), pp. 751-761, 1996

[Basili et al., 1994] Basili, V.R., Caldiera, G., Rombach, H.D.: The Goal Question Metric Approach. In: Encyclopedia of Software Engineering, pp. 528-532, John Wiley & Sons, Inc., New York, 1994

[Basili et al., 2007] Basili, V.R., Heidrich, J., Lindvall, M., Munch, J., Regardie, M., Trendowicz, A.: GMQ + Strategies - Aligning Business Strategies with Software Measurement. In: Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM'07), Madrid, Spain, 2007

[Basili et al., 1999] Basili, V.R., Shull, F., Lanubile, F.: Building Knowledge through Families of Experiments. In: IEEE Transactions on Software Engineering, vol. 25(4), pp. 456-473, July 1999

[Basili & Weiss, 1984] Basili, V.R., Weiss, D.: A Methodology for Collecting Valid Software Engineering Data. In: IEEE Transactions On Software Engineering, pp. 728-738, Nov. 1984

[Basili, 1992] Basili, V.R.: Software Modeling and Measurement: The Goal/Question/Metric Paradigm. Technical Reports, University of Maryland, 1992

[Baskerville & Wood-Harper, 1996] Baskerville, R.L., Wood-Harper, A.T.: A critical perspective on action research as a method for information systems research. Journal of Information Technology, 11 (3), pp. 235-246, 1996

[Becker & Niehaves, 2007] Becker, J., Niehaves, B.: Epistemological perspectives on IS research: a framework for analysing and systematizing epistemological assumptions. Information Systems Journal, vol. 17, pp. 197-214, 2007

[Beltagui, 2003] Beltagui, F.: Features and Aspects: Exploring feature-oriented and aspect-oriented programming interactions. Technical Report No: COMP-003-2003; Computing Department, Lancaster University, Lancaster, 2003

[Bergmans & Aksit, 2001] Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters. Commun. ACM, vol. 44(10), pp. 51-57, 2001

[Bergmans, 1994] Bergmans, L.: Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs. PhD thesis, University of Twente, 1994

[Bernardi & Lucca, 2010] Bernardi, M.L., Lucca, G.A.: A metric model for aspects' coupling. In: Workshop on Emerging Trends in Software Metrics at ICSE'10, Cape Town, South Africa, 2010

[Bezdek, 1993] Bezdek, J.C.: Fuzzy models - what are they, and why. IEEE Transactions on Fuzzy Systems, vol. 1(1), pp. 1-6, 1993

[Bieman & Kang, 1995] Bieman, J. M., Kang, B.: Cohesion and reuse in an object-oriented system. SIGSOFT Softw. Eng. Notes vol. 20, Issue SI, pp. 259-262, 1995

[Boehm, 1981] Boehm, E.: Software Engineering Economics. Prentice-Hall, 1981

[Booch, 1994] Booch, G.: Object-oriented Analysis and Design with Applications. Benjamin-Cummings, Redwood City, California, 1994

[Borella, 2003] Borella, J.: Design Patterns Using Aspect-Oriented Programming. MSc thesis, IT University of Copenhagen, 2003

[Boudreau et al., 2001] Boudreau, M.C., Gefen, D., Straub, D.: Validation in IS Research: A State-of-the-Art Assessment. MIS Quarterly, Vol. 25, No. 1, pp. 1-16, 2001

[Bowen et al., 1983] Bowen, T.P., Post, J.V., Tai, J., Presson, P.E., Schmidt, R.L.: Software Quality Measurement for Distributed Systems. Guidebook for Software Quality Measurement. Technical Report RADC-TR-83-175 Volume 2, July 1983

[Breivold et al., 2008] Breivold, H.P., Crnkovic, I., Land, R., Larsson, S.: Using Dependency Model to Support Software Architecture Evolution. In: 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, 2008

[Briand et al., 1998] Briand, L.C., Daly, J.W., Wüst, J.: A Unified Framework for Cohesion Measurement in Object-Oriented Systems. Empirical Software Engineering vol. 3(1), pp. 65-117, 1998

[Briand et al., 1999] Briand, L.C., Daly, J.W., Wüst, J.: A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering vol. 25(1), pp. 91-121, 1999

[Briand et al., 1999] Briand, L.C., Morasca, S., Basili, V.R.: Defining and Validating Measures for Object-Based High-Level Design. IEEE Transactions on Software Engineering vol. 25(5), pp. 722-743, 1999

[Briand et al., 2001] Briand, L.C., Wüst, J., Lounis, H.: Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs. Empirical Software Engineering, vol. 6(1), pp. 11-58, 2001

[Brichau et al., 2000] Brichau, J., De Meuter, W., De Volder, K.: Jumping Aspects. In: Workshop on Aspects and Dimensions of Concerns at ECOOP'00, Sophia Antipolis and Cannes, France, 2000

[Brito e Abreu et al., 2002] Brito e Abreu, F., Poels, G., Sahraoui, H.A., Zuse, H.: Quantitative Approaches in Object-Oriented Software Engineering. Kogan Page, Paris, 2002

[Brooks, 1995] Brooks, F.P.: The mythical man-month: After 20 years. IEEE Software vol. 12, pp. 57-60, 1995

[Bruntink & Deursen, 2004] Bruntink, M., Deursen, A. van, Engelen, R. van, Tourwe, T.: An Evaluation of Clone Detection Techniques for Identifying Cross-Cutting Concerns. In: International Conference on Software Maintenance (ICSM 2004). IEEE Computer Society, pp. 200-209, 2004

[Burrows et al., 2010] Burrows, R., Ferrari, F., Garcia, A., Taiani, F.: An empirical evaluation of coupling metrics on aspect-oriented programs. In: Workshop on Emerging Trends in Software Metrics at ICSE'10, Cape Town, South Africa, 2010

[Burrows et al., 2010] Burrows, R., Ferrari, F., Lemos, O., Garcia, A., Ta?ani, F.: The Impact of Coupling on the Fault-Proneness of Aspect-Oriented Programs: An Empirical Study. In: 21st IEEE International Symposium on Software Reliability Engineering (ISSRE'10), San Jose, CA, 2010

[Cai, 2006] Cai, Y.: Modularity in Design: Formal Modeling and Automated Analysis. PhD thesis, University of Virginia, Charlottesville, VA, 2006

[Castor et al., 2009] Castor, F., Cacho, N., Figueiredo, E., Garcia, A., Rubira, C.M., de Amorim, J.S., da Silva, H.O.: On the modularization and reuse of exception handling with aspects. Softw. Pract. Exper. vol. 39(17), pp. 1377-1417, 2009

[Ceccato & Tonella, 2004] Ceccato, M., Tonella, P.: Measuring the Effects of Software Aspectization. In: 1st Workshop on Aspect Reverse Engineering, Delft, Netherlands, 2004

[Cecez-Kecmanovic, 2007] Cecez-Kecmanovic, D.: Critical Research in Information Systems: The Question of Methodology. In: 15th European Conference on Information Systems (ECIS'07), St. Gallen, Switzerland, 2007

[Chaumun et al., 2000] Chaumun, M. A., Kabaili, H., Keller, R. K., Lustman, F., Saint-Denis, G.: Design Properties and Object-Oriented Software Changeability. In: 13th Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 2000

[Chidamber & Kemerer, 1994] Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng. 20(6) , pp. 476-493, Jun. 1994

[Clarke & Banaissad, 2005] Clarke, S., Banaissad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Upper Saddle River: Addison-Wesley, 2005

[Clifton & Leavens, 2003] Clifton, C., Leavens, G.T.: Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy. In: Software-engineering Properties of Languages for Aspect Technologies (SPLAT'03), Boston, MA, 2003

[Clifton & Leavens, 2002] Clifton, C., Leavens, G.T.: Spectators and Assistants: Enabling Modular Aspect-Oriented Reasoning. Technical Report 02-10, Iowa State University, 2002

[Clifton, 2005] Clifton, C.: A design discipline and language features for modular reasoning in aspect-oriented programs. Phd thesis, Department of Computer Science, Iowa State University, USA, 2005

[Cline et al., 1998] Cline, M., Lomow, G., Girou, M.: C++ FAQs. Addison Wesley, 1998

[Coad & Yourdon, 1991] Coad, P., Yourdon, E.: Object-Oriented Analysis. Prentice Hall, 1991

[Coady & Kiczales, 2003] Coady, Y., Kiczales, G.: Back to the future: a retroactive study of aspect evolution in operating system code. In: 2nd Inter. Conf. on Aspect-oriented software development (AOSD'03), Boston, Massachusetts, 2003

[Coleman et al., 1994] Coleman, D., Ash, D., Lowther, B., Oman, P.: Using metrics to evaluate software system maintainability. IEEE Computer, vol. 27(8), pp. 44-49, 1994

[Colyer et al., 2004] Colyer, A., Clement, A., Harley, G., Webster, M.: Eclipse AspectJ: Aspect-oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison Wesley Professional, Reading MA, 2004

[Constantinides et al., 2004] Constantinides, C., Scotinides, T., Störzer, M.: AOP considered harmful. In: 1st European Interactive Workshop on Aspect Systems (EIWAS), 2004

[Czarnecki & Eisenecker, 2000] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Techniques, and Applications, Addison-Wesley, Boston, MA, 2000

[Dahl et al., 1972] Dahl, O.J., Dijkstra, E.W., Hoare, C.A.: Structured Programming. Academic Press Ltd., 1972

[Dantas & Walker, 2006] Dantas, D.S., Walker, D.: Harmless advice. In Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp.383-396, New York, 2006

[Davison et al., 2004] Davison, R.M., Martinsons, M.G., Kock, N.: Principles of Canonical Action Research. Information Systems Journal 14(1), pp. 65-86, 2004

[De Win et al., 2002] De Win, B., Piessens, F., Joosen, W., Verhanneman, T.: On the importance of the separation-of-concerns principle in secure software engineering. In: ACSA Workshop on the Application of Engineering Principles to System Security Design, Boston, Massachusetts, 2002

[Denier et al., 2005] Denier, S., Albin-Amiot, H., Cointe, P.: Expression and Composition of Design Patterns with Aspects. In: 2nd French Workshop on Aspect-Oriented Software Development (JFDLPA'05), Lille , France, 2005

[Dijkstra, 1976] Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs, 1976

[Dijkstra, 1968] Dijkstra, E.W.: GoTo statement considered harmful. Communications of the ACM, vol 11(3), pp. 147-148, 1968

[Dijkstra, 1974] Dijkstra, E.W.: On the role of scientific thought. Netherlands, 1974,
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

[Durr, 2008] Durr, P.E.A.: Resource-based Verification for Robust Composition of Aspects. PhD thesis, University of Twente, 2008

[Eaddy et al., 2008] Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V.: Do Crosscutting Concerns Cause Defects? In: IEEE Transactions on Software Engineering, vol. 34, pp. 497-515, 2008

[Easterbrook et al., 2007] Easterbrook, S.M., Singer, J., Storey, M.A., Damian, D.: Selecting Empirical Methods for Software Engineering Research. In F. Shull, J. Singer and D. Sj?berg(eds) Guide to Advanced Empirical Software Engineering, Springer, 2007

[Eick et al., 2001] Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S.,; Mockus, A.: Does Code Decay? Assessing the Evidence from Change Management Data. In: IEEE Trans. Softw. Eng., vol. 27(1), pp. 1-12, January 2001

[Evermann, 2007] Evermann, J.: A meta-level specification and profile for AspectJ in UML. Journal of Object Technology, vol. 6(7), Special Issue: Aspect-Oriented Modeling, pp. 27-49, 2007

[Fenton & Melton, 1990] Fenton, N.E., Melton, A.: Deriving Structurally Based Software Measures. J. Syst. Software, vol. 12, pp. 177-187, 1990

[Fenton & Pfleeger, 1997] Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. Publishing Company, Boston, 1997

[Ferrari et al., 2010] Ferrari, F., Burrows, R., Lemos, O., Garcia, A., Figueiredo, E., Cacho, N., Lopes, F., Temudo, N., Silva, L., Soares, S., Rashid, A., Masiero, P., Batista, T., Maldonado, J.: An exploratory study of fault-proneness in evolving aspect-oriented programs. In: 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10), Cape Town, South Africa, 2010

[Figueiredo et al., 2008] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., Dantas, F.: Evolving software product lines with aspects: An empirical study on design stability. In: 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, 2008

[Filho et al., 2006] Filho, F.C., Cacho, N., Figueiredo, E., Maranh?o, R., Garcia, A., Rubira, C.M.: Exceptions and aspects: the devil is in the details. In: Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering, Portland, Oregon, 2006

[Filman & Friedman, 2000] Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns at OOPSLA'00, Minneapolis, MN, 2000

[Filman, 2001] Filman, R.E.: What is Aspect-Oriented Programming, revisited. In: Workshop on Multi-Dimensional Separation of Concerns at ECOOP'01, Budapest, Hungary, 2001

[Fjeldstad & Hamlen, 1983] Fjeldstad, R., Hamlen, W.: Application program maintenance-report to to our respondents. In: Tutorial on Software Maintenance, pp. 13-27. Parikh, G. & Zvegintzov, N. (Eds.). IEEE Computer Soc. Press, 1983

[Frakes, 1993] Frakes, W.: Software Reuse as Industrial Experiment. In: American Programmer, vol. 6(9), pp. 27-33, 1993

[France et al., 2003] France, R., Georg, G., Ray, I.: Supporting Multi-Dimensional Separation of Design Concerns. In: 3rd Workshop on Aspect-Oriented Modeling with UML at AOSD'03, Boston, MA, 2003

[Fuentes & Sanchez, 2007] Fuentes, L., Sanchez, P.: Towards Executable Aspect-Oriented UML Models. In: 10th International Workshop on Aspect-Oriented Modeling at AOSD'07, Vancouver, Canada, 2007

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Boston, MA, 1995

[Gao et al., 2004] Gao, S., Deng, Y., Yu, H., He, X., Beznosov, K., Cooper, K.: Applying Aspect-Orientation in Designing Security Systems: a Case Study. In: 16th International Conference on Software Engineering (SEKE'04), Banff, Canada, 2004

[Garcia et al., 2005] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: Modularizing design patterns with aspects: a quantitative study. In: Proceedings of the 4th international Conference on Aspect-Oriented Software Development (AOSD'05), Chicago, Illinois, 2005

[Gauthier & .Pont, 1970] Gauthier, R., .Pont, S.: Designing Systems Programs. Prentice-Hall, Englewood Cliffs, N.J., 1970

[Glass, 2002] Glass, R.L.: Facts and Fallacies of Software Engineering. Addison Wesley, 2002

[Godil & Jacobsen, 2005] Godil, I., Jacobsen, H.: Horizontal decomposition of Prevayler. In: The 2005 Conference of the Centre For Advanced Studies on Collaborative Research, Toronto, Canada, 2005

[Gradecki & Lesiecki, 2003] Gradecki, J.D., Lesiecki, N.: Mastering AspectJ: Aspect-Oriented Programming in Java. Wiley, Canada, 2003

[Grand, 2002] Grand, M.: Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML. John Wiley & Sons, 2002

[Gray, 2002] Gray, J.: Aspect-Oriented Domain-Specific Modeling: A Generative Approach Using a Meta-weaver Framework. PhD thesis. Vanderbilt University, Nashville, TN, 2002

[Greenwood et al., 2007] Greenwood, P., Bartolomei, T.T., Figueiredo, E., Dósea, M., Garcia, A.F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: 21st European Conference on Object-Oriented Programming (ECOOP'07), Berlin, Germany, 2007

[Griswold et al., 2006] Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan,H.: Modular Software Design with Crosscutting Interfaces. IEEE Software, vol. 23(1), pp. 51-60, 2006

[Groher & Baumgarth, 2004] Groher, I., Baumgarth, T.: Aspect-Orientation from Design to Code. In: Workshop on Early Aspects at AOSD'04, Lancaster, UK, 2004

[Groher & Schulze, 2003] Groher, I., Schulze, S.: Generating Aspect Code from UML Models. In: 3rd Workshop on Aspect-Oriented Modeling with UML at AOSD'03, Boston, MA, 2003

[Gudmundson & Kiczales, 2001] Gudmundson, S., Kiczales, G.: Addressing practical software development issues in AspectJ with a pointcut interface. In: Advanced Separation of Concerns, 2001

[Guyomarc'h & Guéhéneuc, 2005] Guyomarc'h, J., Guéhéneuc, Y.: On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics. In: Workshop on Quantitative Approaches in Object-Oriented Software Engineering at ECOOP'05, Glasgow, UK, 2005

[Hachani & Bardou, 2002] Hachani, O., Bardou, D.: Using Aspect-Oriented Programming for Design Patterns Implementation. In: Workshop Reuse in Object-Oriented Information Systems Design, Montpellier, France, 2002

[Hachani, 2003] Hachani, O.: Aspect/UML: extending UML metamodel for Aspect. Research report, France, 2003

[Hachani, 2003] Hachani, O.: AspectJ/UML: extending UML metamodel for AspectJ. Research report, France, 2003

[Hancock & Algozzine, 2006] Hancock, D.R., Algozzine, R.: Doing Case Study Research: A Practical Guide for Beginning Researchers. Teachers College Press, New York, 2006

[Hanenberg & Unland, 2001] Hanenberg, S., Unland, R.: Using and Reusing Aspects in AspectJ. In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA'01, Tampa Bay, Florida, 2001

[Hannemann & Kiczales, 2002] Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: 17th Conference on Object-Oriented Programming Systems, Languages, and Applications, Seattle, 2002

[Hatton, 1998] Hatton, L.: Does OO sync with how we think? IEEE Software, 15(3), pp. 46-54, May/Jun 1998

[Havinga, 2009] Havinga, W.: On the design of software composition mechanisms and the analysis of composition conflicts. PhD thesis, University of Twente, 2009

[Hevner et al., 2004] Hevner, A. R., March, S. T., Park, J., Ram, S.: Design Science in Information Systems Research. MIS Quarterly, vol. 28(1), pp. 75-105, 2004

[Hitz & Montazeri, 1995] Hitz, M., Montazeri, B.: Measuring Coupling and Cohesion in Object-Oriented Systems. In: 3rd International Symposium on Applied Corporate Computing, Monterrey, Mexico, 1995

[Hoffman & Eugster, 2007] Hoffman, K., Eugster, P.: Bridging Java and AspectJ through explicit join points. In: 5th international Symposium on Principles and Practice of Programming in Java (PPPJ'07), Lisboa, Portugal, 2007

[Hohenstein & Jäger, 2009] Hohenstein, U.D., Jäger, M.C.: Using aspect-orientation in industrial projects: appreciated or damned?. In: Proceedings of the 8th ACM international Conference on Aspect-Oriented Software Development (AOSD'09), Charlottesville, Virginia, 2009

[Hopkins & Horan, 1995] Hopkins, T., Horan, B.: Smalltalk: An Introduction to Application Development Using VisualWorks. Prentice Hall, 1995

[Hovsepyan et al., 2010] Hovsepyan, A., Scandariato, R., Van Baelen, S., Berbers, Y., Joosen, W.: From aspect-oriented models to aspect-oriented code?: the maintenance perspective. In: 9th international Conference on Aspect-Oriented Software Development (AOSD'10), Rennes and Saint-Malo, France, 2010

[Hunt, 1997] Hunt, J.: Smalltalk and Object Orientation. Springer, 1997

[IEEE Std 610.12-1990, 1990] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, 1990

[IEEE Std. 1061, 1998] IEEE Std. 1061, Standard for a Software Quality Metrics Methodology, revision. Piscataway, NJ, 1998

[ISO/IEC 14764-1999, 1999] ISO/IEC 14764-1999, Software Engineering-Software Maintenance, 1999

[Jacobson & Ng, 2005] Jacobson, I., Ng, P.: Aspect-Oriented Software Development with Use Cases. Upper Saddle River: Addison-Wesley, 2005

[Jalote, 2005] Jalote, P.: An Integrated Approach to Software Engineering. Springer, New York, 2005

[Kan, 2002] Kan, S.H.: Metrics and models in software quality engineering. Addison Wesley, Boston, 2002

[Kande et al., 2002] Kande, M.M., Kienzle, J., Strohmeier, A.: From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach. Technical Report, Swiss Federal Institute of Technology Lausanne, Switzerland, 2002

[Kande, 2003] Kande, M.M.: A Concern-Oriented Approach to Software Architecture. PhD thesis. Swiss Federal Institute of Technology, Lausanne, Switzerland, 2003

[Kästner et al., 2007] Kästner, C., Apel, S., Batory, D. A Case Study Implementing Features using AspectJ. In: 11th International Conference of Software Product Line Conference (SPLC'07), Kyoto, Japan, 2007

[Katz, 2004] Katz, S.: Diagnosis of harmful aspects using regression verification. In: Workshop on Foundations of Aspect-Oriented Languages at AOSD'04, Lancaster, UK, 2004

[Kellens et al., 2006] Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In: 20th European Conference on Object-Oriented Programming (ECOOP'06), Nantes, France, 2006

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G.: An Overview of AspectJ. In: 15th European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary, 2001

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Cristina Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: LNCS, vol. 1241, pp. 220-242. Springer, Heidelberg, New York, 1997

[Kniesel et al., 2004] Kniesel, G., Rho, T., Hanenberg, S.: Evolvable Pattern Implementations Need Generic Aspects. In: Workshop on Reflection, AOP and Meta-Data for Software Evolution at ECOOP'04, Oslo, Norway, 2004

[Koppen & Störzer , 2004] Koppen, C., Störzer ,M.: PCDiff: Attacking the fragile pointcut problem. In: European Interactive Workshop on Aspects in Software, Berlin, Germany, 2004

[Kouskouras et al., 2008] Kouskouras, K.G., Chatzigeorgiou, A., Stephanides, G.: Facilitating software extension with design patterns and Aspect-Oriented Programming. In: J. Syst. Softw. vol. 81(10), pp. 1725-1737, October 2008

[Kuhlemann, 2007] Kuhlemann, M.: Design Patterns Revisited. School of Computer Science University of Magdeburg, 2007

[Kuhn, 1962] Kuhn, T.S.: The Structure of Scientific Revolutions. University of Chicago Press, 1962

[Kulesza et al., 2006] Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., von Staa, A., Lucena, C.: Quantifying the effects of aspect-oriented programming: A maintenance study. In: 22nd IEEE International Conference on Software Maintenance (ICSM '06), Dublin, Ireland, 2006

[Laddad, 2003] Laddad, R.: AspectJ in Action. Manning, 2003

[Lagaisse et al., 2004] Lagaisse, B., Joosen, W., De Win, B.: Managing semantic interference with aspect integration contracts. In: 5th workshop on Software engineering properties of languages and aspect technologies (SPLAT'07) at AOSD'07, Vancouver, Canada, 2004

[Larkin & Wilson, 1993] Larkin, D., Wilson, G.: Object-Oriented Programming and the Objective-C Language. Addison Wesley, 1993

[Leavens & Clifton, 2007] Leavens, G.T., Clifton, C.: Multiple concerns in aspect-oriented language design: a language engineering approach to balancing benefits, with examples. In: 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT'07), Vancouver, 2007

[Lemos et al., 2006] Lemos, O.A., Junqueira, D.C., Silva, M.A., Fortes, R.P., Stamey, J.: Using aspect-oriented PHP to implement crosscutting concerns in a collaborative web system. In: 24th Annual ACM International Conference on Design of Communication, Myrtle Beach, South Carolina, 2006

[Lesiecki, 2002] Lesiecki, N.: Improve modularity with aspect-oriented programming. http://www.ibm.com/developerworks/library/j-aspectj/, 2002

[Lewis, 2004] Lewis, W.E.: Software Testing and Continuous Quality Improvement. Auerbach, 2004

[Li & Henry, 1995] Li, W., Henry, S.: An empirical study of maintenance activities in two object-oriented systems. Journal of Software Maintenance vol. 7(2), pp. 131-147, 1995

[Lientz et al., 1978] Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of application software maintenance. In: Commun. ACM, vol. 12, pp. 466-471, June 1978

[Lions et al., 2002] Lions, J.M., Simoneau, D., Pilette, G., Moussa, I.: Extending OpenTool/UML Using Metamodeling: an AOP Case Study. In: 2nd Workshop on AOM with UML at UML'02, Dresden, Germany, 2002

[MacCormack et al., 2007] MacCormack, A., Rusnak, J., Baldwin, C.: The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. Harvard Business School Technology & Operations Mgt. Unit Research Paper, vol. No. 08-038, 2007

[Madeyski & Szała, 2007] Madeyski, L., Szała, Ł.: Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. IET Software Journal, vol. 1(5), pp. 180-187, 2007

[Mancoridis et al., 1998] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., Gansner, E.R.: Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In: 6th international Workshop on Program Comprehension (IWPC'98), Ischia, Italy, 1998

[March & Smith, 1995] March, S. T., Smith, G.: Design and Natural Science Research on Information Technology. In: Decision Support Systems, no. 15(4), pp. 251-266, 1995

[Marin et al., 2007] Marin, M., Moonen, L., van Deursen, A.: An Integrated Crosscutting Concern Migration Strategy and its Application to JHotDraw. In: IEEE International Conference on Source Code Analysis and Manipulation (SCAM'07), Paris, France, 2007

[Marot & Wuyts, 2010] Marot, A., Wuyts, R.: Composing aspects with aspects. In: 9th International Conference on Aspect-Oriented Software Development (AOSD'10), Rennes, France, 2010

[Mauch & Birch, 2003] Mauch, J.E., Birch, J.W.: Guide to the successful thesis and dissertation (5th Edition). Marcel Dekker, Inc., New York, 2003

[McAulay et al., 2002] McAulay, L., Doherty, N., Keval, N.: The Stakeholder Dimension in Information Systems Evaluation. Journal of Information Technology , vol. 17, pp. 241-255, 2002

[McGrath, 2005] McGrath, K.: Doing critical research in information systems: a case of theory and practice not informing each other. Information Systems Journal 15(2), pp. 85-101, 2005

[McIlroy, 1968] McIlroy, M.D.: Mass produced software components. In: The NATO Software Engineering Conferences, pp. 138-155, Garmisch, Germany, 1968

[McKee, 1984] McKee, J.: Maintenance as a function of design. In: National Computer Conference and Exposition (AFIPS'84), Las Vegas, Nevada, 1984

[Mens et al., 2004] Mens, T., Mens, K., Tourwé, T.: Software Evolution and Aspect-Oriented Software Development, a cross-fertilisation. ERCIM special issue on Automated Software Engineering, Vienna, Austria, 2004

[Meyer, 1989] Meyer, B.: Object-oriented Software Construction, Prentice Hall, 1989

[Meyers, 1988] Meyers, W.: Interview with Wilma Osborne: developers must design in maintainability. IEEE Software vol. 5(3), pp. 104-105, 1988

[Mezini & Ostermann, 2004] Mezini, M., Ostermann, K.: Untangling Crosscutting Models with Caesar. In: Filman, E.E., Elrad, T., Clarke, S., Aksit, M. (Ed.): Aspect-Oriented Software Development. Addison Wesley, Canada, 2004

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

[Milton, 1985] Milton, J.A.: Research Methodologies and MIS Research. In: Research Methods in Information Systems, E. Mumford et al. (Ed.), Elsevier Science Publishers B.V., Amsterdam, Holland, pp. 103-117, 1985

[Monteiro & Fernandes, 2004] Monteiro, M.P., Fernandes, J.M.: Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns. In: Desarrollo de Software Orientado a Aspectos (DSOA'04), Málaga, Spain, 2004

[Mortensen et al., 2010] Mortensen, M., Ghosh, S., Bieman, J.: Aspect-Oriented Refactoring of Legacy Applications: An Evaluation. In: IEEE Trans. Software Engineering vol. 99, 2010

[Mortensen, 2009] Mortensen, M.: Improving Software Maintainability through Aspectualization. PhD thesis, Department of Computer Science, Colorado State University, CO, 2009

[Mosconi et al., 2008] Mosconi, M., Charfi, A., Svacina, J.: Applying and Evaluating AOM for Platform Independent Behavioral UML Models. In: 7th International Conference on Aspect-Oriented Software Development (AOSD'08), Brussels, Belgium, 2008

[Munoz et al., 2007] Munoz, F., Barais, O., Baudry, B.: Vigilant usage of aspects. In: Workshop on Aspects, Dependencies and Interactions at ECOOP'07, Berlin, Germany, 2007

[Munoz et al., 2008] Munoz, F., Baudry, B., Barais, O.: Improving maintenance in AOP through an interaction specification framework. In: 24th IEEE International Conference on Software Maintenance, Beijing, China, 2008

[Munoz et al., 2008] Munoz, F., Baudry, B., Delamare, R., Le Traon, Y.: Inquiring the usage of aspect-oriented programming: an empirical study. In: 25th International conference on Software Maintentance (ICSM'09), Alberta, Canada, 2008

[Murphy & Schwanninger, 2006] Murphy, G., Schwanninger, Ch.: Guest Editors' Introduction: Aspect-Oriented Programming. In: IEEE Software, vol. 23(1), pp. 20-23, Jan./Feb. 2006

[Niehaves & Stahl, 2006] Niehaves, B., Stahl, B. C.: Criticality, Epistemology, and Behaviour vs. Design - IS Research across different sets of paradigms. In: 14th European Conference on Information Systems (ECIS'06), Göteborg, 2006

[Noda & Kishi, 2001] Noda, N., Kishi, T.: Implementing Design Patterns Using Advanced Separation of Concerns. In: Workshop on Advanced SoC in OO Systems at OOPSLA'01, Tampa Bay, Florida, 2001

[Object Management Group, 2009] Object Management Group, UML, Infrastructure, V2.2. Document Number: formal/2009-02-04, http://www.omg.org/spec/UML, 2009

[Object Management Group, 2009] Object Management Group, UML, Superstructure, V2.2. Document Number: formal/2009-02-02, http://www.omg.org/spec/UML, 2009

[Ongkingco et al., 2006] Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O., Sittampalam, G.: Adding open modules to AspectJ. In: 5th International Conference on Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, 2006

[Oprisan, 2008] Oprisan, A.: Aspect Oriented Implementation of Design Patterns using Metadata. MSc thesis, University of Joensuu, 2008

[Orlikowski & Iacono, 2001] Orlikowski, W., Iacono, C.: Desperately Seeking the "IT" in IT Research - A Call to Theorizing the IT Artifact. Information Systems Research, no. 12(2), pp. 121-134, 2001

[Ossher, 1987] Ossher, H.: A Mechanism for Specifying the Structure of Large Layered Systems. In: Shriver, B., Wegner, P. (eds.): Research Directions in Object-Oriented Programming. MIT Press, Cambridge, MA, pp. 219-252, 1987

[Ostermann, 2003] Modules for Hierarchical and Crosscutting Models. PhD thesis, Technische Universität Darmstadt, 2003

[Ostermann et al., 2011] Ostermann, K., Giarrusso, P.G., Kastner, Ch., Rendel, T.: Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity. In: 25th European Conference on Object-Oriented Programming (ECOOP'11), Lancaster, UK, 2011

[Parnas et al., 1984] Parnas, D.L., Clements, P.C., Weiss, D.M.: The modular structure of complex systems. In Proceedings of the 7th International Conference on Software Engineering, Orlando, Florida, 1984

[Parnas, 1972] Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM, vol. 15(12), pp. 1053-1058. ACM Press, New York, 1972

[Perepletchikov et al., 2007] Perepletchikov, M., Ryan, C., Frampton, K.: Cohesion Metrics for Predicting Maintainability of Service-Oriented Software. In: 7th International Conference on Quality Software (QSIC'07), Portland, Oregon, 2007

[Peters & Pedrycz, 2000] Peters, J.F., Pedrycz, W.: Software Engineering: An Engineerng Approach. John Wiley & Sons, Inc., 2000

[Pigoski, 1997] Pigoski, T.M.: Practical Software Maintenance. Wiley Computer Publishing, 1997

[Piveta & Zancanella, 2003] Piveta, E.K., Zancanella, L.C.: Observer Pattern using Aspect-Oriented Programming. In: 3rd Latin American Conference on Pattern Languages of Programming, Porto de Galinhas, Brazil, 2003

[Pohl et al., 2008] Pohl, Ch., Charfi, A., Gilani, W., Göbel, S., Grammel, B., Lochmann, H., Rummler, A., Spriestersbach, A.: Adopting Aspect-Oriented Software Development in Business Application Engineering. In: 7th International Conference on Aspect-Oriented Software Development (AOSD'08), Brussels, Belgium, 2008

[Ponnambalam, 1997] Ponnambalam, K.: Characterization and Selection of Good Object-Oriented Design. In: Workshop on OO Design at OOPSLA'97, Atlanta, Georgia, 1997

[Pressman, 2005] Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill, New York, 2005

[Przybyłek, 2007] Przybyłek, A.: Post object-oriented paradigms in software development: a comparative analysis. In: International Multiconference on Computer Science and Information Technology (IMCSIT'07), Wisła, Poland, 2007

[Przybyłek, 2008a] Przybyłek, A.: Separation of crosscutting concerns at the design level: An extension to the UML metamodel. In: International Multiconference on Computer Science and Information Technology (IMCSIT'08), Wisła, Poland, 2008

[Przybyłek, 2008b] Przybyłek, A.: The Decorator pattern revisited: an aspect-oriented solution. In: 7th International Conference on Perspectives in Business Informatics Research (BIR'08), Sopot, Poland, 2008

[Przybyłek, 2009] Przybyłek, A.: Beyond object-oriented software development. In: (eds.) Hussain, M.A.: Advances in Computer Science and IT, In-Tech, 2009

[Przybyłek, 2010a] Przybyłek, A.: An empirical assessment of the impact of AOP on software modularity. In: 5th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'10), Athens, Greece, 2010

[Przybyłek, 2010b] Przybyłek, A.: Design Patterns with AspectJ, generics, and reflective programming. In: 5th International Conference on Software and Data Technologies (ICSOFT'10), Athens, Greece, 2010

[Przybyłek, 2010c] Przybyłek, A.: What is wrong with AOP? In: 5th International Conference on Software and Data Technologies (ICSOFT'10), Athens, Greece, 2010

[Przybyłek, 2011a] Przybyłek, A.: Impact of aspect-oriented programming on software modularity. In: Doctoral Symposium at 15th European Conference on Software Maintenance and Reengineering (CSMR'11), Oldenburg, 2011

[Przybyłek, 2011b] Przybyłek, A.: Where the truth lies: AOP and its impact on software modularity. In: Giannakopoulou, D., Orejas, F. (eds.) ETAPS 2011. LNCS, vol. 6603, pp. 447-461. Springer, Heidelberg, 2011

[Przybyłek, 2011c] Przybyłek, A.: Systems Evolution and Software Reuse in Object-Oriented Programming and Aspect-Oriented Programming. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS 6705, pp. 163-178. Springer, Heidelberg, 2011

[Rashid et al., 2010] Rashid, A., Cottenier, T., Greenwood, P., Chitchyan, R., Meunier, R., Coelho, R., Sudholt, M., Joosen, W.: Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe. IEEE Computer vol. 43(2), pp. 19-26, Feb. 2010

[Rashid & Moreira, 2006] Rashid, A., Moreira, A.: Domain Models are NOT Aspect Free. In: 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06), Genova, Italy, 2006

[Raymond, 2003] Raymond, E.S.: The Art of Unix Programming. Addison Wesley, 2003

[Recebli, 2005] Recebli, E.: Pure aspects. MSc thesis, Oxford University, 2005

[Reina et al., 2004] Reina, A. M., Torres, J., Toro, M.: Towards Developing Generic Solutions with Aspects. In: 5th Aspect-Oriented Modeling Workshop at UML'04, Lisbon, Portugal, 2004

[Rentrop, 2006] Rentrop, J.: Software Metrics as Benchmarks for Source Code Quality of Software Systems. MSc thesis, Universiteit van Amsterdam, 2006

[Ribeiro et al., 2007] Ribeiro, M., Dósea, M., Bonifácio, R., Neto, A.C., Borba, P., Soares, S.: Analyzing Class and Crosscutting Modularity with Design Structure Matrixes. In: Proceedings of the 21th Brazilian Symposium on Software Engineering (SBES'07), Joao Pessoa, Brazil, 2007

[Riel, 1996] Riel, A.J.: Object-oriented Design Heuristics, Addison-Wesley, Boston, 1996

[Robillard & Weigand-Warr, 2005] Robillard, M.P., Weigand-Warr, F.: ConcernMapper: simple view-based separation of scattered concerns. In: Workshop on Eclipse technology eXchange at OOPSLA'05, San Diego, CA, 2005

[Ryder & Tip, 2001] Ryder, B.G., Tip, F.: Change impact analysis for object-oriented programs. In: 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, Utah, 2001

[Sant'Anna et al., 2003] Sant'Anna, C., Garcia, A., Chavez, C. Lucena, C., von Staa, A.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: 17th Brazilian Symposium on Software Engineering (SEES'03), Manaus, Brazil, 2003

[Sant'Anna, 2008] Sant'Anna, C.N.: On the Modularity of Aspect-Oriented Design: A Concern-Driven Measurement Approach. Phd thesis, Pontifical Catholic University of Rio de Janeiro, Brazil, 2008

[Sapir et al., 2002] Sapir, N., Tyszberowicz, S., Yehudai, A.: Extending UML with Aspect Usage Constraints in the Analysis and Design Phases. In: 2nd Workshop on Aspect-Oriented Modeling with UML at UML'02, Dresden, Germany, 2002

[Schach, 2007] Schach, S.R.: Object-Oriented and Classical Software Engineering, McGraw-Hill, Singapore, 2007

[Schauerhuber, 2007] Schauerhuber, A. et.al.: A Survey on Web Modeling Approaches for Ubiquitous Web Applications. Technical Report, Vienna University of Technology, 2007

[Schön, 1983] Schön, D. A.: The Reflective Practitioner: How Professionals Think in Action. New York: Basic Books, 1983

[Seaman, 1999] Seaman, C.B.: Qualitative Methods in Empirical Studies of Software Engineering. IEEE Trans. Softw. Eng. 25(4), pp. 557-572, 1999

[Shen & Zhao, 2007] Shen, H., Zhao, J.: An evaluation of coupling metrics for Aspect-Oriented software. In: Technical Report SJTU-CSE-TR-07-04, Center for Software Engineering, SJTU, Shanghai, China, 2007

[Simon, 1996] Simon, H.A.: The Sciences of the Artificial (3rd ed.). Cambridge: MIT Press, 1996

[Soares et al., 2002] Soares, S., Laureano, E., Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. In: 17th ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), Seattle, Washington, 2002

[Sommerville, 2010] Sommerville, I.: Software engineering (9th Edition). Addison Wesley, 2010

[Staijen, 2010] Staijen, T.: Graph-Based Specification and Verification for Aspect-Oriented Languages. PhD thesis, Centre for Telematics and Information Technology, Netherlands, 2010

[Standish, 1984] Standish, T.: An essay on software reuse. In: IEEE Transactions on Software Engineering vol. 10(5), pp. 494-497, 1984

[Steimann, 2005] Steimann, F.: Domain Models Are Aspect Free. In: 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05), Jamaica, 2005

[Steimann, 2006] Steimann, F.: The paradoxical success of aspect-oriented programming. SIGPLAN Not. 41(10), pp. 481-497, Oct. 2006

[Stein et al., 2002] Stein, D., Hanenberg, S., Unland, R.: An UML-based Aspect-Oriented Design Notation. In: Proceedings of the AOM with UML Workshop at AOSD'02, Enschede, Netherlands, 2002

[Stein et al., 2002] Stein, D., Hanenberg, S., Unland, R.: Designing Aspect-Oriented Crosscutting in UML. In: Proceedings of the AOM with UML Workshop at AOSD'02, Enschede, Netherlands, 2002

[Stevens et al., 1974] Stevens, W., G., Myers, L. Constantine: Structured Design. In: IBM Systems Journal, 13(2), pp. 115-139, 1974

[Stochmiałek, 2006] Stochmiałek, M.: AOPmetrics. http://aopmetrics.tigris.org, 2006

[Storey et al., 1999] Storey, M.D., Fracchia, F.D., Müller, H.A.: Cognitive design elements to support the construction of a mental model during software exploration. J. Syst. Softw. vol. 44(3), pp. 171-185, 1999

[Störzer et al., 2006] Störzer, M., Eibauer, U., Schöffmann, S.: Aspect Mining for Aspect Refactoring: An Experience Report. In: Workshop on Towards Evaluation of Aspect Mining at ECOOP'06, Nantes, France, 2006

[Störzer, 2007] Störzer, M.: Impact Analysis for AspectJ - A Critical Analysis and Tool-based Approach to AOP. PhD thesis, School of Computer Science and Mathematics, University of Passau, Germany, 2007

[Sullivan et al., 2005] Sullivan, K., Griswold, W., Song, Y., Chai, Y., Shonle, M., Tewari, N., Rajan, H.: On the criteria to be used in decomposing systems into aspects. In: Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference, Lisbon, Portugal, 2005

[Sullivan et al., 2005] Sullivan, K., Griswold, W., Song, Y., Chai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering, Lisbon, Portugal, 2005

[Tarr et al., 1999] Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: multi-dimensional separation of concerns. In: 21st International Conference on Software Engineering (ICSE'09), Los Angeles, California, 1999

[Taveira et al., 2010] Taveira, J.C., Oliveira, H., Castor, F., Soares, S.: On Inter-Application Reuse of Exception Handling Aspects. In: Workshop on Empirical Evaluation of Software Composition Techniques at AOSD'10, Rennes, France, 2010

[Taveira, 2009] Taveira, J.C. et al.: Assessing Intra-Application Exception Handling Reuse with Aspects. In: 23rd Brazilian Symposium on Software Engineering (SBES'09), Fortaleza, Brazil, 2009

[Tourwé et al., 2003] Tourwé, T., Brichau, J., Gybels, K.: On the Existence of the AOSD-Evolution Paradox. In: Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT) at AOSD'03, Boston, Massachusetts, 2003

[Trindade Leite & Marks, 2005] Trindade Leite, F.C., Marks, A.: Case Study Research in Agricultural and Extension Education: Strengthening the Methodology. In: Journal of International Agricultural and Extension Education, vol. 12(1), 2005

[Tsang et al., 2000] Tsang, S.L., Clarke, S., Baniassad, E.L.: Object Metrics for Aspect Systems: Limiting Empirical Inference Based on Modularity. Technical report, Trinity College, Dublin, 2000

[Tsang et al., 2004] Tsang, S.L., Clarke, S., Baniassad, E.L.: An evaluation of aspect-oriented programming for java-based real-time systems development. In: 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'04), Vienna, Austria, 2004

[Walker et al., 2003] Walker, D., Zdancewic, s., Ligatti, J.: A Theory of Aspects. In: 8th ACM SIGPLAN International Conference on Functional Programming, Uppsala, Sweden, 2003

[Walsham, 2005] Walsham, G.: Learning about Being Critical. Information Systems Journal 15(2), pp. 111-117, 2005

[Wampler, 2007] Wampler, D.: Noninvasiveness and Aspect-Oriented Design: Lessons from Object-Oriented Design Principles. In: 6th International Conference on Aspect-Oriented Software Development (AOSD'07), Vancouver, British Columbia, 2007

[Wirth, 1971] Wirth, N.: Program Development by Stepwise Refinement. Commun. ACM 14(4), pp. 221-227, 1971

[Wirth, 1974] Wirth, N.: On the Composition of Well-Structured Programs. ACM Comput. Surv. 6(4), pp. 247-259, 1974

[Wohlin et al., 2000] Wohlin, C., Runeson, P., Höst, M.: Experimentation in software engineering: an introduction. Kluwer Academic, 2000

[Wulf & Shaw, 1973] Wulf, W., Shaw, M.: Global variable considered harmful. SIGPLAN Notices 8:2, pp. 28-34, 1973

[Yan et al., 2004] Yan, H., Kniesel, G., Cremers, A.: A Meta Model and Modeling Notation for AspectJ. In: 5th Workshop on Aspect-Oriented Modeling at UML'04, Lisbon, Portugal, 2004

[Yin, 2003] Yin, R.K.: Case Study Research: Design and Methods. Sage Publications, Inc., California, 2003

[Yourdon & Constantine, 1979] Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall, New York, 1979

[Yourdon, 1992] Yourdon, E.: Decline and Fall of the American Programmer. Prentice Hall, 1992

[Zakaria et al., 2002] Zakaria, A. A., Hosny, H., Zeid, A.: A UML Extension for Modeling Aspect-Oriented Systems. In: 2nd Workshop on Aspect-Oriented Modeling with UML at UML'02, Dresden, Germany, 2002

[Zelkowitz, 1978] Zelkowitz, M.V.: Perspectives on software engineering. ACM Comput. Surveys, vol. 10, pp. 197-216, June 1978

[Zhang & Jacobsen, 2004] Zhang, C., Jacobsen, H.: Resolving Feature Convolution in Middleware Systems. In: 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 188-205, Vancouver, Canada, 2004

[Zhang et al., 2008] Zhang, S., Gu, Z., Lin, Y., Zhao, J.: Change impact analysis for AspectJ programs. In: 24th IEEE International Conference on Software Maintenance, Beijing, China, 2008

[Zhao, 2002] Zhao, J.: Towards a Metrics Suite for Aspect-Oriented Software. In: Technical Report SE-136-25, Information Processing Society of Japan, 2002

[Zhao, 200] Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. In: 10th International Software Metrics Symposium, Chicago, IL, 2004

Analiza wpływu paradygmatu aspektowego na jakość kodu źródłowego

(rozszerzone streszczenie)

A.1 Wprowadzenie

Jedną z istotnych motywacji rozwoju technik wytwarzania oprogramowania jest dążenie do coraz lepszej separacji zagadnień. Terminu *separacja zagadnień* po raz pierwszy użył Dijkstra [1974] na określenie izolacji poszczególnych zagadnień w osobnych modułach w celu skoncentrowania uwagi tylko na jednym zagadnieniu w danym momencie oraz lokalizacji przyszłych modyfikacji [Dijkstra, 1974]. Pionierzy inżynierii oprogramowania już w latach 70. przedstawiali separację zagadnień jako wiodące kryterium modularyzacji oprogramowania [Parnas, 1972]. Modularność oznacza, że oprogramowanie składa się z luźno powiązanych modułów o wysokiej kohezji (tzn. silnych związkach wewnętrznych), ukrytej implementacji, udostępniających swe usługi poprzez dobrze wyspecyfikowane interfejsy [Yourdon & Constantine, 1979; Meyer, 1989]. Parnas [1972] traktuje modularyzację jako sposób na poprawę elastyczności i czytelności oprogramowania przy jednoczesnym skróceniu czasu jego wytwarzania. Jako korzyści z modularyzacji wymienia również możliwość dokonania znacznych modyfikacji w wybranym module bez potrzeby modyfikacji innych oraz możliwość studiowania systemu poprzez analizę pojedynczych modułów, niezależnie od siebie. Z kolei Wirth [1971; 1974] pisze, że modularność determinuje łatwość, z jaką oprogramowanie może być rozszerzone o nowe wymagania lub dostosowane do zmian w środowisku uruchomieniowym.

Praktyka pokazała, że tradycyjne paradygmaty programowania nie radzą sobie z separacją tzw. zagadnień przecinających (crosscutting concerns) [Tarr et al., 1999]. *Zagadnienia przecinające* dotyczą kwestji technicznych takich jak obsługa sytuacji wyjątkowych czy synchronizacja oraz wymagań niefunkcjonalnych takich jak autoryzacja czy zapewnienie trwałości obiektów. W tradycyjnych językach programowania implementacja takich zagadnień musi być, z powodu braku odpowiednich abstrakcji programistycznych, zagnieżdżona w

implementacji modułów będących rezultatem dekompozycji funkcjonalnej. Prowadzi to do dwóch niepożądanych zjawisk – przeplatania oraz rozpraszania kodu (code tangling and scattering). *Przeplatanie* kodu występuje, gdy implementacje różnych zagadnień współistnieją w ramach jednego modułu. Natomiast *rozpraszanie* występuje, gdy podobne fragmenty implementacji tego samego zagadnienia pojawiają się w wielu modułach. W celu rozwiązania tych problemów, Kiczales wraz z zespołem zaproponowali paradygmat aspektowy [Kiczales et. al., 1997]. Pierwszym, a jednocześnie do dzisiejszego dnia najbardziej popularnym językiem aspektowym jest AspectJ. Język ten został stworzony przez zespół Kiczalesa [Kiczales et al., 2001] jako rozszerzenie Javy. Obecnie rozwijany jest jako projekt Eclipse.

Paradygmat aspektowy (aspect-oriented programming) wprowadza nową jednostkę dekompozycji – *aspekt*. Aspekt podobnie jak klasa może posiadać atrybuty i metody. Dodatkowo, aby implementować zagadnienia przecinające, aspekt może definiować rady (advices), punkty przecięcia (pointcuts), oraz deklaracje między-typowe (inter-type declarations). *Punkt przecięcia* to konstrukcja programistyczna służąca do specyfikacji zbioru punktów złączeń (joinpoints) oraz ekstrakcji kontekstu punktu złączenia. *Punkt złączenia* to identyfikowalna lokalizacja w programie, do której można się odwołać z poziomu aspektu. Do typowych punktów złączeń należą: wywołanie oraz wykonanie metody, odczytanie oraz zapisanie wartości atrybutu, rzucenie wyjątku. *Rada* to sekwencja instrukcji, które zostaną wykonane we wszystkich punktach złączeń wskazanych przez punkt przecięcia skojarzony z daną radą. AspectJ definiuje trzy główne rodzaje rad: before, after i around. Są one aktywowane odpowiednio przed, po lub zamiast punktu złączenia. *Deklaracja między-typowa* umożliwia zmianę struktury klasy poprzez dodanie do niej z zewnątrz (spoza definicji klasy) nowych atrybutów oraz metod. Ponadto można zmienić nadklasę danej klasy oraz zadeklarować jakie interfejsy dana klasa implementuje.

W celu lepszego zrozumienia nowych konstrukcji rozważmy następujący przykład. Mamy za zadanie dodać nowe zagadnienie do istniejącego oprogramowania. Nowe zagadnienie polega na rejestrowaniu (logowaniu) wykonania każdej metody w systemie i pomiarze czasu jej trwania. W paradygmacie obiektowym, moglibyśmy zrealizować to zadanie poprzez modyfikację każdej metody w następujący sposób: na początku metody odczytać i zapamiętać czas systemowy; przed wyjściem z metody wysłać na standardowe

wyjście różnicę między bieżącym czasem systemowym a czasem zapamiętanym. Rozwiązanie to charakteryzuje się dwoma niekorzystnymi zjawiskami: (I) implementacja zagadnienia logowania przeplata się z implementacją podstawowego zagadnienia realizowanego przez metodę (jej efektu funkcjonalnego); (II) implementacja zagadnienia logowania rozproszona jest po wszystkich metodach w całym systemie. Innym sposobem jest stworzenie dla każdej klasy odpowiadającej jej podklasy i przesłonięcie (override) każdej metody w ten sposób, aby przed i po jej wykonaniu odczytać czas systemowy. Różnicę czasu należy podobnie jak w poprzednim rozwiązaniu przesłać na standardowe wyjście. Dodatkową uciążliwością tego rozwiązania jest potrzeba wprowadzenia inwazyjnych modyfikacji w miejscach, gdzie tworzone są instancje klas systemowych – należy utworzyć instancje klas implementujących logowanie. Rozwiązanie to nadal charakteryzuje się rozproszeniem implementacji zagadnienia logowania. Poniższy listing przedstawia implementację przedstawionego wyżej problemu w AspectJ.

```
public aspect TimeLogging {                                    //1
  pointcut eachMethod(): execution(* *.*(..));                 //2
  Object around(): eachMethod() {                              //3
    long start = System.currentTimeMillis();                   //4
    Object tmp = proceed();                                    //5
    long end = System.currentTimeMillis();                     //6
    long time = end - start;                                   //7
    Signature sig=thisJoinPointStaticPart.getSignature();//8
    System.out.println(sig +" - "+ time);                      //9
    return tmp;                                                //10
  }
}
```

W wierszu 2 definiujemy punkt przecięcia `eachMethod()` jako wykonanie dowolnej metody z dowolnej klasy. W wierszu 3 definiujemy radę skojarzoną z uprzednio zdefiniowanym punktem przecięcia. Rada ta „przechwyci” próbę wykonania dowolnej metody i w miejscu oryginalnej metody wykona własne instrukcje. W wierszu 5 za pomocą specjalnej instrukcji AspectJ wykonujemy oryginalną metodę. W wierszu 8 z kontekstu punktu złączenia odczytujemy sygnaturę oryginalnej metody. Rozwiązanie to jest wolne zarówno od zjawiska przeplatania jak i rozpraszania kodu gdyż sprowadza się do dodania nowego modułu, bez potrzeby ingerencji w kodzie modułów już istniejących.

Paradygmat aspektowy przynosi jednak nowe problemy, nieznane dotychczas w produkcji oprogramowania. Nowe konstrukcje wprowadzają nowe typy zależności międzymodułowych, które mogą utrudnić analizę kodu. Ponadto

języki aspektowe powinny posiadać dwie cechy [Filman & Friedman, 2000]: kwantyfikowalność (quantification) oraz nieświadomość (obliviousness). *Kwantyfikowalność* oznacza możliwość wskazania wielu nielokalnych miejsc w kodzie, do których zostanie zastosowana pewna instrukcja. *Nieświadomość* oznacza, że miejsca te nie muszą w żaden sposób być na to przygotowane. Kwantyfikowalność oraz nieświadomość mogą powodować problemy z modularnym wnioskowaniem [Leavens & Clifton, 2007; Figueiredo et al., 2008]. Istnieje zatem istotny z punktu widzenia inżynierii oprogramowania problem badawczy – ocena bilansu korzyści i strat wynikających z stosowania paradygmatu aspektowego.

A.2 Cel rozprawy

Podstawowym celem pracy jest ocena paradygmatu aspektowego z perspektywy możliwości tworzenia lepszej jakości kodu. Jakość kodu będzie rozważana w kategoriach: modularności, możliwości dalszego rozwoju (evolvability) oraz możliwości ponownego użyciu (reusability). Jako punkt odniesienia do oceny paradygmatu aspektowego wybrano paradygmat obiektowy, z dwóch względów: (I) paradygmat obiektowy zajmuje obecnie dominującą pozycję w produkcji oprogramowania; (II) paradygmat aspektowy czerpie z dorobku paradygmatu obiektowego i stanowi jego rozszerzenie.

A.3 Teza rozprawy

Nowo proponowane paradygmaty zyskują znaczenie, jeżeli są efektywniejsze od istniejących w rozwiązywaniu problemów, które uznano za wystarczająco istotne [Kuhn, 1962]. Dotychczasowe publikacje w większości prezentują stanowisko, że paradygmat aspektowy, ze względu na możliwość lepszej separacji zagadnień, umożliwia lepszą modularyzację oprogramowania [Figueiredo et. al., 2008; Filho et. al., 2006; Garcia et. al., 2005; Greenwood et. al., 2007; Sant'Anna et. al., 2003; Soares et. al., 2002], a co za tym idzie lepszą utrzymywalność (maintainability) oraz możliwość ponownego użycia [Beltagui, 2003; Sant'Anna et al., 2003; Mens et al., 2004; Zhao, 2004; Lemos et. al., 2006]. Stanowisko to nie jest jednak poparte przekonującymi dowodami naukowymi. Błędnie przyjmowane jest

założenie, że separacja na poziomie leksykalnym (dobrze wspierana przez paradygmat aspektowy) jest tożsama z separacją zagadnień w rozumieniu Dijkstry i Parnasa.

Punktem wyjścia dla badań, których wyniki przedstawia niniejsza rozprawa była krytyka powyższego założenia oraz związane z nią obserwacje (przedstawione w [Przybylek, 2010a, 2010b, 2011b]), że konstrukcje programistyczne proponowane przez paradygmat aspektowy mogą niekorzystnie skutkować w zakresie modularności kodu programu.

Wyniki badań przedstawionych w niniejszej dysertacji argumentują na rzecz następującej tezy:

I. *Paradygmat aspektowy umożliwia separację zagadnień przecinających na poziomie struktury kodu, narusza jednak podstawowe zasady modularyzacji, takie jak: niskie skojarzenie międzymodułowe, ukrywanie informacji, specyfikacja interfejsów.*

Liczne badania [Bieman & Kang, 1995; Hitz & Montazeri, 1995; Chaumun et al., 2000; Bowen et al., 2007; MacCormack et al., 2007; Perepletchikov et al., 2007; Breivold et al., 2008] pokazują, że lepsza modularność przekłada się na lepszą utrzymywalność oraz możliwość ponownego użycia oprogramowania. Wyniki te dotyczą badań nad oprogramowaniem tworzonym w oparciu o paradygmat strukturalny bądź obiektowy i nie ma pewności, czy te same prawidłowości zachodzą dla paradygmatu aspektowego. Możliwość badań nad utrzymywalnością i ponownym użyciem oprogramowania aspektowego jest ograniczona z powodu braku danych pochodzących z rzeczywistych zastosowań paradygmatu aspektowego w skali przemysłowej. W związku z tym przeprowadzono badania w środowisku laboratoryjnym, ograniczając się do akademickich przykładów. Badania te wykazały, że paradygmat obiektowy w większości przypadków okazał się bardziej efektywny. Wykryto również kilka sytuacji, w których widoczna była wyższość paradygmatu aspektowego. Skala tych badań była jednak zbyt wąska, aby sformułować definitywne konkluzje. Otrzymane wyniki wystarczają jednak do sformułowania dodatkowej tezy niniejszej rozprawy:

II. *W ograniczonym zakresie możliwe jest zastosowanie programowania aspektowego do poprawy modyfikowalności oraz możliwości ponownego użycia oprogramowania.*

A.4 Znaczenie podjętego problemu

Istotnym obszarem inżynierii oprogramowania są badania ukierunkowane na poprawę jakości oprogramowania z perspektywy programisty, w tym również redukcję czasu oraz kosztów jego wytwarzania i utrzymania. Naukowcy poszukują zatem sposobów na lepszą modularyzację systemów, aby ułatwić przyszłe modyfikacje i zwiększyć możliwość ponownego użycia raz wytworzonych modułów. Paradygmat aspektowy, gdyby spełnił związane z nim nadzieje, byłby kolejnym krokiem milowym w rozwoju technik programowania.

Obecnie, paradygmat aspektowy jest tematem licznych dyskusji w społeczności akademickiej. Zajął również trwała pozycję na prestiżowych konferencjach naukowych, m.in. na SPLASH (dawniej OOPSLA), ICSE, ACM SAC, ECOOP. Doczekał się także dedykowanej mu corocznej konferencji – AOSD (http://aosd.net). Tematem przewodnim konferencji AOSD'12 będzie modularność oprogramowania. Ponadto dostawcy oprogramowania tacy jak IBM, Motorola, Siemens i SAP wyrazili chęć poznania i stosowania paradygmatu aspektowego. Naukowcy SAP zaprezentowali nawet harmonogram adopcji paradygmatu aspektowego [Pohl et al., 2008].

Uzasadnione jest więc stwierdzenie, że paradygmat aspektowy jest obecnie w centrum zainteresowania środowisk badawczych zajmujących się technikami budowy oprogramowania oraz budzi zainteresowanie przemysłu informatycznego.

A.5 Metody badawcze

Główną metodą badawczą wykorzystaną w niniejszej dysertacji jest studium przypadków [Yin, 2003]. Jednostkami analiz są systemy informatyczne posiadające zarówno implementację obiektową jak i aspektową. Obie implementacje są porównywane pod względem skojarzenia (coupling) oraz kohezji (cohesion) tworzących je modułów.

Kolejną zastosowaną metodą badawczą jest kontrolowany eksperyment [Basili et al., 1999]. W środowisku laboratoryjnym wytworzono przykładowy program, a następnie poddano go ewolucji poprzez inkrementalną implementację nowych wymagań. W ramach tego procesu zbadano wpływ zastosowanego

paradygmatu (obiektowego i aspektowego) na możliwość rozwoju oraz ponowne wykorzystanie oprogramowania.

Oba badania empiryczne (studium przypadków oraz eksperyment kontrolowany) przeprowadzono posługując się podejściem GQM [Basili et al., 1994] do zdefiniowania celów badawczych oraz kontroli zakresu badań.

W niniejszej rozprawie zastosowano także badania aktywne [Davison et al., 2004; Easterbrook et al., 2007]. Badania te sformalizowano wykorzystując paradygmat „design-science" [Hevner et al., 2004]. W ich wyniku:

- stworzono notację rozszerzającą UML o możliwości prezentacji aspektów na diagramie klas; notacja ta została wykorzystana do wizualizacji kodu AspectJ w rozdziale 5 oraz 7;
- zaproponowano nową implementację wybranych wzorców projektowych; propozycje te ułatwiają użycie wzorców w różnych kontakstach.

A.6 Badania pokrewne

Prace najbardziej zbliżone do badań zaprezentowanych w niniejszej rozprawie można podzielić na cztery kategorie: (I) propozycje metryk skojarzenia międzymodułowego (coupling) odpowiednich dla paradygmatu aspektowego; (II) badania oceniające wpływ paradygmatu aspektowego na modularność oprogramowania; (III) badania oceniające wpływ paradygmatu aspektowego na utrzymywalność oraz ponowne użycie oprogramowania; oraz (IV) prace rozszerzające UML o możliwość modelowania aspektów.

Dotychczas powstało wiele metryk przeznaczonych do oceny skojarzenia modułów w oprogramowaniu aspektowym. Jednakże wartości uzyskane z tych metryk nie mogą być porównywane z wartościami uzyskanymi z metryk obiektowych (w szczególności z metryki CBO zdefiniowanej przez Chidamber & Kemerer [1994]). Zhao [2004], Ceccato & Tonella [2004], Shen & Zhao [2007], and Burrows et al. [2010a; 2010b] zaproponowali metryki mierzące konkretny rodzaj skojarzenia i dlatego nienadające się do bezpośredniego porównywania z metryką CBO. Z kolei metryka zaproponowana przez Sant'Anna et al. [2003] nie uwzględnia wszystkich rodzajów skojarzeń. W szczególności pomija tzw. zależności semantyczne. Zaproponowana w niniejszej rozprawie metryka CBO_{AO} mierzy łączne skojarzenie modułu i jest odpowiednikiem metryki CBO.

Istnieje wiele badań koncentrujących się na porównaniu modularyzacji obiektowej z aspektową w oparciu o metryki skojarzenia i kohezji. Badania te różnią się od prezentowanych w niniejszej rozprawie z kilku powodów. Garcia et al. [2005], Filho et al. [2006], Hoffman & Eugster [2007], Figueiredo et al. [2008], i Castor et al. [2009] porównują globalne wartości uzyskane z metryki skojarzenia oraz kohezji, podczas gdy w rozprawie porównywane są wartości przeciętne. Porównywanie wartości globalnych jest nieuprawnione, ponieważ wówczas im większy system (więcej modułów) tym większa wartość metryk. Natomiast modularność jest ortogonalna do rozmiaru systemu i w związku z tym metryki skojarzenia oraz kohezji nie powinny być od rozmiaru zależne.

Sant'Anna et al. [2003] i Garcia et al. [2005] stosują metryki, które nie uwzględniają zależności semantycznych, a więc pomijają bardzo istotny obszar zależności międzymodułowych. Pozostałe badania empiryczne można podzielić na dwie grupy. W pierwszej grupie [Filho et al., 2006; Greenwood et al., 2007; Madeyski & Szała, 2007; Figueiredo et al., 2008; Castor et al., 2009] nie uwzględniono nowych rodzajów zależności wprowadzanych przez punkty przecięcia. W drugiej grupie [Tsang et al., 2004; Hoffman & Eugster, 2007] skojarzenia wprowadzane przez punkty przecięcia są uwzględniane tylko wówczas, gdy skojarzony moduł jest jawnie wymieniany w definicji punktu przecięcia.

Sant'Anna et al. [2003], Garcia et al. [2005], Filho et al. [2006], Greenwood et al. [2007], Figueiredo et al. [2008] i Castor et al. [2009] mierzą także przeplatanie i rozproszenie kodu używając metryk zdefiniowanych przez Sant'Anna et al. [2003]. Otrzymane przez nich rezultaty świadczą, że implementacje aspektowe są lepsze od ich obiektowych odpowiedników. Rezultaty te były jednak przewidywalne i nieuniknione, ponieważ leksykalna separacja zagadnień jest podstawą paradygmatu aspektowego.

Istnieje również wiele badań ilościowych oceniających wpływ paradygmatu aspektowego na utrzymywalność oraz ponowne wykorzystanie oprogramowania. Różnią się one od badań zaprezentowanych w niniejszej rozprawie głównie stosowanymi metrykami. Kulesza et al. [2006] ocenia obiektową i aspektową implementacje serwisu internetowego przed i po dokonaniu scenariuszy zmian. Do oceny stosują metryki przeplatania i rozproszenia kodu, skojarzenia, kohezji oraz rozmiaru. Metryki te mierzą modularność, a nie utrzymywalność oprogramowania. Sant'Anna et al. [2003] symuluje scenariusze zmian w systemie wielo-agentowym. Dla każdego scenariusza trudność

modyfikacji oceniana jest liczbą modułów, operacji, oraz linii kodu, które zostały dodane, zmienione lub skopiowane. Podobne podejście stosowane jest przez Figueiredo et al. [2008] do oceny stabilności oprogramowania SPL (software product lines), które przechodzi siedem scenariuszy zmian. W badaniach zaprezentowanych w niniejszej rozprawie wykorzystano jedną metrykę do oceny ewolucyjności (liczba atomowych zmian niezbędna do realizacji scenariusza) i jedną do oceny możliwości ponownego użycia oprogramowania (stosunek linii kodu wykorzystanych z poprzedniej wersji do całkowitej liczby linii kodu w programie). Bartsch & Harrison [2008] mierzą ile czasu zajmuje wykonanie scenariusza zmian na przykładzie sklepu internetowego. Takie podejście może być uważane za uzupełniające w stosunku do prezentowanego.

Istnieją także badania jakościowe oceniające wpływ paradygmatu aspektowego na utrzymywalność oraz ponowne wykorzystanie oprogramowania [Hanenberg & Unland, 2001; Koppen & Störzer, 2004; Griswold et al., 2006; Kästner et al., 2007; Munoz et al., 2008; Mortensen, 2009; Taveira et al., 2009; Taveira et al., 2010]. Niniejsza rozprawa stanowi ich rozwinięcie i uzupełnienie. Hannemann & Kiczales [2002] stworzyli aspektowe implementacje 23 wzorców projektowych skatalogowanych przez Gamma et al. [1995]. Dla 12 wzorców znaleźli implementację nadającą się do ponownego wykorzystania. W niniejszej rozprawie przyjęto ich doświadczenia za punkt startowy i zastosowano AspecJ wraz z typami generycznymi i programowaniem refleksyjnym. Używając tych technik poprawiono implementację trzech wzorców.

W ostatniej dekadzie powstało wiele rozszerzeń UML umożliwiających modelowanie aspektów. Propozycje zaproponowane przez [Evermann, 2007; Fuentes & Sanchez, 2007; Gao et al., 2004; Groher & Baumgarth, 2004; Groher & Schulze, 2003; Mosconi et al., 2008; Stein et al., 2002a; Stein et al., 2002b; Zakaria et al., 2002] rozszerzają UML w oparciu o profile. Ich autorzy modelują aspekt jako stereotypowaną klasę, a radę jako stereotypowaną metodę. Ponieważ UML od wersji 2.0 wymaga semantycznej zgodności między typem stereotypowanym i zbudowanym na jego podstawie typem nowym, takie podejście jest nieuprawnione, ponieważ ani aspekt nie jest klasą, ani rada nie jest metodą. Rozszerzenie proponowane w tej rozprawie czerpie z rozwiązań opartych na tworzeniu metamodelu [Lions et al., 2002; Hachani, 2003a; Hachani, 2003b; Kande, 2003; Yan et al., 2004]. W odróżnieniu od propozycji rozszerzających UML 1.x [Lions et al., 2002; Hachani, 2003a; Hachani, 2003b], proponowane rozszerzenie bazuje

na UML 2.2. W przeciwieństwie do propozycji modyfikujących istniejący metamodel UML [Hachani 2003a; 2003b], proponowane rozszerzenie dodaje wyłącznie nowe metaklasy. Z kolei w odróżnieniu od [Hachani, 2003a; Hachani, 2003b; Yan et al., 2004] proponowane rozszerzenie dostarcza dedykowaną notację graficzną dla nowych konstrukcji wprowadzonych przez paradygmat aspektowy.

A.7 Wkład rozprawy w rozwój dziedziny

Poniżej omówiono główne obszary, w których wyniki zawarte w niniejszej rozprawie wnoszą istotny wkład w rozwój dziedziny związanej z zastosowaniem paradygmatu aspektowego do wytwarzania i rozwijania oprogramowania.

A.7.1 Ocena wpływu paradygmatu aspektowego na modularność oprogramowania

Wykazano, że stosowane dotychczas aspektowe metryki skojarzenia wyliczone dla implementacji aspektowej nie mogą być porównywane z metryką CBO wyliczoną dla implementacji obiektowej tego samego systemu. Wynika to z tego, że nie uwzględniają one wszystkich rodzajów zależności skojarzenia (coupling dependencies) wprowadzanych przez konstrukcje aspektowe. Na bazie krytyki istniejących metryk oraz odwołując się do idei leżącej u podstaw metryki CBO (jeżeli do zrozumienia modułu X niezbędne jest uprzednie przeanalizowanie modułu Y to moduł X jest skojarzony z modułem Y), zaproponowano nową metrykę skojarzenia CBO_{AO}, która jest pełnym odpowiednikiem metryki CBO.

Następnie wykorzystano metrykę skojarzenia CBO_{AO} oraz metrykę kohezji LCOM (zaadoptowaną na potrzeby paradygmatu aspektowego przez Ceccato & Tonella [2004]) do porównania modularności implementacji obiektowych z implementacjami aspektowymi 11 rzeczywistych systemów (Tabela 1) oraz 23 wzorców projektowych (Rysunek 1).

Tabela 1. Wartości metryk rozmiaru oraz modularności dla 11 rzeczywistych systemów

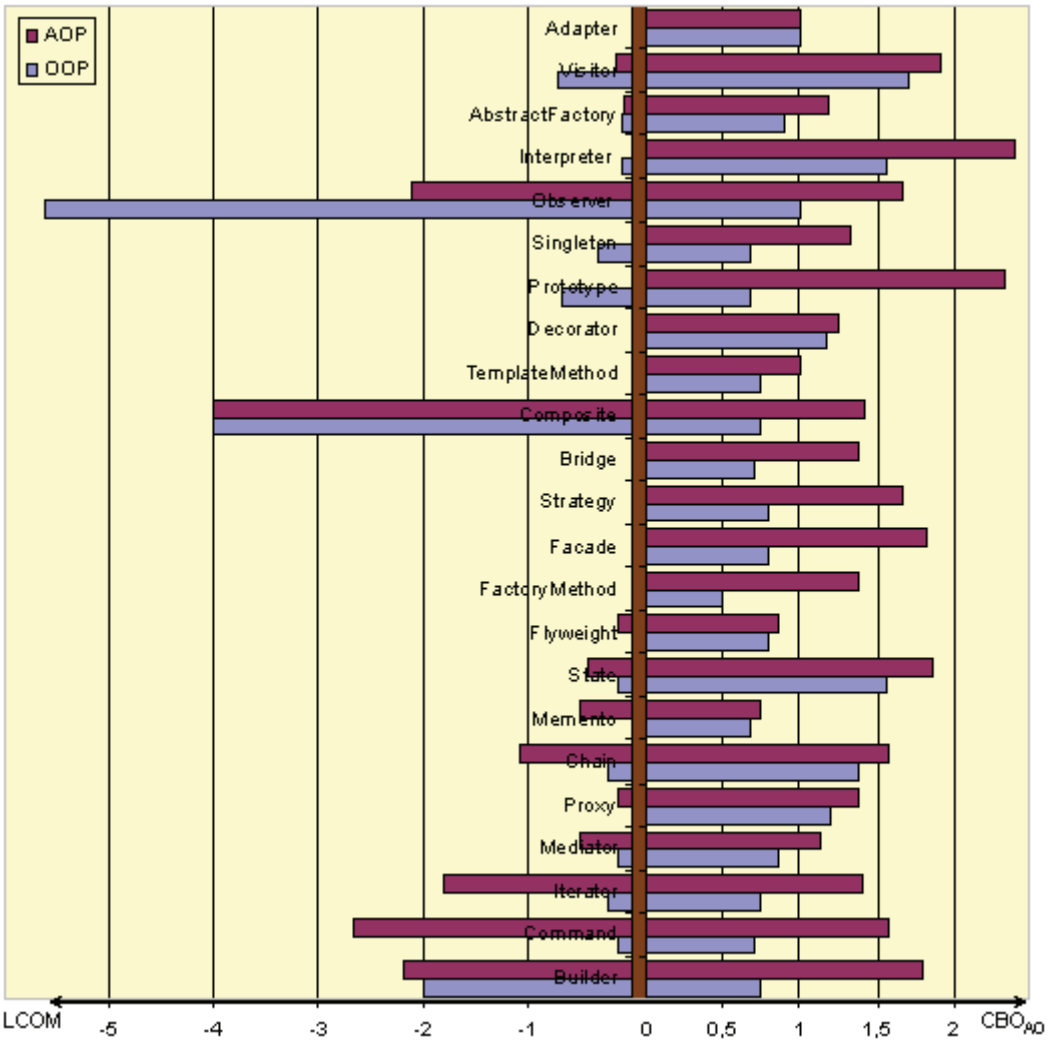
I	II	III	IV	V	VI
		VS	**LOC**	**CBO_{AO}**	**LCOM**
Telestrada	**OO**	233	3424	0,81	1,86
	AO	242(18)	3350	0,95	2,17
	Δ	4%	-2%	18%	16%
PetStore	**OO**	345	17798	2,32	20,63
	AO	382(37)	17914	2,76	20,19
	Δ	11%	1%	19%	-2%
CVS	**OO**	257	18876	5,76	71,31
	AO	261(4)	19423	-	73,90
	Δ	2%	3%	-	4%
EImp	**OO**	123	8708	1,84	1,53
	AO	126(3)	9041	-	1,68
	Δ	2%	4%	-	10%
Checkstyle	**OO**	283	18083	7,61	16,01
	AO	330(23)	20101	7,41	22,67
	Δ	17%	11%	-3%	42%
Health Watcher	**OO**	88	6096	3,19	9,24
	AO	103(12)	5768	4,20	7,63
	Δ	17%	-5%	32%	-17%
JHotDraw	**OO**	398	22724	3,57	75,04
	AO	438(31)	23167	3,66	65,70
	Δ	10%	2%	3%	-12%
Hypercast	**OO**	370	50492	3,31	67,24
	AO	391(7)	51207	3,42	67,00
	Δ	6%	1%	4%	-0,4%
Prevayler	**OO**	167	5043	1,87	9,31
	AO	168(55)	4179	2,56	7,01
	Δ	1%	-17%	37%	-25%
Berkeley DB	**OO**	340	41651	4,38	126,31
	AO	452(107)	38770	4,73	78,21
	Δ	33%	-6,9%	8%	-38%
HSQLDB	**OO**	402	80736	4,11	226,91
	AO	413(25)	76210	4,12	247,30
	Δ	3%	-6%	0,3%	9%

Kolumna III (Vocabulary Size) przedstawia liczbę modułów składających się na daną implementację. Dodatkowo dla implementacji aspektowych podano w nawiasie liczbę aspektów. Kolumna IV (Lines of Code) prezentuje liczbę linii kodu. Wiersze oznaczone 'Δ' zawierają procentową różnicę między wartością metryki dla implementacji obiektowej i aspektowej. Pomimo, że dokładne wartości CBO_{AO} nie są znane dla aspektowej implementacji CVS i EImp, wartości te są wyższe niż w odpowiadającym im implementacjach obiektowych.

W przypadku rzeczywistych systemów, tylko jedna implementacja aspektowa charakteryzowała się mniejszym średnim skojarzeniem niż odpowiadająca jej

implementacja obiektowa, podczas gdy dziesięć implementacji miało średnie skojarzenie większe. Implementacje aspektowe wykazały się lepszą[1] średnią kohezją w sześciu przypadkach, podczas gdy obiektowe w pięciu.

W przypadku wzorców aspektowych przewaga paradygmatu obiektowego była jeszcze bardziej widoczna (im krótsze słupki tym implementacja lepsza). Tylko dla jednego wzorca średnie skojarzenie było identyczne dla obu implementacji, natomiast dla pozostałych wzorców moduły w implementacjach obiektowych były luźniej powiązane. Średnia kohezja była wyższa dla dziewięciu implementacji obiektowych i dla sześciu aspektowych. Dla pozostałych ośmiu wzorców obie implementacje miały identyczną średnią kohezję.



Rysunek 1. Wartości metryk modularności dla 23 wzorców projektowych.

[1] lepsza oznacza wyższą kohezję, czyli taką, która ma mniejszą wartość LCOM

Ponieważ zastosowane metryki nie wyczerpują wszystkich wymiarów modularności przedyskutowano także inne kryteria. W efekcie uzasadniono, że paradygmat aspektowy istotnie narusza podstawowe zasady modularyzacji głoszone przez Parnasa, Dijkstrę oraz innych guru inżynierii oprogramowania. W szczególności paradygmat aspektowy:

- promuje programowanie niestrukturalne;
- narusza enkapsulację;
- wprowadza interfejsy bez jawnej specyfikacji;
- utrudnia modularne wnioskowanie;
- prowadzi do naruszenia kontraktu między modułem bazowym a jego klientami;
- eskaluje skojarzenia między modułami.

Przeanalizowano również propozycje nowych języków aspektowych, które starają się przywrócić modularność kosztem redukcji nieświadomości (obliviousness). W propozycjach tych podejście aspektowe traci jednak w dużej mierze możliwość bezinwazyjnej modyfikacji istniejących modułów, a jednocześnie powraca problem rozpraszania oraz przeplatania kodu.

A.7.2 Ocena wpływu paradygmatu aspektowego na możliwości rozwoju oraz ponownego użycia oprogramowania

W celu oceny wpływu paradygmatu aspektowego na możliwości rozwoju oraz ponownego użycia oprogramowania przeprowadzono kontrolowany eksperyment, w którym stworzono prosty system implementujący problem producenta-konsumenta. Następnie system ten poddano inkrementalnym modyfikacjom polegającym na realizacji nowych zagadnień przecinających. Na każdym etapie stworzono zarówno wersję obiektową jak i aspektową. Okazało się, że implementacja aspektowa jest lepsza tylko w przypadku odłączania pewnych zagadnień przecinających oraz w przypadku implementacji zagadnienia logowania, które jest sztandarowym przykładem zastosowania paradygmatu aspektowego. Ocenę poszczególnych implementacji dokonano przy pomocy dwóch metryk. Pierwsza metryka (AC – Atomic Changes) zliczała liczbę atomowych modyfikacji niezbędnych do transformacji z jednej wersji oprogramowania do kolejnej. Druga metryka (RL – Reuse Level) mierzyła stosunek linii kodu wykorzystanych z poprzedniej wersji oprogramowania do całkowitej liczby linii kodu. Linie kodu

uznawano za ponownie wykorzystane wyłącznie w przypadkach, kiedy ponownie używano modułu, w którym zostały pierwotnie umieszczone. Rezultaty eksperymentu przedstawia Tabela 2.

Tabela 2. Wartości metryk AC oraz RL dla poszczególnych scenariuszy

Scenariusz	AC		RL	
	OOP	AOP	OOP	AOP
Dodanie zagadnienia synchronizacji	7	19	0,71	0,66
Dodanie znacznika czasu	8	19	0,85	0,67
Dodanie zagadnienia logowania	9	6	0,88	0,95
Dodanie nowych metod dostępowych	9	16	0,73	0,58
Usunięcie logowania oraz znakowania	5	3	0,74	1,00

Z przeprowadzonego eksperymentu uzyskano następujące doświadczenie. W przypadku kodu aspektowego programista analizujący pewną klasę w izolacji, nie jest w stanie stwierdzić, czy modyfikacja implementacji tej klasy (bez naruszania interfejsu) narusza jakieś aspekty. Ponadto, definiując punkty przecięcia programista musi posiadać globalną wiedzę o strukturze programu. Ponieważ definicje punktów przecięcia opierają się na strukturalnych własnościach programu, definicje te mogą stać się niepoprawne w trakcie dalszej ewolucji programu. Ponadto, w większości przypadków aspekty nie mogą być generyczne, ponieważ muszą zawierać informacje specyficzne dla konkretnego użycia, np. modyfikowane klasy. Istotnie ogranicza to możliwość ponownego wykorzystania raz zdefiniowanych aspektów.

Zbadano również możliwości poprawy aspektowych implementacji wzorców projektowych zaproponowanych przez Hannemann & Kiczales [2002] oraz Borella [2003]. Wykorzystując typy generyczne stworzono nową implementację wzorca Dekorator, która niemal w całości może być wykorzystana w różnych kontekstach. Jedynym zadaniem programisty pozostaje zdefiniowanie w subaspekcie jakie zdarzenia mają powodować dekorację oraz wskazanie dekorowanych obiektów. Zaproponowano także generyczną implementację wzorca Proxy, jednak w tym przypadku korzyści w odniesieniu do implementacji zaproponowanej przez Hannemann & Kiczales [2002] nie są aż tak znaczące. Ponadto, stosując AspectJ wraz z programowaniem refleksyjnym stworzono domyślną implementację klonowania dla wzorca Prototyp.

Wszystkie zaproponowane implementacje wzorców projektowych można bezinwazyjnie włączyć do istniejącego oprogramowania, definiując w subaspektach jedynie specyficzne informacje o uczestnikach wzorca.

Wpływ paradygmatu aspektowego na utrzymywalność (maintainability) nie jest oczywisty. Z jednej strony lokalizacja kodu związanego z danym wzorcem w ramach aspektu może poprawić utrzymywalność. Z drugiej strony wyższa złożoność poznawcza (cognitive complexity) utrudnia zrozumienie kodu. Wyjaśnienie tych zależności wymaga dalszych szeroko zakrojonych badań eksperymentalnych.

A.7.3 Rozszerzenie metamodelu UML na potrzeby modelowania aspektów

Zaproponowano rozszerzenie metamodelu UML o elementy umożliwiające dołączenie aspektów do diagramu klas. Nowe elementy specyficzne dla paradygmatu aspektowego zdefiniowano w analogiczny sposób jak zdefiniowany jest metamodel UML. Możliwość wizualizacji kodu aspektowego poprawia śledzenie zależności (traceability) pomiędzy projektem oprogramowania a jego implementacją. Zaproponowaną notację wykorzystywano do graficznej reprezentacji programów będących przedmiotem badań.

A.8 Upubliczniony dorobek badań

Wyniki badań przedstawione w niniejszej rozprawie zostały już opublikowane na dziewięciu międzynarodowych konferencjach oraz jako rozdział w książce. Tabela 3 wiąże te publikacje z poszczególnymi rozdziałami rozprawy.

Tabela 3. Pokrycie publikacjami rozdziałów rozprawy

Publikacja	Rozdział
[Przybyłek, 2007]	2, 3
[Przybyłek, 2008a]	4
[Przybyłek, 2008b]	7.2
[Przybyłek, 2009]	2, 3
[Przybyłek, 2010a]	6, 5.2
[Przybyłek, 2010b]	7.2
[Przybyłek, 2010c]	3.3
[Przybyłek, 2011a]	1, 8
[Przybyłek, 2011b]	6, 5.2
[Przybyłek, 2011c]	7.1, 5.3

A.9 Podsumowanie

Wśród badaczy paradygmatu aspektowego rozpowszechnione jest przekonanie, że poprawia on modularność oprogramowania w odniesieniu do paradygmatu obiektowego. Niniejsza dysertacja podważa to przekonanie zarówno na gruncie teoretycznym (rozdział 3.3) jak i empirycznym (rozdział 6). Podstawą przeprowadzonej krytyki jest odwołanie się do pojęcia separacji zagadnień w rozumieniu Dijkstry i Parnasa oraz wykazanie, że nie może ono być utożsamione z leksykalną separacją zagadnień. W obecnym ujęciu, paradygmat aspektowy poprzez zapobieganie przeplataniu i rozpraszaniu kodu poprawia jedynie separację leksykalną. Natomiast, jak to zademonstrowano w rozprawie, paradygmat aspektowy narusza podstawowe zasady modularyzacji wypracowane przez Parnasa, Dijkstrę, Yourdona, Constantine, Meyera oraz inne autorytety w inżynierii oprogramowania.

W ten sposób wykazano pierwszą część tezy niniejszej rozprawy, która brzmi następująco:

> *Paradygmat aspektowy umożliwia separację zagadnień przecinających na poziomie struktury kodu, narusza jednak podstawowe zasady modularyzacji, takie jak: niskie skojarzenie międzymodułowe, ukrywanie informacji, specyfikacja interfejsów.*

Panuje również przeświadczenie, że paradygmat aspektowy poprawia utrzymywalność oraz możliwości ponownego użycia oprogramowania. Badania eksperymentalne mające na celu weryfikację tego stwierdzenia (rozdział 7) ani przeprowadzona analiza rezultatów badań innych naukowców (rozdział 7.1.7) nie potwierdzają takiego stanowiska. W przeprowadzonym i opisanym w rozprawie kontrolowanym eksperymencie, podczas implementacji nowych wymagań tylko w jednym przypadku na cztery wersja aspektowa okazała się lepsza od obiektowej. Jednak ograniczony rozmiar badań nie pozwala na stawianie definitywnych konkluzji o wyższości któregokolwiek paradygmatu. Wystarczająco mocne badania będą możliwe do zrealizowania dopiero wtedy, gdy paradygmat aspektowy zostanie zaakceptowany przez przemysł. Niemniej, w warunkach laboratoryjnych zaobserwowano także sytuacje (np. implementacja zagadnienia logowania, implementacja niektórych wzorców projektowych), w których implementacja aspektowa zapewniała lepszą modyfikowalność oraz możliwość ponownego wykorzystania oprogramowania. Uzasadnia to drugą tezę tezy niniejszej rozprawy:

W ograniczonym zakresie możliwe jest zastosowanie programowania aspektowego do poprawy modyfikowalności oraz możliwości ponownego użycia oprogramowania.