Universidade Federal de Pernambuco - UFPE
Centro de Informática - CIn

Pós Graduação em Ciência da Computação

MODULARITY ANALYSIS OF USE CASE IMPLEMENTATIONS

Fernanda Rodrigues dos Santos d'Amorim

DISSERTAÇÃO DE MESTRADO

Recife - PE
06 de agosto de 2010

Universidade Federal de Pernambuco - UFPE
Centro de Informática - CIn

Fernanda Rodrigues dos Santos d'Amorim

MODULARITY ANALYSIS OF USE CASE IMPLEMENTATIONS

*Trabalho apresentado ao Programa de Pós Graduação em
Ciência da Computação do Centro de Informática - CIn da
Universidade Federal de Pernambuco - UFPE como requi-
sito parcial para obtenção do grau de Mestre em Ciência
da Computação.*

Orientador: *Paulo Henrique Monteiro Borba*

Recife - PE
06 de agosto de 2010

Catálogo na fonte
Bibliotecária Jane Souto Maior, CRB4-571

d'Amorim, Fernanda Rodrigues dos Santos
Modularity analysis of use case implementations / Fernanda
Rodrigues dos Santos d'Amorim - Recife: O Autor, 2010.
xi, 84 folhas : il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Dissertação (mestrado) Universidade Federal de
Pernambuco. Cln. Ciência da computação, 2010.

Inclui bibliografia.

1. Engenharia de software. 2. Projeto e arquitetura de
software. I. Borba, Paulo Henrique Monteiro (orientador). II.
Título.

Dissertação de Mestrado apresentada por **Fernanda Rodrigues dos Santos d'Amorim** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Modularity Analysis of Use Case Implementations**", orientada pelo **Prof. Paulo Henrique Monteiro Borba** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Sergio Castelo Branco Soares
Centro de Informática / UFPE

Prof. Claus Brabrand
IT University of Copenhagen (ITU)

Prof. Paulo Henrique Monteiro Borba
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 6 de agosto de 2010.

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

To Alcides Restelli Tedesco (Tchê).

ACKNOWLEDGEMENTS

Primeiramente, gostaria de agradecer ao Professor Paulo Borba por me orientar e me motivar durante estes dois anos de mestrado. Pelos comentários e perguntas desafiadoras que ajudaram a aprimorar os resultados do meu trabalho.

Agradeço a toda minha família. Em especial: a minha mãe, que sempre foi o meu maior exemplo de vida; a Marcelo, meu querido marido; e aos meus 2 filhos amados, Léo e João, que apesar de sempre sugarem minhas energias quando mais precisava trabalhar, com um sorriso eram capazes de me recarregar (mesmo que virtualmente) e me darem força para continuar.

Gostaria de agradecer também aos amigos feitos durante o mestrado, em especial, Carlos, Leopoldo, Márcio, Rodrigo e Henrique. Podem ter certeza que vocês me ajudaram de alguma forma durante estes anos. Também gostaria de agradecer aos demais membros do SPG por sempre promoverem discussões interessantes que agregaram valores à minha formação.

Finalmente, agradeço ao CNPq por financiar a minha pesquisa.

RESUMO

Atualmente, arquitetura baseada em componentes é a abordagem mais utilizada no desenvolvimento de softwares complexos; esta tem como principal objetivo a atribuição dos requisitos da aplicação aos componentes. Uma das técnicas mais difundidas para especificação de requisitos é a utilização de Casos de Uso. Em geral, arquiteturas de software baseadas em componentes resultam em implementações onde o código relativo a um caso de uso está espalhado e entrelaçado em diversos componentes do sistema, caracterizando um *crosscutting concern*. Isto ocorre porque técnicas tradicionais, como Orientação a Objetos (OO), não oferecem mecanismos que sejam capazes de modularizar este tipo de *concern*. Recentemente, novas técnicas de modularização, como aspectos, mixins e classes virtuais, foram propostas para tentar resolver este problema. Estas técnicas podem ser usadas para agrupar o código relacionado a um único caso de uso em uma nova unidade de modularização.

Este trabalho analisa qualitativa e quantitativamente o impacto causado por este tipo de modularização de casos de uso. Nós exploramos duas técnicas baseadas em Orientação a Aspectos (OA): (i) Casos de Uso como Aspectos - onde utilizamos os construtores de AspectJ para isolar todo código relativo à implementação de um caso de uso em um aspecto; e (ii) Casos de Uso como Colaborações Plugáveis - onde usamos os construtores de CaesarJ para modularizar implementações de casos de uso através de uma composição hierárquica de colaborações. Nós executamos dois estudos de casos onde comparamos as implementações OA de casos de uso com sua implementação OO. No processo de avaliação extraímos métricas tradicionais e contemporâneas incluindo coesão, acoplamento e separação de *concerns* e analisamos modularidade em termos de atributos de qualidade de software como: plugabilidade, rastreabilidade e suporte para desenvolvimento em paralelo. Nossos resultados indicam que modularidade é um conceito relativo e sua análise depende de outros fatores além do sistema alvo, das métricas e da técnica aplicada.

Palavras-chave: Modularidade, Casos de Uso, Programação Orientada a Aspectos, Avaliação Empírica

ABSTRACT

Today, component-based architecture is the most used approach to develop complex softwares; it relies on a process where the main goal is the assignment of requirements to components. One of the most popular technique for requirement specification is Use Cases. Component-based architecture often results in implementations with use cases code tangled and scattered across components characterizing what is so called a cross-cutting concern. This happens because traditional modularity techniques, like Object Orientation (OO), do not provide a modularization mechanism to deal with this kind of concern. Recently, new modularity techniques, like aspects, mixins, and virtual classes have been proposed to solve this problem. One can use such techniques to group together code related to a single use case.

This work analyzes qualitatively and quantitatively the impact of this kind of use case modularization. We explore two techniques based on Aspect Orientation (AO): (i) Use Case as Aspect - where we use AspectJ constructs to isolate all code related to a use case into an aspect; and (ii) Use Case as Pluggable Collaboration - where we use CaesarJ constructs to enable the modularization of use cases code through a hierarchical composition of collaborations. We run two case studies where we compare the AO implementation of use cases with their OO reference implementation. In the evaluation process we extract traditional and contemporary metrics, including cohesion, coupling and separation of concerns and analyze modularity in terms of software quality attributes, such as, support for parallel development, traceability, changeability, and pluggability. Our findings indicate that modularity is a relative concept and its analysis depends on others factors beyond the chosen system, metrics, and the applied technique.

Keywords: Modularity, Use Cases, Aspect Oriented Programming, Empirical Evaluation

CONTENTS

Chapter 1—Introduction	1
1.1 Motivation and Purpose	1
1.2 Contributions	4
1.3 Outline	4
Chapter 2—Background	6
2.1 Use Case Driven Development	6
2.2 Software Modularity	8
2.3 Modularizing Crosscutting Concerns	9
2.4 Aspect-Oriented Programming	10
2.4.1 AspectJ	10
2.4.2 CaesarJ	12
2.5 Software Maintenance and Evolution	14
Chapter 3—Modularity of Use Case Implementations	17
3.1 Use Case Implementations as Crosscutting Concerns	17
3.2 Techniques	21
3.2.1 Use Case as Aspect - with AspectJ	22
3.2.2 Use Case as Pluggable Collaboration - with CaesarJ	26
3.3 Metrics	30
3.3.1 Metrics Description	33
3.3.1.1 Separation of Concerns	33
3.3.1.2 Complexity	34
3.3.1.3 Changeability	35
3.3.1.4 Traceability	36
3.3.1.5 Pluggability	36
3.3.1.6 Parallel Development	37
3.3.2 Metrics Computation	37

Chapter 4—Case Studies	39
4.1 Library System	40
4.1.1 System Selection	41
4.1.2 System Design	41
4.1.3 Setup	42
4.1.4 Change Requests (CR)	43
4.1.5 Results	44
4.1.5.1 Quantifying Separation of Concerns (SoC)	44
4.1.5.2 Quantifying Complexity of Source Code	48
4.1.5.3 Quantifying Changeability	52
4.1.5.4 Quantifying Traceability, Pluggability and Parallel Development	53
4.2 Health Watcher	56
4.2.1 System Selection	57
4.2.2 System Design	57
4.2.3 Setup	58
4.2.4 Change Requests (CR)	59
4.2.5 Results	61
4.2.5.1 Quantifying Separation of Concerns (SoC)	61
4.2.5.2 Quantifying Complexity of Source Code	64
4.2.5.3 Quantifying Changeability	67
4.2.5.4 Quantifying Traceability, Pluggability and Parallel Development	69
4.3 Threats to Validity	71
4.4 Conclusions and Discussion	72
Chapter 5—Conclusions	74
5.1 Related Work	75
5.1.1 Use case Implementations Modularity Techniques	75
5.1.2 Empirical Studies in Aspect-Orientation	76
5.2 Future Work	77

LIST OF FIGURES

3.1	Library Management System represented by interconnected components.	17
3.2	Tangling and scattering when realizing use cases.	19
3.3	Use cases to components, based on [JN04]	20
3.4	Classes composed from different use cases code, based on [JN04]	22
3.5	Element Structure and Use Case Module	23
3.6	Use case implementations with use case modules, based on [JN04]	23
3.7	Check Out Book use case module representation	25
3.8	Element Structure and Use Case Collaborations	27
3.9	Representation of the Check Out Book use case collaboration	29
3.10	Software Quality Attributes	30
4.1	Library System Use Case Model.	41
4.2	Library Management System Architecture.	42
4.3	Library System experiment setup.	43
4.4	Library System - Separation of Concerns	46
4.5	Library System - Vocabulary Size and Lines of Code	49
4.6	Library System - Coupling and Cohesion	50
4.7	Library System - Average Coupling and Cohesion	51
4.8	Library System - Deph of Iheritance Tree	51
4.9	Use Case Model for the Original HW Implementation.	56
4.10	HW AO original architecture design.	58
4.11	Case Study Setup.	59
4.12	HealthWatcher Use Case model after CR1.	60
4.13	HealthWatcher - Separation of Concerns	63
4.14	HealthWatcher - Vocabulary Size and Lines of Code	65
4.15	HealthWatcher - Coupling and Cohesion	66
4.16	Health Watcher - Average Coupling and Cohesion	67
4.17	HealtWatcher - Deph of Iheritance Tree	67

LIST OF TABLES

3.1	GQM to assess modularity improvement.	32
4.1	Library System - Separation of Concerns Metrics	45
4.2	Library System - Degree of Tangling Metrics	48
4.3	Library System - UC_OO Complexity Metrics	48
4.4	Library System - UC_ASP) Complexity Metrics	49
4.5	Library System - Changeability Absolute Values	52
4.6	Library System - Changeability Percentage	52
4.7	Library System - Pluggability	54
4.8	Library System - Parallel Development	54
4.9	Library System - Traceability	55
4.10	HeathWatcher AO imlementation - Separation Of Concerns	62
4.11	Health Watcher - Degree of Tangling	64
4.12	HealthWatcher - Use Case Implementations OO	64
4.13	HealthWatcher - Use Case as Aspect	64
4.14	HealthWatcher - Changeability Absolute Values	68
4.15	HealthWatcher - Changeability Percentage	68
4.16	Health Watcher - Pluggability	69
4.17	HealthWatcher - Parallel Development	70
4.18	HealthWatcher - Traceability	70

LISTINGS

2.1	AspectJ Intertype Declaration Example	11
2.2	AspectJ Pointcut Example	11
2.3	AspectJ Advice Example	11
2.4	CaesarJ cclass Example	12
2.5	CaesarJ Virtual Class Example	13
2.6	CaesarJ Collaboration Example	14
2.7	CaesarJ Mixin Example	14
3.1	AspectJ constructs examples	24
3.2	Code for aspect CheckOutBook.aj	25
3.3	Caesar Class	27
3.4	Collaboration LibraryDomain	28
3.5	LibraryDomain	28
3.6	LibraryApp	28
3.7	LibrarySystem composed by use case collaborations	28
3.8	CheckOutBook Collaboration	29
3.9	Concern model file	38

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION AND PURPOSE

Development and Maintenance of complex software systems are challenging tasks. Modularity plays an important role to achieve success on these tasks. It allows to break down a problem into smaller parts and isolate different concerns into separate modules. In this scenario, developers make use of several well-established modularization mechanisms aiming at producing systems that are easier to reuse (in other projects), evolve and develop in parallel. These capabilities enable projects to better manage time, scope, cost and quality.

Component-based decomposition [HC01, CL02] is one of the most used modularity technique to develop complex software. It is based on a process where the final goal is the assignment of system requirements to components. Use cases is a traditional form of requirement specification that focuses on how the system will be used by people or other systems to accomplish particular goals [BS03]. The user-centric view of the system helps to prioritize lists of requirements and to ensure that the final system meets the user specification. The use case model and the requirement specification represent the problem domain and drive the rest of the development process. This is why this approach to software development is known as *use case driven development* [JB00]. In use-case driven development one can isolate a use-case abstraction in a single modular unit in all development phases except implementation [Jac03]. Usually, component-based architectures lead to a system with use cases code spread over multiple components. This happen because traditional modularity techniques, such as Object Orientation, decompose use case implementations (how use cases are realized into code) across components. A typical use case is realized by a collaboration of components and each component may be related to various use cases. **Thereafter, most use-cases are crosscutting concerns with respect to implementation.**

Some potential problems arise because of the crosscutting nature of use case implementations. A software system in its requirement, analysis and design phases has a well-defined structure involving actors and use cases. During implementation, these artifacts are spread into multiple, sometimes overlapping software modules [BL08]. This results in systems with a higher level of complexity regarding traceability issues, as we

cannot distinct which software module implements which scenario. In this situation, it is not easy to uncover requirements just looking at the code; this impacts understandability hindering the learning curve for developers. Poor understandability leads to poor maintainability. We cannot keep *use case modularity* at the implementation level and the resulting system has its use cases code scattered and tangled. The loss of use case modularity can result in implementations that are difficult to maintain under changing requirements. Modifying or adding a use case in order to fulfill a change request can cause invasive changes as the developer has to deal with a great number of files, components, classes, methods, and etc. Such modifications can introduce inconsistencies in the code and can increase maintenance cost.

As observed so far, we can state that a use case implementation is fundamentally a crosscutting concern. It is well known that crosscutting concerns reduce modularity, making programs harder to understand, develop in parallel, maintain, and evolve [KLL⁺97, EZS⁺08]. Modularity techniques have been recently proposed to overcome the limitation of Object Orientation (OO) and others traditional modularity techniques on dealing with such kind of concerns. Among them, Aspect-Oriented Programming (AOP) [KLL⁺97] has been considered the main technology on providing new mechanisms to improve modularity of crosscutting concerns. For example, transaction control is a common concern that requires consistent handling, usually, in a great number of classes. AOP offers a new form of modularity in order to encapsulate the crosscutting concerns and, this way, recover the overall system modularity. These concerns (aspects) are identified, modularized, developed, and then composed to generate the whole system in a weaving process. For instance, the modularization mechanisms of AOP offer new possibilities to promote separation of crosscutting concerns. Therefore, they can be used for isolating use case implementations, creating modules with code related to a single use case [Jac03, HHM04], which we explore here.

The main goal of our work is to evaluate to what extent *modularity of use case implementations improve the overall system's modularity*. With this purpose, we (i) analyze how uses cases are structured at the code level; (ii) compare different forms of use case implementations; (iii) and, finally, point out which form of use case decomposition is more suitable to accomplish improvement on the overall system's modularity.

In addition, we answer the following research questions in this dissertation:

- RQ1. Does modularity of use case implementations reduce modularity of non use-case concerns?
- RQ2. Does modularity of use case implementations improve overall maintenance and evolution?

RQ3. Which modularity dimensions benefit from modularized use case implementations?

We review these questions and summarize the evaluation of our work in Chapter 5.

In this work, we apply AOP to isolate use case implementations. We explore two techniques: (i) Use Case as Aspect - where we use AspectJ [KHH⁺01] constructs to isolate all code related to a use case into an aspect; and (ii) Use Case as Pluggable Collaboration - where we use CaesarJ [AGMO06] constructs to enable the modularization of use cases code through a hierarchical composition of collaborations. It is well known that AOP is effective to modularize conventional crosscutting concerns, such as, persistence, concurrence and transaction control. But to the best of our knowledge it is not yet clear how AOP behaves for modularizing use case implementations. We find some works [HHM04, Jac03, BL08] showing how AOP can be used to enable such modularization; they also report some benefits, such as, improvement on support for parallel development, traceability, and changeability. However, their observations rely only on a simple hypothetical evaluation. We were not able to uncover empirical evidences of such effects. In fact, a quantitative evaluation and also a qualitative evaluation based on evidence data are missing in order to provide a fair analysis regarding the real modularity gain when we modularize use case implementations.

To this end, we run two cases studies where we compare use cases implemented using AOP mechanisms (with AspectJ and CaesarJ) and use cases implemented in the traditional object oriented way (with Java). In the evaluation process we assess qualitatively and quantitatively the impact of this kind of use case modularization. We setup a GQM [BCR94] plan to guide the selection of software metrics. We group the metrics based on a set of software quality attributes that are the focus of our evaluation. For instance, we extract traditional and contemporary metrics, including cohesion, coupling and separation of concerns and analyze modularity in terms of software quality attributes, such as, support for parallel development, traceability, changeability, and pluggability. Our findings indicate that modularity is a relative concept and its analysis depends on others factors beyond the chosen system, metrics, and the applied technique.

By producing a quantitative evaluation about modularity of use case implementations with AOP, we provide the means to understand, in a more precisely way, the impact of this new technique. It is also possible to objectively point out if, when, and how, such kind of modularization should be used more efficiently and, at the same time, be able to identify where it can be improved.

1.2 CONTRIBUTIONS

The main contributions of this research are the following:

- A detailed process on the application of two AOP-based techniques to separate use case implementations in source code.
- A detailed process on the selection of software metrics related to software quality attributes used in the evaluation of this work.
- Results of two case studies complementing the empirical body of knowledge on the use of AOP. The main goal of our studies is to analyze and evaluate how AOP behaves on modularizing use cases implementations. We provide qualitative and quantitative analysis considering maintainability related tasks and a rigorous evaluation of the techniques, which considers different attributes such as separation of concerns, complexity of source code, changeability and support for parallel development.
- Based on the study results, we were able to answer our research questions and indicate which scenarios benefit best when a system has its use case implementations as the main form of modular decomposition.

1.3 OUTLINE

The remainder of this dissertation is organized as follows:

- Chapter 2 reviews essential concepts used throughout this work. Namely, Use-Case Driven Development, Software Modularity, Separation of Concerns, and Software Maintenance and Evolution.
- Chapter 3 discusses why use case implementations are considered crosscutting concerns, detailing two AOP-based technique used to modularize use case implementations. This chapter also presents how we select the metrics and software quality attributes used in the evaluation of this work. We provide a detailed description of the assessment procedure and the qualified metrics.
- Chapter 4 presents two cases studies where we evaluate the impact on the overall system's modularity of use case implementations. We apply two AOP-based technique to modularize the use case implementations on the target systems and compare with their OO implementation.

- Chapter 5 summarizes the contributions of this research, discusses some related and future work, and presents our conclusions.

CHAPTER 2

BACKGROUND

In this work, we use Aspect-Oriented Programming (AOP) to modularize use case implementations. In addition, we evaluate the impact of such kind of modularization by a means of case studies based on maintenance and evolution tasks. In this context, our work is placed among some research domains, such as Use Case Driven Development, Aspect-Oriented Programming, and Software Modularity, Maintenance and Evolution.

This chapter reviews these essential concepts to help the reader understand our work and evaluate its contributions. In the next Section, we present the concepts behind Use Case Driven Development; an approach to software development where use cases guide the whole process. In Section 2.2, we recall the importance of Modularity to software development and in Section 2.3 we see how Separation of Concerns play an important role to achieve such modular design. In Section 2.4, we briefly explain the concepts of Aspect-Oriented Programming and detail the main points of two aspect language used in this work: AspectJ and CaesarJ. Finally, in Section 2.5, we discuss the main points of Software Maintenance and Evolution.

2.1 USE CASE DRIVEN DEVELOPMENT

Use case driven development is a model driven approach to software development where use cases guide the whole process. It was proposed by Ivar Jacobson [JB00] and according to him, in its simple form it has the following sequence of models: use case, design, and implementation. Within each iteration of the software life cycle the team develop a set of activities listed bellow:

1. Find the use cases and specify each use case.
2. Design each use case.
3. Design and implement each component.
4. Test each use case

Notice that all activities, except the number 3 (designing and implementing each component), are use case based; thus the term use case driven development. During

these activities the following key artifacts are developed: use cases, use case realizations, and components.

Use Cases. Use cases offer a systematic and effective way of representing and communicating requirements of software systems. A use case specifies the behavior of a system and describes sequences of actions, including variants, that a system performs to yield an observable result to an actor [BRJ05]. All use cases together comprise of all the possible ways of using the system. Formally, a use case is a class-like construct that describes a related set of usages of the system by a particular actor (user) type. The use case model contains actors and use cases, and the relationships between them. It is a kind of requirements model. To make the use case model simple, a language constraint has been enforced on the kind of acceptable relationships between use cases. The goal is to be able to separate concerns: each use case represents a concern of a set of stakeholders. Thus, relationships between use cases as class like things are the only acceptable relationships [JB00].

Use Case Realizations. The use-case model is an external perspective of the system; it does not represent the internal building blocks. The internals of the system are introduced in the design model. Each use case in the use-case model is realized by one use-case realization in the design model. A use-case realization is a UML collaboration describing (e.g., using sequence diagrams) which components participate, how they interact, and what responsibilities they take on to realize the use case. Since each use case is a different concern in the use-case model, each use-case realization is a different concern in the design model. The realization of a use case touches many components (scattering), and a component contains pieces of several use-case realizations (tangling) during implementation [JB00].

Components. Each component is specified by collecting all the responsibilities assigned to it over all use cases. It is quite simple since the responsibilities come directly from the use cases, which the component participates in realizing. Now the component developer has to compose and implement all the responsibilities into a consistent whole. There are some issues to deal with: (i) how to reconcile the different needs of the use cases, for instance, reconciling overlapping operations; and (ii) conflicts, for instance those due to concurrency (e.g., deadlock). Finally each component is unit tested against its specification [JB00].

A major problem today is that traditional languages, like Objected-Oriented, Functional, and Imperative do not support separation of crosscutting concerns. Because of

that, the impacts of the different use cases on a component can not be kept separate at the implementation level.

A set of new technologies (detailed below) emerged with the purpose of overcoming the limitation of these traditional languages on handling crosscutting concerns. Therefore, now we have new possibilities to keep use case code separate all the way down to code.

Jacobson [JB00] observed that if we can keep use cases separate during implementation we will be able to develop all the abovementioned activities maintaining the focus on use cases. This way, within each iteration of the software life cycle the team will develop a new sequence of activities highlighted below:

- 1 find the use cases and specify each use.
- 2 design each use case.
- 3 code each use case and compose its code within the respective component.
- 4 test each use case.

In this work, we assume that the development is use case driven. Thus, use cases guide the whole process and are used as units in the assignment of tasks to developers.

2.2 SOFTWARE MODULARITY

One of the key principles of solving complex problems is that dividing a problem into smaller parts enhances understandability and traceability [Pol57]. This is one of the main reasons why systems are modularized. The concept of modularity applied to software development was first introduced by Parnas [Par72]. The goal of modular programming is to break down a program into independent concerns, implement the concerns as separate modules, and compose the modules to produce the final system.

Modularity is closely related to design decisions that decompose and organize the system into a set of modules. The attributes expected in a modular design are outlined in what follows:

- **Comprehensibility:** a modular design allows developers to understand a module looking only at: (i) the implementation of the module itself; and (ii) the interfaces of the other modules referenced by it;
- **Changeability:** a modular design enables local changes. If changes are necessary in the internal implementation of a module A, the other modules that depend exclusively on A's interface will not need to change, since there is no modification in the module interface;

- **Parallel Development:** the specification of module interfaces enables the parallel development of modules. Different teams might only focus in their own modules, reducing the time-to-market and the need of communication.

So, programs constructed via modular programming are easier to understand, develop in parallel, and evolve. The criteria followed to accomplish this decomposition is the key issue to achieve good modularity, which is claimed to provide lower development/maintenance effort and costs [PI87].

2.3 MODULARIZING CROSSCUTTING CONCERNS

The main goal of a system is to satisfy requirements or, more generally, concerns. A typical definition for the term ‘concern’ is *any consideration that can impact the implementation of a program* [Azu01]. For example, a concern can be a functional requirement, a nonfunctional requirement, or a design constraint on the system. In the *ideal world*, we want to break down a program into modules that clearly separate the different kinds of concerns. Although some concerns can be realized by distinct and separate modules, in general, we find many concerns that impact multiple modules. This kind of concerns are known as *crosscutting concerns*.

Separation of concerns is a key point to achieve modularity of a software system. It provides analyzing a specific concern with none or very little knowledge of others. According to this principle, we should decompose a program in such a way that each one of the resulting modules implements one concern. Indeed, a non-modular design, without a clear separation of concerns, leads to several problems. Eaddy et al. [EZS⁺08], for example, recently argued that scattering of concerns cause defects.

Modular Design and Separation of Concerns are so pervasive in software engineering that they have guided the design of processes, techniques, and programming languages. Different technologies have been studied and developed motivated by the need to find better separation of concerns, among which are well-established modularization techniques such as structured programming [Dij69], abstract data types [LZ74], and object-orientation (OO) [PCW84].

Despite all these powerful modularization mechanisms they still provide inadequate separation of crosscutting concerns. Trying to solve this problem, a whole new set of techniques such as composition filters [AWB⁺93], mixins [BC90], hyperslices [OT02], adaptive methods [Lie96], aspects [KLL⁺97], etc, have emerged with a common goal: to enable improved separation of crosscutting concerns. These techniques provide developers with new possibilities for keeping such concerns separate in the source code.

For instance, Aspect-Oriented Programming (AOP) [KLL⁺97] emerged as a means for better modularizing concerns that lead to scattering and tangling when designed using Imperative, Object-Oriented, and Functional programming techniques. AOP has been consolidating as the main technology to modularize crosscutting concerns, in both industry and research environment. Because of that, we choose to explore AOP in the modularization of use cases implementations. In the next Section, we present the main idea behind AOP and also two AOP languages that are used in this work.

2.4 ASPECT-ORIENTED PROGRAMMING

Aspect Oriented Programming(AOP), proposed by Kiczales et al. [KLL⁺97], is a programming paradigm of growing interest. AOP was designed to overcome the limitation of Object Orientation (OO): conventional modularity units, such as classes and services are not able to keep crosscutting concerns separate. AOP provides means to tackle this problem by eliminating two important causes for the loss of modularity: code tangling and code scattering originated by the implementation of crosscutting concerns.

Code tangling happens when there is an introduction of code into a module that does not implement the same requirement for which the module was designed.

Code scattering happens when there is an introduction of code that implements one requirement into several modules. AOP tries to solve these problems by encapsulating the crosscutting concerns into aspects, removing them from the modules they are deforming and, with specific constructs, applying the code again to the places where it should be executed, in compile or run-time.

In this work, we use techniques based on two majors AOP languages: AspectJ [KHH⁺01] and CeasarJ [AGMO06] to modularize use case implementations.

2.4.1 AspectJ

AspectJ is an aspect-oriented extension to Java. Programming with AspectJ involves both aspects and classes to separate concerns. Concepts that are well defined with object-oriented constructs are implemented in classes. Crosscutting concerns are usually separated using units called aspects, which are integrated with the classes through a process called weaving.

Each AspectJ aspect defines a functionality that affects different parts of the system. Aspects may define pointcut designators, advices, and intertype declarations. Pointcut match join points, which are a set of points during program execution flow, where we may want to execute a piece of code. Code to be executed at join points matched by

pointcuts is declared as an advice. Inter-type declarations are structures that allow introducing fields and methods into a class, changing the hierarchy of a type (making a class extend another class or implement an interface), and turning checked exceptions into unchecked exceptions. In what follows we provide a simple language description exemplifying each of the abovementioned language constructs.

Intertype Declarations - with this construct a programmer can add methods, fields, or interfaces to existing classes from a separate aspect.¹ This example adds an `acceptVisitor` method to the `Point` class.

Listing 2.1 AspectJ Intertype Declaration Example

```

1
2 aspect VisitAspect {
3   void Point.acceptVisitor(Visitor v) {
4     v.visit(this);
5   }
6 }
```

Pointcuts - this construct allows a programmer to specify join points — well-defined points in the execution of a program —, such as method call, object instantiation, or variable access. All pointcuts are expressions (quantifications) that determine whether a given join point matches. For example, this pointcut matches the execution of any instance method in an object of type `Point` whose name begins with `set`.

Listing 2.2 AspectJ Pointcut Example

```

1
2 pointcut set() : execution(* set*(..) ) && this(Point);
```

Advice - this construct allows a programmer to specify code that will run at a join point matched by a pointcut. The actions can be performed before, after, or around the specified join point. In the example bellow, the advice refreshes the display every time something on `Point` is set, using the pointcut declared above.

Listing 2.3 AspectJ Advice Example

```

1
2 after () : set() Display.update();
```

AspectJ also supports limited forms of pointcut-based static checking and aspect reuse (by inheritance). In this work, we use AspectJ constructs to implement the Use Case as Aspect technique (detailed in the next Chapter).

¹All AspectJ examples used in this section were gathered from [KHH⁺01]

2.4.2 CaesarJ

In this section, we provide an overview of the CaesarJ mechanisms we use in this work.

CaesarJ is an aspect-oriented language with a strong support for reusability. It combines the aspect-oriented constructs, pointcut and advice, with advanced object-oriented modularization mechanisms such as mixins, virtual inner classes, and wrappers [AGMO06].

CaesarJ shares some important concepts with AspectJ. Like AspectJ, CaesarJ is also an extension to the Java language and uses a joinpoint model similar to the one defined in AspectJ. It uses an AspectJ-like language expression to define pointcuts and advices (as exemplified in the above section). CaesarJ supports dynamic deployment of advice (all advice of a CaesarJ aspect can be activated and deactivated). It also supports deploy on object (the capability to constrain the captured joinpoints to those that originate from a given target object). CaesarJ also provides a second kind of class (in addition to plain Java class), the CaesarJ **cclass** that is roughly equivalent to the aspect modules of AspectJ. In what follows we illustrate the CaesarJ constructs we use in this work.

CaesarJ Classes are represented through the **cclass** keyword in order to separate them from plain Java class. Inheritance hierarchies of Caesar classes and Java classes are strictly separated in CaesarJ. A **cclass** may not inherit from a Java class and vice versa. However, object composition is allowed and interface implementation [AGMO06]. ²

Listing 2.4 CaesarJ cclass Example

```

1 public interface AJavaInterface {}
2 public class APlainJavaClass {
3     ACaesarClass aRefToACaesarClass;
4 }
5 public cclass ACaesarClass implements AJavaInterface {
6     APlainJavaClass aRefToAPlainJavaClass;
7     java.util.Vector aRefToAnotherPlainJavaClass;
8 }

```

Virtual Classes [MMP89] comprise the capability to treat inner, nested classes, polymorphically. In the CaesarJ context, the term **virtual** has the same meaning as in the context of the OO languages, that is, class members that can be overridden in subclasses and whose actual implementation is dynamically bound, or late bound, according to the actual type of the object at runtime. In CaesarJ, any class nested within a **cclass** is a virtual class and thus can be dynamically bound to different concrete classes in different circumstances; it works the same way method calls are late bound to the most specific

²All CaesarJ examples used in this section were gathered from [AGMO06]

implementation in plain Java. Differently from plain Java inner classes, Virtual classes can be redefined in any subclass of the enclosing class [AGMO06].

Listing 2.5 CaesarJ Virtual Class Example

```

1 public cclass Graph {
2     public cclass Edge {
3         Node start, end;
4     }
5     public cclass UEdge extends Edge {...}
6     public cclass Node {...}
7 }
8
9 public cclass WeightedGraph extends Graph {
10     public cclass Edge {
11         float cost;
12     }
13     public cclass Node {
14         float cost;
15     }
16 }

```

In the example above, the classes `WeightedGraph` `Edge` and `Node` are redefined in order to add the `cost` attribute. Virtual classes allow a programmer to incrementally refine a set of collaborating classes. The interesting point is that the references to virtual classes are always bound to the most specific redefinition known in the context of the enclosing object. So, the super-class of `WeightedGraph.UEdge` is bound to `WeightedGraph.Edge`. Moreover, accessing the variables `start` and `end` in the context of `WeightedGraph` allows us to use the newly introduced properties of `WeightedGraph.Node`, although the declaration was made in `Graph.Edge` itself [AGMO06].

Collaboration is a top-level CaesarJ class containing a set of collaborating `cclass`. Each `cclass` containing at least one inner `cclass` is considered as a collaboration. This abstraction usually involves multiple, different classes. In mainstream OO languages, this means that collaborations are a structural manifestation of crosscutting. In CaesarJ, participants in collaborations are declared as inner, nested, virtual classes. Using virtual classes developer can define base class collaboration and incrementally refine it with new features in its sub-collaborations. Independently developed features can be smoothly merged by applying mixin composition on the sub-collaborations [AGMO06].

Listing 2.6 CaesarJ Collaboration Example

```

1 // collaboration Graph
2 public cclass Graph {
3     /*
4      * nested cclasses
5      */
6     public cclass Edge {...}
7     public cclass UEdge extends Edge {...}
8     public cclass Node {...}
9 }

```

Mixin Composition specifies additional functionality to not just one, but to a set of existing modules in a transparent and non-invasive way. Mixins are also known as abstract subclasses [BC90]. In this context, mixins are seen as a mechanism to specify a subclass that can be used to extend more than one class. In CaesarJ, a component can be created by composing several `cclass`. Using mixin composition, different components can be composed to build more complex components without compromising the independence of each component. Mixin composition is the way by which CaesarJ creates complete components out of various parts. Two or more Caesar classes can be composed using the `&`-operator [AGMO06].

Listing 2.7 CaesarJ Mixin Example

```

1 public cclass ColoredGraph extends Graph {
2     public cclass Node {
3         Color color;
4     }
5 }
6
7 public cclass ColoredWeightedGraph extends ColorGraph & WeightedGraph {
8     ...
9 }

```

In the example above, the `&`-operator is used to compose a new collaboration having the properties from `ColorGraph` and `WeightedGraph`. The interesting point here is that the composition mechanism is propagated into the inner classes. All compatible members from both collaborations are also composed together, e.g., after the composition `ColoredWeightedGraph.Node` will have the `cost` and the `color` property.

Beyond the abovementioned constructs explored in this work, CaesarJ also supports: Aspect Deployment which allows an aspect to be instantiated and activated at run-time; and Wrappers Mechanism which is used to perform mappings between components.

2.5 SOFTWARE MAINTENANCE AND EVOLUTION

In this work we run two case studies based on maintenance and evolution tasks. In order to be able to choose a set of changes that best fit our studies, we discuss the importance

of these activities and its impact along the life cycle of a software system.

Software development is the activity of making software changes. According to [Som06], three types of changes are possible:

- Perfective - which corresponds to 50 - 65% of the total.
- Corrective - which corresponds to 17 - 21 % of the total.
- Adaptive - which corresponds to 18 - 25 % of the total.

Perfective change deals with updating the software according to modifications in user requirements. Corrective change deals with fixing bugs in the code. Adaptive change deals with adapting the software to new environments. All changes to the system can be characterized by these three types. Corrective change is “traditional maintenance” while the other types are considered as “software evolution” [?].

Previous research [Ede93, Erl00] shows that the largest costs of software production occur during maintenance, after the development phase is complete, contributing up to 90% of the total. According to Erdil and colleagues [?], maintenance can only happen efficiently if the earlier phases are done properly. There are four major problems that can slow down the maintenance process: unstructured code, maintenance programmers having insufficient knowledge of the system, documentation being absent, out of date, or at best insufficient, and software maintenance having a bad image. The success of the maintenance phase relies on these problems being fixed earlier in the life cycle.

It is well known that a software system will be under constant pressures for changes during its entire life cycle. This is an unavoidable consequence of the nature of software and the changing environment in which it is used [BR00]. One possible approach that contributes to diminish the impact is to design, develop and maintain a system in ways that facilitate and reduce the impact of individual changes. This process is known as change isolation. There are some methods available to support this process that range from code level construction of modules, to business level purchase of commercial off the shelf products. The use of some technique that promotes change isolation can lead to:

- reduction on maintenance costs.
- modular design that is easier to understand.
- better reuse of modules or components.
- extension of the system lifespan.
- prorogation of the system replace.

Software developers and architects should consider some factors to promote change isolation, among them, the change characteristics of the software should be analyzed carefully. For example, which type of change is likely to occur more frequently. With this kind of information on hand, the one can promote a module decomposition that absorbs better the change effects to this specific application.

It is a fact that changes will inevitably arise during the software life cycle, some common causes are: political decisions, hardware related changes, operating system upgrades over time and competition/evolution (where new features are added to satisfy customer needs). So, maintenance cost is not only a result of poor design, but also a result of the changing user and environment requirements. With the changeability of software, even if a system was developed “right at first time”, it would require modification as it would need to change after its first use. The system is almost instantly dealing with outdated requirements [BR00]. Because of that, it is essential to prepare a software for maintenance prior to delivery. There are good methods and bad methods that can be used to build systems if change isolation is a design goal. Choosing the right approach is essential to lower the impact of changes [BR00].

So, in this work we apply aspect-based modularity techniques to isolate the use case implementations into their own module; this is a way to promote change isolation. An important requirement in terms of system maintenance and evolution is the capability to think about a system in terms of what parts are more suitable to change. If a system is constantly affected by changes on its functional requirements, modularizing use cases code can be essential to diminish maintenance costs. One of our goals is to be able to quantify how modularized use case implementations impact the overall system’s modularity. Changeability plays an important role in this process as an indicator of how a system behaves when a set of maintenance and evolution tasks are applied to it.

CHAPTER 3

MODULARITY OF USE CASE IMPLEMENTATIONS

In this chapter we detail the problem we want to address. In Section 3.1 we show why use case implementations are fundamentally a crosscutting concern. Then, in Section 3.2 we present two AOP-based techniques used to modularize use case implementations. Finally, we discuss the metrics used to analyze and evaluate the techniques regarding modularity improvement.

3.1 USE CASE IMPLEMENTATIONS AS CROSSCUTTING CONCERNS

Today, component-based architectures is the most used technique to develop complex softwares. It relies on a process where the main goal is the assignment of requirements to components. Traditional modularity techniques, like Object Orientation, are applied to allow the system's decomposition into well-defined components. Components are used as modularity units that have specific purpose and responsibilities. The set of components in a system represents its static structure as depicted in Figure 3.1.

Figure 3.1 Library Management System represented by interconnected components.

Figure 3.1 shows the components for a Library System. We use this toy system as an example here and throughout the rest of this dissertation. We explain this system in detail in Section 4.1, as it is also used as a case study to consolidate the techniques, metrics and the overall evaluation process adopted in this work. Briefly, this system provides the functionalities to add, remove, check in, and check out books to be used by both members and library staff. It shows components of diverse kinds. The **Staff Screen** and

Member Screen deal with presenting information to users and accepting and validating input from them. The **Add Book**, **Check In**, **Check Out** components encapsulate the business and control logic for the respective functionalities. The **Book**, **Book Copy**, and **User** components maintain information in a data store.

We use the Layer [BMR⁺08] architectural pattern to structure this system in layers that group together components of same general purpose, as the ones presented above (user interface, business logic, and data access). This separation of roles and responsibilities across components is essential to deliver a system that can easily be adapted when changes are introduced.

Components are crucial to design, implement, test and to understand the system. In fact, components are a key factor to promote reuse: they encapsulate requirements that are more likely to change together and they can keep some concerns separate. Ideally, we want to be able to separate all kinds of concerns into components, and develop and maintain each one independently. But, when we use Object Orientation (or any other traditional modularity technique) to decompose a system, we find that although some concerns can be realized by distinct and separate components, in general, we have many concerns that impact multiple components. As discussed in Section 2.3, they are known as crosscutting concerns.

There are different kinds of crosscutting concerns. Infrastructure concerns like logging, persistence, and distribution are classical examples of such concerns. They are typically used to address non-functional requirements. However, a crosscutting concern can deal with functional requirement as well. We find that the implementation of functional requirements, which can be specified as use cases, is fundamentally a crosscutting concern. That happens because traditional modularity techniques [Dij69, LZ74, PCW84] decompose use case implementations across components. This is illustrated in Figure 3.2

Figure 3.2 shows use cases in different grey scale colors on the left. On the right, we have components with multiple colors. Each color represents the code that implement the respective use case. Notice the limitation of components to keep separate use case implementations; as such they become scattered and tangled over the codebase. Tangling and scattering are the two main causes of modularity loss resulted by the implementation of crosscutting concerns.

Tangling happens when each component contains the implementation (code) to satisfy distinct use cases. For example, in Figure 3.2, we see that the **Book** component is involved with the realization of three different use cases. The component instead of being dedicated to a particular use case implementation, participates in many. This can make the code hard to understand as the developer has to be aware of code that is not specific to the use case he is dealing with.

Figure 3.2 Tangling and scattering when realizing use cases.

Scattering happens when code that realize a particular use case is spread across multiple components. For example, in Figure 3.2, we see that the realization of **Check In Book** enforces additional code on five components. So, anytime the requirements associated with that use case change, we must update many components.

It is well known that crosscutting concerns reduce modularity, making programs harder to understand, develop in parallel, maintain, and evolve [EVS⁺08, KLL⁺02]. All these problems are also present in the context of use case implementations.

As seen in Section 2.1, in a use case driven development use cases guide the whole development process. Use cases start at requirements, are translated to realizations during analyzes and design and to test cases in testing. As illustrated in Figure 3.3, *each* use case in the use case model is translated to *one* use case realization in the design model. But, during implementation, one use case realization can not be mapped to just one component. At this point, a use case unit cannot be seen any more. The 1:1 mapping from use cases to assets we had in the previous phases of the development process is over. Use case implementations impose a N:M relation between use cases code and components causing traceability loss. For instance, it is not easy to uncover requirements by reading the source code of each component or a set of components. This harms understandability hindering the learning curve for developers. Poor traceability leads to poor understandability which leads to poor maintainability.

As discussed in Section 2.5, in a typical software environment, changes are requested even while the software is still being developed. Thus, a simple addition or modification on a use case specification leads to a very invasive process of change, where a developer has to deal with many source files, classes, methods, etc. Such situation increases maintenance

Figure 3.3 Use cases to components, based on [JN04]

and evolution costs.

Pluggability is another important issue to consider. In this situation, where a use case implementation cut across multiple components, it is not easy to turn on/off a specific use case in the system. Being able to plug or unplug a use case with minimum effort is an essential capability for some particular contexts, for example, in a software product line [VdLSR07, PBvdL05].

So far, we find that one of the most used modularity techniques, Object Orientation, can help us to keep concerns separate into components, but only to a limited extent. A class can keep the detail of an object separate; a component can hide the data and computation related to a specific functionality. However, these mechanisms are not sufficient when dealing with crosscutting concerns. Therefore, we demand another approach to modularity, for example, AOP.

This work focus on this specific kind of crosscutting concern: *functional use case implementations*, that is, how functional requirements specified as use cases are realized into code. It is well known that AOP is effective to modularize crosscutting concerns, such as, persistence, logging and transaction. But to the best of our knowledge it is not yet clear how AOP behaves for modularizing use case implementations. We find some works [HHM04, Jac03, BL08] showing how AOP can be used to enable such modularization; they also report some benefits, such as, improvement on parallel development, traceability, and changeability. However, theirs observations rely only on a poor qualitatively evaluation. We were not able to uncover empirical evidences of such effects. In fact, a quantitatively evaluation is missing in order to provide a fair analysis regarding the real modularity gain when we modularize use case implementations.

So, we run two case studies aiming at comparing and evaluating use cases implemented using AOP-based techniques (with AspectJ and CaesarJ) and use cases implemented in the traditional object-oriented way (with Java). We analyze the impact on some traditional and recent internal quality metrics, including cohesion, coupling and separation of concerns and report to which extent this kind of modularization reduces or eliminates the abovementioned problems analyzing its contribution (positive or negative) to the system's modularity. We detail the techniques and metrics in the following sections.

3.2 TECHNIQUES

As stated in the previous section, we use Aspect-Oriented Programming (AOP) to modularize use case implementations, which is a specific kind of crosscutting concern. AOP tackles the problem by providing a mechanism to compose additional behavior from a separate unit (aspect) into a class. After that, a weaving process composes the units; this composition can occur during compilation or runtime. With this mechanism we can modularize the implementation of different use cases into separate modules.

We choose AOP because it is the most used technology to achieve separation of crosscutting concerns by the academia and the industry. An increasing number of research papers [GBF⁺, CFCF⁺06, GSC⁺04, GSF⁺06, KSG⁺06] have been published in the current decade evaluating and comparing the implementation of crosscutting concerns in AO and OO, which is also the final goal of this dissertation. To this end, we apply techniques, based on AspectJ [KHH⁺01] and CaesarJ [AGMO06] (two major aspect languages) to separate the implementations of use cases in source code.

The techniques were selected in order to test different AOP mechanisms that match with the way use cases are implemented. While working with AspectJ we use aspects (intertypes declaration, pointcuts, and advices), with CaesarJ we explore the collaborative design approach (collaborations, virtual classes, and mixins). The techniques are not tied to any specific aspect language but with the constructs offered by them. Therefore, one can use other AOP language to implement a technique since it provides the same crosscutting mechanism used in the technique design. We use the example defined below to illustrate the implementation details of each technique.

Let's consider three specific use cases in the Library Management System: **Add Book**, **Check Out Book** and **Check In Book**. Briefly, the functionalities are as follows:

- To add a book, we create the book and one respective book copy.
- To check out a book, we check if the book is available, that is, if the book has at least one book copy available. If the book is available, we assign it to the user

(member).

- To check in a book, we find the book record and collect the information about the book copy assigned to the user. We set the book copy as available and remove the book copy record from the user.

	Book		Book Copy		User	
Add Book	create();		create();			
Check In Book	find();		getHolder(); setAvailable();		removeBookCopy();	
Check Out Book	checkAvailability();		setUnavailable();		assignBookCopy();	

Figure 3.4 Classes composed from different use cases code, based on [JN04]

Figure 3.4 shows how these functionalities, represented horizontally, cut across classes in the system. Each class lists the operations required by the specific use case. It shows that various functionalities impose different operations on the respective classes, characterizing the crosscutting nature of use case implementations.

With this scenario in mind, we explain next how each approach uses AOP concepts to modularize use case implementations.

3.2.1 Use Case as Aspect - with AspectJ

Use Case as Aspect is based on Jacobson work [Jac03]. It aims at “slicing” a system by use cases. Each slice represents a new modularization unit (use case module) containing classes that are specific to the implementation of a use case and aspects containing extensions to classes that are shared between components of the system.

Applying this approach consists on dividing a system into two separate parts: the core components and the set of use case modules. The core corresponds to the element structure of the system. It identifies the elements defining components and the system’s architecture. The element structure determines where all components reside and also the relationships between them. The set of use case modules corresponds to the implementation of the functional use cases of the system.

This is illustrated in Figure 3.5. On the left we see the element structure organized hierarchically in terms of layers, packages and classes. On the right, we see the use case module. To simplify, we only show one use case. The use case module contains specific elements and element extensions that will be composed (in our case, through AspectJ constructs) with the element structure.

Figure 3.5 Element Structure and Use Case Module

Let's consider the Library System example to illustrate how use case implementations are separated and then composed together into the whole system.

In Figure 3.6 the horizontal axis shows the elements structure that identifies the classes in the system. The vertical axis shows the use case structure. It identifies the use cases being realized, each with a different color. Each horizontal row depicts a use case module containing the extensions of classes needed to realize the use case for that row. Thus we have the **AddBook** use case module, the **CheckInBook** use case module and the **CheckOutBook** use case module.

Figure 3.6 Use case implementations with use case modules, based on [JN04]

Each use case module contains partial class definitions (class extensions) specific to the use case implementation. We use AspectJ constructs to merge together the extensions to a class and the result is depicted in the bottom row of figure 3.6. The resulting Book

class is a composition of all class extensions from the respective aspects as illustrated by the different colors.

From these observations we find that while implementing functional use cases we deal with two situations: the need to define a complete new class that is dedicated to the realization of the use case; and the need to add behavior to classes shared by others use cases. In the first situation, we simply define the new class using the system base language, in our case Java. In the second, we use aspects, in our case AspectJ constructs as follows:

- **Intertype declaration** - used to compose new features (attributes, operations and relationships) into classes defined in the system's core.
- **Advices** - used to extend existing functionalities at extension points assigned by pointcuts into classes defined in the system's core.
- **Aspects** - used to group and organize intertypes declaration and advices.

Listing 3.1 shows an example of each AspectJ construct.

Listing 3.1 AspectJ constructs examples

```

1 public privileged aspect BookCheckOut {
2     ...
3     //intertype declaration adding the status attribute to the BookCopy class
4     private boolean BookCopy.status;
5
6     //intertype declaration adding the setStatus(..)
7     //operation to the BookCopy class
8     public void BookCopy.setStatus(boolean sta) {...}
9
10    //extension point where additional behavior is added at run time
11    pointcut createBookCopy(BookCopy bc): execution(BookCopy.new(..)) && this(bc);
12
13    //advice containing extra behavior to the operation designated by the
14    //createBookCopy pointcut
15    after(BookCopy bc): createBookCopy(bc) { bc.status=true; }
16    ...
17 }
```

For each functional use case in the system we define a use case module. Thus for the **Check Out Book** use case, there is a corresponding **Check Out Book** use case module. The **Check Out Book** contains the following:

1. **Specific Classes.** The **CheckOutBook** class is specific to this use case realization and is not needed by any other use case in the system.
2. **Specific Extensions.** The **Book**, **BookCopy** and **User** classes are needed by several use case realizations. However, **checkAvailability()**, **setUnavailable()**

and `assignBookCopy()`, are specific operations to the **Check Out Book** use case realization. This is defined within a class extension in the use case module.

Figure 3.7 Check Out Book use case module representation

Figure 3.7 depicts the **Check Out Book** use case module. At implementation, each module is represented by a package with the same name as its corresponding use case. The **Check Out Book** use case module contains classes `CheckOutBookHandler` and `CheckOutBookForm` specific to its realization and an aspect `CheckOutBook` containing classes extensions that are also specific to its realization. This extensions are achieved using intertype declarations and advices. This way, we can keep the specific of each use case separate.

Listing 3.2 shows the simplified code that implements the **Check Out Book** use case as an aspect.

Listing 3.2 Code for aspect `CheckOutBook.aj`

```

1
2 public privileged aspect BookCheckOut {
3     ...
4     public BookCopy Book.checkAvailability() {
5         // code to check if a book is available
6     }
7     public void BookCopy.setUnavailable() {
8         //code to make a book copy unavailable
9     }
10    public void User.assignBook(BookCopy bc) {
11        // code to assign a book to a user
12    }
13    ...
14 }
```

An import issue to consider is how to deal with use case interactions. As shown, a use case module contains elements that are specific to the implementation of a particular use case. However, sometimes we find elements that are required by more than one use case. For example, the `find()` operation in the `Book` class is used by the **Check In Book** and **Check Out Book** use cases. So, we choose to factor out this operation to its own module and create the **Find Book** (non use case) module. This module is used by both use cases; a trace link should be created in the design model to highlights this dependency relation. Another possibility is to extract the shared elements to the system's core. This is an important design decision that should be taken carefully. In order to apply a consistent treatment, we decide to factor out to the system's core, elements shared by at least three use cases.

Finally, the set of all use case implementations and the core of the system are composed together leading to a system where the primary decomposition are use cases implementations.

3.2.2 Use Case as Pluggable Collaboration - with CaesarJ

Use Case as Pluggable Collaboration aims at modularizing a use case implementation through an hierarchical composition of collaborations. A *collaboration* is a new modularization mechanism present in some extended object-oriented languages, like CaesarJ [AGMO06] and ObjectTeams [Her07] with the purpose of grouping multiple classes and refinements into an enclosing module. This concept allows implementing crosscutting concerns separate from each other. In order to compose the whole system, the collaborations have to be bound (plugged) to it.

A use case implementation is a crosscutting concern that is often implemented by multiple collaborating objects. Therefore, the concept of collaboration matches perfectly with the nature of use cases. A collaboration offers modules that can encapsulate the multiple interdependent types that participate in a use case realization. In this work, we choose the CaesarJ language to provide the implementation of use cases as pluggable collaborations.

The implementation steps used here are quite similar to the ones used in the Use Case as Aspect approach. We also split a system in two main parts: the element structure (core) and the set of use case collaborations. The element structure corresponds to an hierarchy of collaborations representing the layers in the system architecture. The set of use case collaborations corresponds to the implementation of the functional use cases of the system. For each functional use case, we implement one use case collaboration. The resulting system is the composition of all use cases collaboration.

This is illustrated in Figure 3.8. On the left we see the element structure organized hierarchically in terms of layers. Each layer represents a collaboration that extends the functionalities of the layer immediately below. On the right, we see the use case collaboration. To simplify, we only show one use case. Each use case collaboration extends the upper most layer in the element structure. It adds classes that are specific to the implementation of the use case and also refines classes (adding extra behavior) defined in the chain of the extended collaborations. As shown in the Figure, the “super” collaboration contains the classes defined at the extended collaboration and their own defined classes. Different shapes indicate elements defined by different collaborations. Shadows indicate that extra behavior is added by refining some “inherited” class.

Figure 3.8 Element Structure and Use Case Collaborations

Allowing the implementation of use cases as pluggable collaborations, we use the following CaesarJ constructs:

- **Caesar Class** - We define Caesar classes using the `cclass` keyword in order to separate them from plain Java class.

Listing 3.3 Caesar Class

```

1 public cclass Book {
2     private String isbn; ...
3     public String getISBN() {
4         return isbn;
5     } ...
6 }
```

- **Collaboration** - We use collaborations to group a set of collaborating classes. Each `cclass` having at least one inner `cclass` is considered as a collaboration.

Listing 3.4 Collaboration LibraryDomain

```

1 public cclass LibraryDomain {
2
3     public cclass Book {...}
4     public cclass BookCopy {...}
5     public cclass User {...}
6     ...
7 }

```

- **Virtual Classes** - All inner classes within a `cclass` are considered as virtual classes, because they are handled in a similar way as virtual methods (see Section 2.4.2). We can redefine virtual classes in any subclass of the enclosing class.

Listing 3.5 LibraryDomain

```

1 public cclass LibraryDomain {
2
3     public cclass Book {
4         private String isbn;
5     }
6     public cclass BookCopy {...}
7     public cclass User {...}
8     ...
9 }

```

Listing 3.6 LibraryApp

```

1 public cclass LibraryApp extends
2     LibraryDomain {
3     public cclass Book {
4         private Map bookCopy;
5     }
6     public cclass Operation {...}
7     public cclass Exit {...}
8     ...
9 }

```

As illustrated above, we define the `LibraryApp` collaboration as an extension of `LibraryDomain`. We redefine the class `Book` in order to add the `bookCopy` attribute. Virtual classes allow us to incrementally refine a set of collaborating classes. All compatible members from both collaborations are composed together, for example, after the composition `LibraryApp.Book` will have the `isbn` and the `bookCopy` property.

- **Mixin Composition** - Using the `&`-operator, we can compose two or more Caesar classes as mixins (see Section 2.4.2).

Listing 3.7 LibrarySystem composed by use case collaborations

```

1 public cclass LibrarySystem extends
2     AddBook &
3     BookCheckOut &
4     BookCheckIn {
5     ...
6 }

```

In the example above, we use the `&`-operator to compose a new collaboration having the properties from `AddBook`, `BookCheckOut` and `BookCheckIn`. The key point showed here is that the composition mechanism is propagated into the inner classes and all compatible members are composed together as showed in Listing 3.6.

For each functional use case in the system, we define a new pluggable collaboration. Thus for the **Check Out Book** use case, there is a corresponding **Check Out Book** collaboration. Figure 3.9 illustrates its main elements.

Figure 3.9 Representation of the Check Out Book use case collaboration

1. **Definition of Specific Classes.** The **CheckOutBook** is specific to this use case realization and is completely defined in the use case collaboration.
2. **Classes Refinements.** The **Book**, **BookCopy** and **User** classes are needed by several use case realizations. Because of that, they are defined at the **LibraryDomain** collaboration. However, **checkAvailability()**, **setUnavailable()** and **assignBookCopy()**, are specific operations to the **Check Out Book** use case realization. These operations are defined in the **CheckOutBook** collaboration, by refining the respective **cclass**.

Listing 3.8 shows the simplified code for the **CheckOutBook** use case collaboration.

Listing 3.8 CheckOutBook Collaboration

```

1 public cclass UcCheckOutBook extends LibraryApp {
2
3     public cclass CheckOutBook {...}
4     public cclass Book {
5         public BookCopy checkAvailabillity() {...}
6     }
7     public cclass BookCopy {
8         public boolean setUnavailable() {...}
9     }
10    public cclass User{
11        public void assignBookCopy() {...}
12    }
13    ...
14 }
```

The set of all use case collaborations are composed together as mixins to provide the whole system implementation as shown in Listing 3.7.

So, in this section we showed the techniques used to modularize use case implementations. Despite using different crosscutting mechanism (aspects and collaborations) the approaches have similar design. A use case module corresponds to a use case collaboration. Implementing class extensions with intertypes declarations has the same effect as implementing the extensions by refining virtual classes. The main difference resides on the composition mechanism. The Use Case as Aspect technique composes use case modules including them in the compiler build path. The Use Case as Collaboration technique uses the Mixin mechanism (semantically defined) in order to compose the use case collaborations that constitute the final system.

3.3 METRICS

We apply the techniques discussed in the previous section to modularize use case implementations in two case studies. The studies are detailed in the next chapter. In this section, we discuss the selection of software metrics aiming to quantitatively evaluate each technique.

The main goal of our evaluation process is to answer how a system behaves regarding modularity improvement when modularized by use case implementations. To this end, we analyze the impact on modularity through a set of software quality attributes that are affected directly by the system's modular implementation, as illustrated in Figure 3.10.

For each attribute, we need to define a set of metrics that enables its quantitative evaluation. Based on that, we setup a Goal-Question-Metric (GQM) model [BCR94] to drive the selection process. Using such approach, we can choose meaningful metrics correlating them with the software quality attributes.

Figure 3.10 Software Quality Attributes

The GQM has been frequently adopted in the selection of metrics. According to Basili et al. [BCR94], the measurement model is divided in three levels as listed below:

- Conceptual level (Goal) - A goal is an expectation regarding an object (product, process, resource), from a point of view , with respect to a model of quality, relative to a particular environment.
- Operational level (Question) - A set of questions is used to characterize the way the assessment/achievement of a specific goal is going to be performed. Questions try to characterize the object of measurement with respect to a selected quality.
- Quantitative level (Metric): A set of data is associated with every question in order to answer it in a quantitative way.

In short, this approach consists on defining metrics in a top-down manner following the steps bellow:

1. Identifying the goals to be met.
2. Elaborating the questions which allow to understand if the goals are met.
3. Identifying the metrics required to answer the questions.

The correct use of such approach can ensure that the identified metrics are relevant to understand if the goals are being met, also providing traceability between goals and metrics. Table 3.1 shows the GQM model we develop to define the metric suite used in the evaluation process of this work.

Table 3.1 GQM to assess modularity improvement.

Goal: Increase system's modularity	
Purpose	Increase
Issue	Modularity of use case implementations
Object	System
Viewpoint	Software Engineer
Questions and Metrics	
Q1 - What is the potential crosscutting level of use case implementations?	
M1.1 - CDC	Concern diffusion over components
M1.2 - DOSC	Degree of scattering over classes
M1.3 - ADOSC	Average degree of scattering
M1.4 - DOT	Degree of tangling
M1.5 - ADOT	Average degree of tangling
Q2 - How modularized use case implementations impact the complexity of source code?	
M2.1 - CBO	Coupling between modules
M2.2 - LCO	Lack of cohesion over operations
M2.3 - VS	Vocabulary size
M2.4 - DIT	Depth of inheritance tree
M2.5 - LOC	Lines of Code
Q3 - What is the effort to modify use case implementations? (Changeability issue)	
M3.1 - NAC	Number of added components
M3.2 - NCC	Number of changed components
M3.3 - NALOC	Number of added lines of code (LOC)
M3.4 - NCLOC	Number of changed lines of code (LOC)
Q4 - What is the effort to trace changes in use case specifications from requirements to implementation? (Traceability issue)	
M1.1 - CDC	Concern diffusion over components
M4.1 - NSFC	Number of source files related to the change
Q5 - What is the the effort to plug/unplug a use case in the system? (Pluggability issue)	
M3.2 - NCC	Number of changed components to enable/disable a use case
M5.1 - NBDLOC	Number of blocks with dead LOC
M5.1 - NDLOC	Number of dead LOC
Q6 - What is the impact on parallel development caused by modularizing use case implementations?	
M6.1 - CI	Core Influence - System's core LOC / Total LOC use cases
M6.2 - NAC	Number of adjustments in the Core

3.3.1 Metrics Description

In this section, we detail the selected metrics. We describe the purpose and discuss how each metric impacts modularity. Most metrics we choose have already been successfully used in several case studies [GBF⁺, CFCF⁺06, GSC⁺04, GSF⁺06, KSG⁺06]. Some of them are refinements of classical metrics used in the OO context [CKM94] adapted to be used in the AO context [CT04]. Others are newly defined, as the ones for evaluating separation of concerns [GSF⁺06, EAM07].

3.3.1.1 Separation of Concerns

- **Concern diffusion over components (CDC)** - Number of classes associated with a concern [GSF⁺06].
- **Degree of Scattering across Classes (DOSC)** - Degree to which the concern code is distributed across classes. Varies from 0 to 1. When DOSC is 0 all the code is in one class. When DOSC is 1 the code is equally divided among all the classes. Differently from CDC, DOSC not only considers which elements are involved in the implementation of a concern, but also how the code is distributed among those elements. [EAM07]
- **Average degree of scattering (ADOS)** - Is the average (arithmetic mean) of the degree of scattering across all concerns. ADOS gives an indication of the overall modularity of the program. [EAM07]
- **Degree of tangling (DOT)** - Degree to which a program element (e.g., component, classes, aspect) is dedicated to one or more concerns of the program. It is very similar to DOSC except that DOT is with respect to program elements instead of concerns. [EAM07]
- **Average degree of tangling (ADOT)** - Is the average (arithmetic mean) of the degree of tangling across all program elements [EAM07]

The main property that defines a module is that its implementation is localized, so a concern that is scattered is by definition not modular [EAM07]. Furthermore, the components that contain code of a scattered concern are less modular than if the scattered concern did not exist. From these observations, we have the average degree of scattering (ADOS) as an indicator of the overall modularity of the system. Likewise, the average degree of tangling (ADOT) is an indication for how well concerns are separated in the

system. Ideally, a system should have a low average degree of scattering (ADOS) and a high average degree of tangling (ADOT) [EAM07].

Metrics that measure crosscutting are dependent on the concerns and on the component granularity level [EAM07]. Therefore, we have to discuss two important issues to tackle these dependences: (i) concern selection and (ii) concern assignment.

Concern selection is the task responsible for choosing the *right* set of concerns to assure meaningful results. Besides the concern metrics definition, Eaddy and colleagues [EVS⁺08] also provide guidance on how to select concerns. They argue that concerns should provide the rationale for implementation. For example, *persistence* should not be selected as a concern if the developer did not consider it. This way, they indicate that requirement documents should be the start point to this task. They also emphasized the importance of selecting a set of concerns that cover most of the code. This reduces sample bias since all concerns are considered, not just those that are crosscutting and in the focus of analysis [EVS⁺08].

Concern assignment is the task responsible for determining the relationship between a concern and a program element. We use the prune dependency rule created by Eaddy et al. [EAM07] to decide which elements are responsible for implementing a concern. In short, this rule states the following:

A program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned.

Besides using this rule we also determine that *classes* are used as the component granularity level. This way, we tie the scope of the assignment task contributing to diminish the impact of human error.

3.3.1.2 Complexity

- **Coupling between components (CBC)** - Counts the number of modules or interfaces declaring methods or fields that are possibly called or accessed by a given module. It is an equivalent of the CBO metric from CK metrics suite [CKM94].
- **Lack of cohesion over operations (LCO)** - Counts the number of pairs of operations working on different class fields minus pairs of operations working on common fields (zero if negative) [CT04]. When all methods do not access any field, then $LCO = 0$ [CKM94]. LCO is associated with the pairwise dissimilarity between different operations belonging to the same module. Operations working on separate subsets of the module fields are considered dissimilar and contribute to the increase

of the metric's value. LCO will be low if all operations in a class or an aspect share a common data structure being manipulated or accessed [CT04].

- **Vocabulary size (VS)** - Counts the number of implementation assets (classes and aspects).
- **Depth of inheritance tree (DIT)** - Counts the length of the longest path from a given module to the class/aspect hierarchy root [CT04].
- **Lines of code (LOC)** - Counts the number of lines. We count each line based on the definition implemented on the AOP metric tool: (i) every statement is in a single line, (ii) class, aspect, method and advice headers are counted as a one line, (iii) imports, comments, javadocs and empty newlines are not counted, (iv) newline is not created for a opening curly braces, (v) newline is created for a closing curly braces, and (vi) string constants are counted as a single line (even when it contains new lines) [aop].

We want to have as many decoupled components as possible. Excessive coupling between components harms modular design and decrease the possibility of reuse. Higher CBM values imply in a system more sensitive to changes as they are propagated to other modules, which makes more difficult the changing process.

We want to have cohesive methods inside a class, since it promotes encapsulation. Lack of cohesion, usually, indicates that classes should be analysed carefully as candidates to be split into two or more subclasses. Low cohesion increases complexity, thus increasing the chances of errors during the development process.

We want to pursue a shallow inheritance tree. Higher DIT values imply higher complexity. A deep class/aspect in the hierarchy probably inherits a great number of operations, thus making it more difficult to understand and change. Deeper trees indicate higher design complexity, since more operations and modules are involved.

Higher VS values imply in a verbose system which, generally, leads to higher complexity.

3.3.1.3 Changeability

- **Number of added components (NAC)** - Counts the number of components added to satisfy a change request in the program.
- **Number of changed components (NCC)** - Counts the number of components modified to satisfy a change request in the program.

- **Number of added lines of code (NALOC)** - Counts the number of lines of code added to satisfy a change request in the program.
- **Number of changed lines of code (NCLOC)**- Counts the number of lines of code modified to satisfy a change request in the program.

Higher values of changeability metrics indicate higher complexity in the changing process. Ideally, we want to have a localized change, where modifications are confined at the module responsible for implementing the changed requirement. Changes that affect multiple modules are error prone and increase maintenance and evolution costs. Therefore, lower values are desired for all changeability metrics.

3.3.1.4 Traceability

- **Concern diffusion over components (CDC)** - Counts the number of components that implements a concern.
- **Number of source files (NSFC)** - Counts the number of source files modified to satisfy a change request in the system.

Lower values of traceability metrics are also expected. They assure that a requirement is well localized and provide easier ways to create and maintain traceability links from requirements to source code.

3.3.1.5 Pluggability

- **Number of changed components (NCC)** - Counts the number of changed components to enable or disable a use case.
- **Number of blocks with dead LOC (NBDLOC)** - Counts the number of blocks (continuing lines) that needs to be commented on removing all code related to a use case from the binary code of the program.
- **Number of dead LOC (NDLOC)** - Counts the number of lines of code that needs to be commented on removing all code related to a use case from the binary code of the program.

Lower values of pluggability metrics is an indicator of less effort to enable or disable a specific use case in the system.

3.3.1.6 Parallel Development

- **Core Influence (CI)** - Measures the influence of the amount of code defined in the system's core.
- **Number of adjustments in the Core (NAC)** - Counts the number of adjustments needed to be done in the core when implementing use cases code.

Core influence is the relation between the lines of code in the system's core and the total lines of code in the system. Higher CI values imply that the system has a great amount of code shared among various use cases, thus, introducing extra management effort and complexity to parallel development.

NAC brings an indicator of how well the system's core was designed in order to provide the necessary code shared by use cases. Each time the implementation of a specific use case updates the core, it can introduce side effects to others. This kind of situation also harms parallel development.

3.3.2 Metrics Computation

Assuring that our results can be replicated, we need to define the steps used to assess each of the metrics described in the previous section.

In the computation of metrics addressing complexity of source code (CBO, LCO, DIT, LOC) we use the *AOP metrics* tool [aop]. It is a common metrics suite tool for Java and AspectJ. It implements the metrics based on the Ceccato and Tonnela [CT04] metrics suite (which adapts the CK metrics [CKM94] to contemplate the effects of the software *aspectization*). We were not able to find any tool that defines and computes these metrics for CaesarJ programs, because of that, our analysis is limited to a qualitative approach regarding the CaesarJ releases.

The computation of Separation of Concerns metrics requires more effort. As shown, It depends on two manual tasks: concern selection and concern assignment. There is no tools available in order to enhance this process. The ConcernTagger, an Eclipse plugin, developed by Eaddy [con] helps in the concern assignment task and calculates the CDC, DOSC and DOT metrics. However, it only works on pure java programs which prevents its use in our context.

In order to provide a more consistent way to calculate these metrics we use these steps: (i) we follow the concern selection and assignment directives; (ii) with the collected data, we create a concern model xml file; (iii) we compute the metrics.

Listing 3.9 shows a simplified example of this file. It contains the definition of all selected concerns. Each concern includes a list of components associated with its im-

plementation. Each component contains the number of statements (loc) responsible for implementing the concern. With this information, we are able to calculate all concern metrics used in this work. Besides documenting the process of concern selection and assignment, we use the concern model file as input to a simple tool (developed in java) that reads its content and computes the metrics (CDC, DOSC, DOT, ADOSC and ADOT).

Listing 3.9 Concern model file

```

1 <concern_model>
2   <concern name="AddBook">
3     <component name="Book" loc="12"/>
4     <component name="BookCopy" loc="6"/>
5     ...
6   </concern>
7   <concern name="CheckInBook">
8     <component name="Book" loc="12"/>
9     <component name="User" loc="9"/>
10    ...
11  </concern>
12  ...
13 </concern_model>

```

Metrics like NCC (number of changed components), NAC (number of added components), NALOC (number of added lines of code), and NDLOC (number of dead line of code) are computed by processing the resulting xml file from the AOP metrics tool. The calculation is based on the difference in the number of program elements and on the difference in the number of lines of code between consecutive releases (in this work, releases are just delivered based on change requests). To assure a correct computation, we need to guarantee that differences on two consecutive releases are produced **only** by code added or modified in order to provide the implementation of the specific change request.

The CI (core influence) metric is calculated by processing both xml files (concern model and AOP metrics output). We use the concern model xml file to gather all components participating in the system's core implementation and the AOP metrics xml file to find the LOC value for each of these components.

The remaining metrics NCLOC (number of changed line of code), NSFC (number of source file), NBDLOC (number of blocks with dead line of code), and NAC (number of adjustment in the core) are calculated manually by analyzing the source code after each change request.

CHAPTER 4

CASE STUDIES

As previously discussed, the main goal of aspect-oriented programming (AOP) is to promote better modularity through new modularization mechanisms for modularizing crosscutting concerns. In fact, while AOP has been consolidating as the main technology to modularize crosscutting concerns, in both industry and research fields, the understanding of its impact on key modularity related software attributes is still a challenge to practitioners and researchers.

Some case studies in the literature analyze how AOP promotes superior modularity and separation of concerns in the implementation of traditional crosscutting concerns, such as, distribution [SLB02, SBL06], persistence [RC03, SBL06], and exception handling [CFCF⁺06]. Others, analyze the scalability of AOP in the presence of broader design changes [GBF⁺, KSG⁺06]. However, none of them addresses use case implementations as the main focus of analysis. Moreover, use case implementations are not considered as crosscutting concerns and are just used as a factor of change, that is, they only report how an introduction or modification of a use case impacts *commons* crosscutting concerns (distribution, persistence, exception handling, and etc).

In order to fill this gap and enhance the body of knowledge on the use of AOP, this chapter presents two systematics case studies in which we evaluate the impact of aspect-oriented (AO) and object-oriented (OO) use case implementations on the system's modularity. We apply two AOP-based techniques, discussed in Section 3.2, to modularize use case implementations and compare with their OO implementation. Our investigation complements the existing empirical studies, since our evaluation process quantifies: (i) the effects of AOP on the achievement of separation of concerns and others equally important software quality attributes, such as, complexity, changeability, traceability, pluggability, and support for parallel development; (ii) and the maintainability of both AO and OO solutions while implementing a set of change requests (with different types and scope) on the target system.

The studies are based on a typical evolution and maintenance environment. From a base implementation, we evolve the target system through a set of change requests. We select changes from different natures (type and scope) that are commonly applied to a system under maintenance and evolution. Regarding their scope, we classify changes

as use case-scoped and system-scoped. Use case-scoped are changes that affect directly use case specifications: adding, removing and modifying a particular functionality on the system. This type of change aims to evaluate how use case implementations behave when changes are derived directly from them. While system-scoped changes deal with modifications of non-functional features affecting various use case implementations at the same time. Regarding their types, we classify changes as corrective, perfective, and adaptive (as discussed in Section 2.5). On selecting diverse kinds of changes, we can analyze the effects of modularizing use case implementations from different points of view, assuring that the results cover a wide range of possibilities and are not just built upon changes that can favor or degrade a specific concern in our evaluation.

This chapter is organized as follows: in Section 4.1 we detail the Library Management System case study, where we use a toy application in order to consolidate the techniques and the overall evaluation process. Then, in Section 4.2 we present the HealthWatcher case study, which uses a benchmark system for AOP studies allowing a solid and reliable evaluation and analysis. Finally, in Section 4.3 we detail factors that impose threats to the reliability of our findings and in Section 4.4 we discuss the drawn conclusions from both studies.

4.1 LIBRARY SYSTEM

The Library System (LS) is a toy application we implemented with the purpose of consolidating and testing the effectiveness of two AOP-based techniques on modularizing use case implementation: Use Case as Aspect and Use Case as Pluggable Collaboration (presented in the previous chapter, see Section 3.2). The LS requirements (use cases) are based on Bhole's work [BL08].

The system implements a simplified set of functionalities in order to manage a Library. It can be used by both members and library staff. The original OO implementation is composed of seven use cases: **AddBook**, **RemoveBook**, **AddUser**, **RemoveUser**, **AddBookcopy**, **RemoveBookcopy**, and **ShowMenu**. These are administrative operations, used only by the library staff. As the system evolves, the **CheckInBook**, **CheckOutBook**, and **Authentication** functionalities are incorporated to it. Figure 4.1 show the complete use case model of the system.

During the study, we apply the Use Case as Aspect and the Use Case as Collaboration techniques to isolate all use case implementations of the OO original LS. Then, we compare these implementations (use cases implemented as aspects and as collaborations) with the original OO implementation (use cases implemented in the traditional object oriented way, where use cases code impact multiple components). The study is based

Figure 4.1 Library System Use Case Model.

on a typical maintenance and evolution scenario, where a system undergoes through a series of changes request (CR), delivering releases that are further assessed. It is worth to emphasize that we apply the same CR's to the three implementations and that the changes are cumulative, i.e., each further CR is applied to the release produced by a previous CR (except the first CR, that is applied to the base implementation).

4.1.1 System Selection

We decide to use a toy application as our first case study because we need a simple system where we can consolidate the techniques, the assessment procedure, and the analysis and evaluation process. Although being a simple application, the Library Management System was designed to supply the necessary elements to enable the use of all concepts explored by each technique.

4.1.2 System Design

The original implementation of LS is Object Oriented and developed in Java. Its design is based on the Layer architectural pattern [BMR⁺08]. This pattern separates components with the same purpose into layers aiming at isolating changes. Figure 4.2 shows how the LS components are organized into layers, packages, and also how use cases cut across the system elements. The system classes are structured in two main layers: **Application** and **Data**. The **Application** layer contains components that deal with user interface (GUI) and business logic. GUI components are responsible for presenting information to users and accepting and validating input from them. Business components implement the

system business rules. The **Data** layer is responsible for abstracting data access. In the base implementation, data are volatile and stored in **Lists**. In addition to the system's layers, the **Model** package stores components responsible for implementing the domain objects. These objects represent the core concepts of the application; they are used by components pertaining to all architectural layers and have few implementation logic. Books and BookCopies are examples of such components.

Figure 4.2 Library Management System Architecture.

4.1.3 Setup

Figure 4.3 illustrates the steps used to execute the case study. It is organized in 3 phases. In the first phase, we apply the use case modularization techniques to generate the base implementation of the study: (imp1) use cases implemented following the Use Case as Aspect technique where all methods, fields and extra behavior needed to implement the functionality of a use case are grouped in an aspect, constituting a use case module (see Section 3.2.1); (imp2) use cases implemented using the Use Case as Pluggable Collaboration technique where all methods, fields and extra behavior needed to implement a use case are grouped into a collaboration (see Section 3.2.2); and (imp3) use cases OO, where use cases are implemented in the traditional object oriented way with Java (all code related to a use case is scattered over the components of the system). In the second phase, we evolve the system based on a set of change requests (described below) producing four releases of each implementation. Notice that the first CR is applied to the base implementations while the further CR's are applied to each release produced by the previous CR (changes are cumulative). In the third phase, we assess the peer releases of each implementation applying the metrics (defined in Section 3.3) to compare and evaluate the overall system's modularity on accomplishing the required changes. Prior to the

assessment, we define the set of chosen concerns that guides the application of concern metrics.

Figure 4.3 Library System experiment setup.

4.1.4 Change Requests (CR)

In this section we describe the changes that we use to evolve the system. These are typical modifications that are frequently applied to systems under maintenance. It is worth to emphasize that changes are cumulative, that is, the release produced by the application of a CR is used as base code to the following CR.

- **CR1** - Addition of two use cases: **CheckInBook** and **CheckOutBook**. With this CR, two completely new functionalities are added to the LS. This is a use case-scoped and a perfective type of change. It affects classes in all layers of the system.
- **CR2** - Addition of the **Authentication** use case. The library system is modified to authenticate every user by introducing a login (username and password) feature before accessing the system menu. After the validation of login information, the

system associates the user with a type (member or staff). The functionalities are categorized as **Member** or **Administrator**. Depending on the user's type certain operations are allowed to be performed by the user. After checking the user's type, the system shows different menu options regarding each type. This is a use case-scoped and a corrective type of change.

If we just look at the above CR description, we could classify the change as perfective as well. However, the original requirement specification already included this association between users and category which was not contemplated in the base implementation of the system. Because of that, we classify this change as corrective.

- **CR3** - Removal of 1 use case. The AuthenticateUser use case is removed from the system. This modification reveal how costly it is to plug/unplug a use case in the system. This is a use case-scoped and perfective type of change.
- **CR4** - Persistence is added to the Library System. Now, we use a XML file to store information about books, users, operation types and authentication information. This is a system-scoped and an adaptive type of change.

4.1.5 Results

In this section, we evaluate and analyze the results obtained after the assessment phase. We organize our analysis based on the quality software attributes that are under evaluation: separation of concerns, complexity of source code, changeability, traceability, pluggability and parallel development. This set of attributes was defined during the application of the GQM approach in order to guide the selection of metrics used in this work (see Section 3.3).

4.1.5.1 Quantifying Separation of Concerns (SoC)

Table 4.1 exhibits the absolute values collected after CR2 (which contemplates the biggest number of use case concerns). It shows, for each implementation, the Degree of Scattering over Classes (DOSC), Concern Diffusion over Components (CDC) and Lines of Code (LOC) metrics for all use case implementations in the system, and also for others equally important concerns. We analyze three implementations of the system: **uc_oo** (Use Case Object Oriented), **uc_asp** (Use Case as Aspect) and **uc_col** (Use Case as Collaboration). We use **class** as component granularity in all concern metrics extracted in this work.

The set of chosen concerns (used in the application of the SoC metrics) was selected

Table 4.1 Library System - Separation of Concerns Metrics

Concern Name	uc_oo			uc_asp			uc_col		
	dosc	cdc	loc	dosc	cdc	loc	dosc	cdc	loc
1 - AddBook	0.47	5	49	0.45	3	54	0	1	51
2 - RemoveBook	0.53	5	51	0.51	3	56	0	1	53
3 - AddUser	0.59	6	50	0.58	4	55	0	1	52
4 - RemoveUser	0.67	6	49	0.64	4	54	0	1	49
5 - AddBookCopy	0.76	6	46	0.67	3	42	0	1	38
6 - RemoveBookCopy	0.73	6	48	0.65	3	45	0	1	41
7 - ShowMenu	0.49	2	25	0.90	11	55	0.48	12	58
8 - BookCheckin	0.69	8	68	0.54	3	60	0	1	56
9 - BookCheckout	0.80	11	102	0.53	3	89	0	1	84
10 - AuthenticateUser	0.73	6	134	0.67	3	161	0	1	155
11 - Core	0.92	18	302	0.91	18	302	0.91	18	299
12 - GUI	0.92	11	315	0.94	19	322	0.93	17	319
13 - Business	0.86	11	114	0.90	11	94	0.9	11	89
14 - Data	0.65	4	66	0.87	8	62	0.87	8	57
Average DOSC	0.701			0.698			0.323		

in order to observe two previously discussed properties: (i) it contains concerns that represent all important aspects in the system implementation; (ii) and it covers most of the code. This way, we reduce bias in the analysis process as we contemplate a broader range of concerns and not just the ones in the focus of analysis (use case implementations) [EZZ⁺08]. Thus, for the LS, the selected concerns comprehends all functional use cases in the system, the architecture layers (Data and Application) and the system's core (elements that are shared among most use case implementations). Notice that the application layer is composed of GUI and Business elements. As they represent different concepts, we decide to use GUI and Business as concerns instead of the layer (application) itself.

The application of SoC metrics is useful to quantify how effective is the separation of use case implementations in the AO implementations of the system. Since the main goal of the AO solutions is to provide the modularization of these crosscutting concerns (using AspectJ and CaesarJ constructs), we could expect superior SoC outcomes (lower values) in favor of the AO implementations. In fact, all the measures in Figure 4.4 confirm that the AO implementations (**uc_asp** and **uc_col**) exhibit better results with respect to all use cases concerns investigated (1 to 10), except the **ShowMenu** (7). However, when analyzing the complete set of concerns, we see that while improving the modularity of

use case implementations, we degrade (in lesser scale) the modularity of others concerns like **GUI** (12), **Business** (13), and **Data** (14). As a matter of fact, we can see, in the last line of Table 4.1, that the average DOSC (ADOSC) value is practically the same for the **uc_oo** and the **uc_asp** implementations. This happens because while improving functional use case implementations we affect negatively the modularity of concerns that have a relation with use cases. For example, functional use cases always contains code related to the **Data** concern. So, when we modularize use case implementations, we extract the code related to **Data** access to separate use case modules, thus, scattering the **Data** concern.

(a) Degree of Scattering Over Classes

(b) Concern Diffusion Over Classes

Figure 4.4 Library System - Separation of Concerns

Now, let's look at the Vocabulary Size (VS) and Lines of Code (LOC) values associated with each concern. As previously observed, higher values of VS tend to occur in AO solutions because we add extra components (aspects) in order to modularize cross-cutting concerns. Lower values of LOC occur because AOP, usually, avoid duplicated code as an effect of its quantification property applied to homogeneous crosscutting con-

cerns. Because of the heterogeneous nature of use case implementations, we do not gain with quantification. This way, we have a small increase in the LOC values in the AO implementations for most concerns.

The CDC and DOSC values for the **uc.col** implementation should be analyzed carefully. For all use case concerns, they present the *desired* values for these metrics. A DOSC value equals to 0 and a CDC value equals to 1 means that a concern is completely localized and confined in just one component. This happens because of the way a use case collaboration is implemented. A collaboration is a set of collaborating **cclass** in order to implement the whole functionality of a use case (see Section 3.2.2). For use cases that does not have a relationship with others, their implementations are restrict to just one component. However, within this component (implemented as a collaboration) we define new classes that are specific to a use case implementation, and also refine existing classes (that are shared among various use cases) adding extra behavior that are related only to this use case implementation. Therefore, a collaboration contains a set of others components in its definition, although being considered as just one component. Because of that, we can see in Figure 4.4 that the curves (CDC and DOSC) for the **uc.col** implementation present superior results than the **uc.asp** implementation. But even with these great results regarding functional use case concerns, the **uc.col** implementation shows the same tendency on degrading the modularity of some previously well-modularized concern in the **uc.oo** implementation, like **Data** and **Business**.

An important consideration in order to provide a fair comparison between the two AO implementations is that a component in CaesarJ is closely related to a package in Java/AspectJ regarding granularity level. As we saw in the previous section 3.3.1.1, granularity is a key concept to assure that concern metrics produce reliable results. Due to the way we use the CaesarJ class in the **uc.col** solution, although comparing both AO implementations at the **class** level, we find these disparities on the results. If we had considered package as components instead of class in the **uc.asp** solution, we also would have DOSC values equal to 0 and CDC equals to 1, producing more aligned results with the **uc.col** implementation.

Table 4.2 shows the Average Degree of Tangling (ADOT) metric and the consolidated data on the number of dedicated components in the system. Lower ADOT value indicates a system with less tangled code resulting in better modularity. The results show that the AO implementations have a slight advantage regarding code tangling. This occurs mainly because we extract all code related to use case implementations making components, like the **RepositoryXX**, that have high DOT values in the **uc.oo** implementation, dedicated to the **Core** concern in the AO implementations. Notice the lower value regarding the number of components in the **uc.col** solution due also to the notion of components we

assume in our evaluation.

Table 4.2 Library System - Degree of Tangling Metrics

	# Components	Tangled	%Not Tangled	ADOT
UC_OO	38	29	23%	0.45
UC_Aspect	50	34	32%	0.38
UC_Collaboration	24	8	25%	0.41

As a result, our analysis indicates that SoC metrics are very important as a modularity indicator, but they are extremely dependent of the set of chosen concerns and the notion of components as discussed above. For example, if we had just considered the set of functional use cases we would state that we had an absolute modularity gain. However, while analyzing all important concerns, regarding the LS implementation, we obtain a relative improvement on the overall system's modularity.

4.1.5.2 Quantifying Complexity of Source Code

In this section, we discuss how modularized use case implementations impact positively or negatively on the complexity of source code by quantifying its effects on coupling, cohesion, and size metrics. The measures were gathered according to the system perspective; that is, they represent the total of metric values associated with all classes and aspects in the system implementation.

Tables¹ 4.3 and 4.4 present the absolute values for all collected metrics related to the Use Case OO (**uc_oo**) and to the Use Case as Aspect (**uc_asp**) implementations, respectively. As mentioned before, we did not collect these metrics for the **uc_col** implementation due to the lack of an assessment tool that can define and compute such metrics in the CaesarJ context.

Table 4.3 Library System - UC_OO Complexity Metrics

Release	CBC	CBC/VS	LCO	LCO/VS	DIT	LOC	VS
Base	51	1.70	119	3.97	19	577	30
CR1	73	2.03	212	5.89	25	726	36
CR2	79	2.08	246	6.47	25	869	38
CR3	72	2.00	215	5.97	25	731	36
CR4	72	2.00	215	5.97	25	733	36

¹Legend: CBC - Coupling between components, VS - Vocabulary size, LCO - Lack of Cohesion over operations, DIT - Depth of inheritance tree and LOC - Line of code

Table 4.4 Library System - UC_ASP) Complexity Metrics

Release	CBC	CBC/VS	LCO	LCO/VS	DIT	LOC	VS
Base	61	1.56	106	2.72	25	621	39
CR1	87	1.85	149	3.17	33	780	47
CR2	95	1.90	179	3.58	33	941	50
CR3	87	1.85	140	2.98	33	780	47
CR4	87	1.85	140	2.98	33	786	47

Figure 4.5 plots the absolute values for Vocabulary Size (VS - number of components) and Lines of Code (LOC) throughout the releases. The VS graphic shows that the **uc_asp**

Figure 4.5 Library System - Vocabulary Size and Lines of Code

implementation tends to be worse in all releases. This is due to the introduction of aspects that are used to modularized the use cases. For each use case in the system, we need to define, at least, one new aspect or collaboration which contains the specific code related to the use case implementation. This is an unavoidable effect of software *aspectization*: a system tends to have more components, but less tangled and scattered code; as the addition of new aspect components aims at modularizing the crosscutting concerns.

Analyzing the Lines of code (LOC) values we have an interesting outcome; previous works [SLB02, KSG⁺06] observed that although defining more components, AO solutions tend to have less LOC. However, in the context of use case implementations, we do not see this effect. This happens because of the heterogeneous nature of this specific crosscutting concern. Heterogeneous crosscutting concerns imply minor gain with quantification, which implies no reduction on LOC values. In our case, we have an increase in LOC values which means a slightly increase on the complexity of source code.

Now, let's analyze Coupling Between Components (CBC) and Lack of Cohesion over Operations (LCO) metrics.

Figure 4.6(a) shows that there is a small difference in favor of the **uc_oo** implementation with respect to the absolute value of the coupling metric (CBC). It happens mainly because, when we modularize the crosscutting concerns into aspects, we reduce the coupling of system classes. However, these aspects still have to hold references to the classes in which they introduce some state or behavior. In this case, an aspect implementing the functionalities of a use case introduces code in at least 3 different classes. This contributes to increase the coupling regarding the system perspective. However, considering the AO implementation has more components (classes and aspects) demonstrated by the VS metric, we can observe that it has produced, on average (CBC/VS), more decoupled classes and aspects.

Figure 4.6 Library System - Coupling and Cohesion

Figure 4.6(b) shows the superiority of the **uc_asp** implementation with respect to the cohesion metric (LCO), which presents lower values in all releases. The **uc_asp** solution is, on average, 27% superior in the absolute value and also 45% superior in the average values per component (LCO/VS). The production of components with a higher cohesion was an expected effect in the AO solution of use case implementations; this can be explained by the fact that when we modularize use case code into an aspect or a collaboration, we separate methods that access different fields (mainly because they address different use cases code) to distinct aspects. This action contributes to increase the cohesion of the **uc_asp** implementation (decrease the LCO value). Although the use case aspects encapsulate crosscutting behavior applied to different components, these behaviors are directly related to each other producing lower LCO values. For example, a use case aspect contains all methods related to a use case implementation. Each method address a component of a different layer in the system. However, a method from an upper layer calls a method from the layer immediately below, this way, all methods are related to

each other considering the chain of architecture layer of the system.

Figure 4.7 displays the graphics for the average cohesion and coupling per component. Observing the curves, the **uc_asp** implementation has better results for both coupling and cohesion.

Figure 4.7 Library System - Average Coupling and Cohesion

Finally, Figure 4.8 shows that Depth of Inheritance Tree (DIT) metric presents better results in the **uc_oo** implementation of the system, showing similar behavior among the releases. The slightly higher values of the **uc_asp** solution is due to the creation of an aspect hierarchy in order to enable code reuse.

Figure 4.8 Library System - Depth of Inheritance Tree

Regarding complexity of source code, we conclude that in all releases, the **uc_asp** implementation presents better results with respect to average coupling and cohesion while the **uc_oo** presents better results with respect to depth of inheritance tree (DIT) and size metrics (VS and LOC).

4.1.5.3 Quantifying Changeability

In this section, we analyze the impact of changes quantifying its effect on a set of metrics presented in Table 4.5, such as, number of added or changed components (aspects/classes) and number of added LOC. The purpose of using these metrics is to quantitatively assess the propagation of change effects, when applying a set of change requests. Lower values of the metrics denote a less invasive and a more localized process of change; which is a desired side effect produced by well-modularized systems.

Table 4.5 Library System - Changeability Absolute Values

Release	Changed Comp.			Added Comp.			Added LOCs		
	uc_oo	uc_asp	uc_col	uc_oo	uc_asp	uc_col	uc_oo	uc_asp	uc_col
CR1	4	3	4	6	6	2	149	165	145
CR2	5	0	0	2	3	1	145	161	153
CR3	3	0	1	0	0	0	0	0	0
CR4	2	6	7	2	2	2	58	30	28
Total	14	9	12	10	11	5	352	356	326

Looking at Table 4.5 we see that much more extensive changes (in terms of number of changed components) occur in the **uc_oo** implementation. Up to 25% fewer components are changed in the AO implementations (**uc_asp** and **uc_col**) throughout the releases (see Table 4.6). This indicates that the AO implementations conform more closely to the Open-Closed principle [Mey88] which states that “*software should be open for extension, but closed for modification*”. It was observed in previous works [KSG⁺06, GBF⁺, GSF⁺06], that, generally, AO solutions require more new components to implement a change. However, OO solutions require existing components to be modified more extensively to implement the same change. We confirm this behavior in this study.

Table 4.6 Library System - Changeability Percentage

Release	% Unchanged Comp.			%Changed Comp.		
	uc_oo	uc_asp	uc_col	uc_oo	uc_asp	uc_col
CR1	86.7	92.4	83.4	13.3	7.6	16.6
CR2	86.1	100.0	100.0	13.9	0.0	0.0
CR3	92.1	100.0	96.3	7.9	0.0	3.7
CR4	94.5	87.2	75.0	5.5	12.8	25.0
Total	59.4	79.6	54.7	40.6	20.4	45.3

Propagation of Use Case Scoped Changes is superior in the AO implementations. When considering the change requests (CR1, CR2, and CR3) where the AO implementations are superior, they require fewer changes to components (in terms of modified components) and more added components. Considering the number of components manipulated (added + changed) at each change, the AO implementations are, on average, 50% superior than the **uc_oo** implementation. This happens because these CR's are applied directly to use case implementations which are implemented, in the AO solutions, as separate modules. While, the **uc_oo** implementation requires new fields and methods to be added directly to the system's core classes in order to implement the same changes in these releases.

Propagation of System Scoped Changes is superior in the Use Case OO implementation. When considering the change request where **uc_oo** implementation is superior (CR4) the inverse behavior is observed; the **uc_oo** implementation now require fewer changes to components and the same number of components to be added in the system. This happens because when implementing a system scoped change, the code related to the change, in the AO implementations, can be spread into the use cases units (aspect or collaboration). For example, in CR4 we change how the Library information is stored; this impacts all use cases in the system. Thus, while in the **uc_oo** implementation this change is restricted to the **Repository** elements confined in the **Data** layer; in the AO implementations, it is spread throughout all use case modules producing a more scattered change.

From these observations, we can conclude that propagation of change effects is extremely dependent on the type of change. For example, if a system has its use case implementations modularized, it will absorb better the effects of use case scoped changes. But, will have a more invasive process of change when subjected to broader scoped changes (system-scoped).

4.1.5.4 Quantifying Traceability, Pluggability and Parallel Development

In this section, we analyze how the isolation of functional use case implementations affects traceability (from requirement to source code), pluggability and parallel development.

Table 4.7 presents the metrics that evaluate how costly it is to plug/unplug a use case implementation regarding the three implementations of the system: Use Case OO (**uc_oo**), Use Case as Aspect (**uc_asp**) and Use Case as Collaboration (**uc_col**). We collect the results after the application of CR3 (Removal of 1 use case in the system). We consider that a use case code is completely removed from the system when we exclude the dedicated components, and comment the lines of code related to its implementation on

classes that share code with others use cases. After that, we count how many components we need to modify; also, how many blocks and lines of code we need to comment in order to completely eliminate the use case implementation. Looking at the metrics values on

Table 4.7 Library System - Pluggability

Changed Comp			blocks with Dead LOC			#Total Dead LOC		
uc_oo	uc_asp	uc_col	uc_oo	uc_asp	uc_col	uc_oo	uc_asp	uc_col
5	1	2	6	0	2	28	0	2

Table 4.7 we can conclude that the AO implementations (**uc_asp** and **uc_col**) are superior than the **uc_oo**. While we need to modify 5 components in the **uc_oo** implementation, we just need to change 1 component in the **uc_asp** and 2 in the **uc_col** implementation. This was an expected result due to the fact that the AO solutions were developed aiming to modularize use cases code, therefore, having use case implementations confined in less components. Because of that, we can conclude that is easier to plug/unplug a use case in the AO implementations of the system.

Table 4.8 presents the results regarding parallel development. The metrics quantify the effects of such attribute by computing the Core Influence (CI), that is, the amount of code that is shared between use cases in the system. In the **uc_oo** implementation, use cases code are spread over the system components, therefore, we consider the **core** as the system itself. So, the higher is the amount of code in the system's core, the higher is its influence on contributing to a more complex parallel development management. We also measure the Number of Adjustment in Core (NAC). This metric computes the

Table 4.8 Library System - Parallel Development

Releases	Core Influence			# adjustment in core		
	uc_oo	uc_asp	uc_col	uc_oo	uc_asp	uc_col
CR1	1	0.46	0.43	0	0	0
CR2	1	0.38	0.37	0	3	3
CR3	1	0.32	0.31	0	0	0
CR4	1	0.38	0.37	0	0	0

number of necessary adjustments in the system's core to accomplish modifications as the system evolves. Most of the adjustments are due to the fact that when we introduce new use cases in the system, we need to factor out some methods that were specific to the realization of an existing use case, but in a further release turn to be shared with others.

In the **uc_oo** implementation, we consider the core as being the whole system, because of that, we do not compute any *adjustment*, as all methods are already defined in the core.

Analyzing these metrics values we can conclude that the AO solutions (**uc_asp** and **uc_col**) demonstrate superior result than the **uc_oo** solution. This can be justified by the fact that modularizing use case implementations produces a system where it is easier to assign independent use cases code to different developers; thus, contributing to decrease the complexity on managing parallel development. Both approaches will still depend on a Version Control Tool and on a specific parallel development policy used by the development team. These actions deal with conflict resolutions that, in theory, tend to happen less in the AO implementations, even though, sometimes, adjustments in the system's core are still necessary, as seen in release CR2. We have made these observations considering that the development process is use case driven, thus, developers implement use cases and not components. If, for example, we had another approach to software development and a developer had to develop the **Data** layer, modularizing use case code would increase the complexity on managing parallel development.

Finally, Table 4.9 presents the results for traceability metrics. We want to be able to quantify how easy it is to trace changes from use case specifications to code. With this purpose, we select two metrics: CDC (number of components that implement a use case) and #SF (number of source files that implement use case). Lower values for these metrics imply on a system where is easier to create traceability links from use cases specifications to code assets. Looking at the results (gathered after CR3), we can conclude that AO implementations (**uc_asp** and **uc_col**) promotes better conditions to track functional system changes as the system evolves.

Table 4.9 Library System - Traceability

	uc_oo		uc_asp		uc_col	
Use Cases	CDC	#SF	CDC	#SF	CDC	#SF
AddBook	5	5	3	3	1	3
RemoveBook	5	5	3	3	1	3
AddUser	6	6	4	4	1	3
RemoveUser	6	6	4	4	1	3
AddBookCopy	6	6	3	3	1	3
RemoveBookCopy	6	6	3	3	1	3
ShowMenu	2	2	11	11	12	12
BookCheckin	8	8	3	3	1	3
BookCheckout	11	11	3	3	1	3
AuthenticateUser	6	6	3	3	1	3

4.2 HEALTH WATCHER

The HealthWatcher (HW) [SLB02] is a real web-based information system with available OO and AO implementations. The main goal of the HW system is to improve the quality of services provided by public health care institutions. For this purpose, citizens use the HW system to register complaints regarding health issues, and health care institutions use the HW system to investigate the reported complaints and take the required actions.

The HW original implementation is composed of ten use cases. Figure 4.9 shows its use case model.

Figure 4.9 Use Case Model for the Original HW Implementation.

In this study, we apply the Use Case as Aspect technique (see Section 3.2.1) to isolate all use case implementations of the AO original HW implementation. Then, we compare this implementation (use cases implemented as aspects) with the original AO solution (use cases implemented in the traditional object oriented way, where use cases code impact multiple components). The study follows the same steps used on the LS case study, where the target system is exposed to a set of changes request (CR). Each CR produces releases that are further evaluated and analyzed regarding modularity improvement.

We select the HW AO implementation as our start point in this study because we want to investigate how the modularization of use case implementations impact on already modularized concerns (like persistence, distribution and concurrency). In the HW OO implementation, these concerns cut across the system components. Hence, on modularizing use cases code, these concerns would still be spread in a similar way. Therefore, we would not be able to quantify the effects of use case modularization on previous well-modularized non use case concerns.

4.2.1 System Selection

We choose the Health Watcher (HW) system as the subject of this study because of the following:

1. Its design has a significant number of crosscutting and non-crosscutting concerns, such as, persistence, concurrency, and distribution; beyond that, their implementations involve the application of mainstream technologies commonly used in industrial contexts, like Java Remote Method Invocation (RMI), Servlets, and Java Database Connectivity (JDBC).
2. A use case document is available, which is essential in the definition of the concerns used to guide the extraction of concern based metrics; the selected concerns should represent the rationale of implementation and cover most of the code, as discussed in Section 3.3.1.1.
3. Qualitative and quantitative studies of the HW system have been conducted [GBF⁺, KSG⁺06] which allows us to correlate our results with them.

4.2.2 System Design

The AO implementation of the HW system is developed with AspectJ. Its architecture is also based on the Layer pattern having the system classes structured in three main layers: **View**, **Business** and **Data**. The crosscutting concerns **distribution**, **persistence**, and **concurrency** are implemented as aspects. Figure 4.10 shows a simplified class diagram of the AO original implementation, highlighting the system's architecture through its main components. Note that use case implementations cut across components defined in the hierarchy of layers.

The **View** layer is related to the HW web user interface. The Java Servlet API is used to implement this layer. The **Business** layer is responsible for implementing the classes that define the system business rules. The HWFacade is the unique point of interaction with this layer; it uses record components to interact with the **Data** layer. The **Data** layer is responsible for abstracting the functionality of database persistence using the JDBC API. Components in the **Model** package are responsible for implementing the domain objects. These objects represent the core concepts of the application; they transit among all architectural layers and have a few implementation logic. Complaint and Employee are examples of core concepts in the HW system. The **Distribution** aspects distributes the system services provided by the **Business** layer; it is implemented using the RMI technology. The **Persistence** aspects modularize transaction control and connection pool.

Figure 4.10 HW AO original architecture design.

The **Concurrency** aspects deal with code that implements concurrency control policy. The abovementioned design decisions have shown to be effective to modularize most of the non-functional system concerns: distribution, persistence and concurrency. However, code relative to use case implementations is still tangled and scattered throughout the system's components.

4.2.3 Setup

Figure 4.11 illustrates the steps used to execute the case study. It is organized in 3 phases. In the first phase, we apply the use case modularization technique to generate the base implementations of the study: (imp1) use cases implemented following the Use Case as Aspect (**uc_asp**) technique (all methods, fields and extra behavior needed to implement the functionality of a use case are grouped in an aspect, constituting a use case module); and (imp2) Use cases OO (**uc_oo**), where use cases are implemented in the traditional object oriented way with Java (all code related to a use case is scattered over the components of the system). In the second phase, we evolve each implementation based on a set of change requests (described below) producing four releases. Each change request is applied at the most recent release, for example, CR2 is applied in the release generated after the implementation of CR1. In the third phase, we assess the peer releases applying the metrics (defined in Section 3.3) to compare and evaluate the overall system's

modularity on accomplishing the required changes. Prior to the assessment, we define the set of chosen concerns that guides the application of concern-based metrics.

Figure 4.11 Case Study Setup.

4.2.4 Change Requests (CR)

We evolve both implementations of the base release according to the following change requests:

- **CR1** - Addition of 6 new functional use cases. The functionalities introduced by these new use cases represent typical operations encountered in the maintenance of information systems. They naturally involve modification of modules implementing several system concerns. The new use cases requires changes in classes of all the system's layers. Figure 4.12 shows the updated use case model of the HW system after CR1. The new use cases are represented by a different (darker) color. This is a use case-scoped and perfective type of change.

CR2 - Modification in 2 use cases. In this request we contemplate different kinds of changes: change on data manipulation and change on functional behavior.

Figure 4.12 HealthWatcher Use Case model after CR1.

In the first situation, the **InsertEmployee** use case is changed to add a new field on the Employee record. The original Employee data includes name, login and password. Now, we want to store the date on which the employee is registered in the HW system.

In the second, the **UpdateMedicalSpecialty** use case is modified to allow a **Medical Specialty** to be updated by directly entering its code. In the original implementation, to update a medical specialty we first list all medical specialties registered in the system; then, from this list we select the one we want to update. Now, we have the option to update a medical specialty using its code instead of selecting it from a list.

These are use case scoped and perfective type of change.

- **CR3** - Removal of 1 use case. This modification removes the SearchMedicalSpecialty use case. It excludes all specific files and comment all lines of code related to the use case. This change aims at providing a specific evaluation on how easy is to plug/unplug a functional use case in the system. This is a use case-scoped and perfective type of change.
- **CR4** - Change of the persistence mechanism. This modification alter the persistent mechanism to store data on a XML file instead of a database (JDBC). This type of change aims at evaluating how an adaptation of a non-functional concern affects the functional use cases in the system. This is a system-scoped and adaptive type of change.

4.2.5 Results

In the next sections, we analyze the results following the same structure used in the LS case study; we evaluate each quality software attribute using the set of metrics defined to quantify their effects on the system's modularity. In general, the HW results confirms the outcomes obtained in the LS case study.

4.2.5.1 Quantifying Separation of Concerns (SoC)

Table 4.1 exhibits the absolute values collected after CR2 (which contemplates the biggest number of use case concerns). It shows, for each implementation, the degree of scattering over classes (DOSC), the concern diffusion over components (CDC), and the lines of code (LOC) metrics for all use case implementations in the system, and also for others equally important concerns.

As discussed before, we chose the concerns following the guideline to obtain a set of concerns that represents a rationale of the system implementation and that covers most of the code. Thus, for the HW case study, the selected concerns comprehends all functional use cases in the system, the crosscutting concerns: persistence, distribution and concurrency (that are already modularized in the HW base implementation of the study), and the system's core (elements that are shared among all the use case implementations). As mentioned before, since the main goal of the **uc_asp** solution is to provide the modularization of use case implementations, we could expect superior SoC outcomes in favor of the **uc_asp** implementation. In fact, the curves in Figure 4.13 confirm this impression where the **uc_asp** implementation exhibits better results with respect to all use cases concerns (4-19) investigated. However, when analyzing the **Distribution** (1), **Concurrency** (2), and **Persistence** (3) concerns we see that their modularization is heavily degraded. It happens because these concerns are directly related to use cases, that is, all use cases contain persistence and distribution code in their implementation; when modularizing use case implementations, we extract the code related to **Persistence**, **Concurrency** and **Distribution** to its own use case aspect, thus, scattering these concerns over all use case modules.

Regarding the Average Degree of Scattering (ADOSC) metric, we have similar outcomes as the ones obtained in the LS case study. We see in the last line of Table 4.1 that the ADOSC value is slightly better in the **uc_asp** implementation. This happens because while improving functional use cases implementation, we affect negatively the modularity of all concerns that have a relation with them; in this case, **persistence**, **concurrency** and **distribution**. The small advantage in favor of the **uc_ao** implementation is

Table 4.10 HeathWatcher AO imlementation - Separation Of Concerns

	uc_oo			us_asp		
Concern Name	dosc	cdc	loc	dosc	cdc	loc
1 - Distribution	0.78	9	347	0.81	24	370
2 - Concurrency	0.49	4	187	0.54	6	187
3 - Persistence	0.84	17	2144	0.94	32	1815
4 - SearchDisease	0.69	9	239	0.49	3	270
5 - UpdateHealthUnit	0.68	8	263	0.55	3	185
6 - SearchSpecialitiesByHealthUnit	0.75	9	129	0.65	3	137
7 - UpdateMedicalSpeciality	0.78	10	201	0.74	4	216
8 - InsertDiseaseType	0.68	8	112	0.49	2	127
9 - InsertNewEmployee	0.70	8	100	0.43	2	105
10 - InsertNewSymptom	0.64	8	96	0.50	2	113
11 - UpdateSymptom	0.80	10	291	0.64	4	329
12 - UpdateEmployee	0.75	9	93	0.66	3	77
13 - InsertNewHealthUnit	0.62	8	92	0.50	2	105
14 - InsertNewComplaint	0.80	12	417	0.48	4	555
15 - UpdateComplaint	0.67	9	255	0.59	3	273
16 - SearchComplaint	0.48	8	178	0.45	2	189
17 - InsertNewMedicalSpeciality	0.62	8	93	0.50	2	106
18 - SearchHealthUnitBySpecialities	0.76	9	138	0.63	3	155
19 - Login	0.24	6	70	0.24	2	71
20 - Core	0.93	53	1919	0.93	53	1919
Average DOSC	0.709			0.602		

due to the fact that we have a big set of use case concerns contributing to diminish the average scattering of the system. Now, let's look at the LOC values associated with each concern on Table 4.10. The LOC values are very similar for both implementations. As discussed before, we have almost no gain with quantification while modularizing use case implementations due to its heterogeneous nature. This effect contributes to maintain the LOC values almost the same.

Table 4.11 shows the Average Degree of Tangling (ADOT) metric and the consolidated data on the number of dedicated components in the system. Lower ADOT value indicates a system with less tangled code, which contributes to enhance modularity. The results show that the **uc_asp** implementation has a significant advantage regarding code tangling. This occurs mainly because we extract all code related to use case implementations making components, like **HWFacade**, **RecordXX**, **RepositoryXX**, that have high DOT values in the **uc_oo** implementation, dedicated to the **Core** concern in the **uc_asp**

(a) Degree of Scatering over classes

(b) Concern Diffusion over classes

Figure 4.13 HealthWatcher - Separation of Concerns

implementation.

After analyzing all SoC metrics, we confirm their importance as a modularity indicator, but we reinforce that they are extremely dependent on the set of chosen concerns. For example, if we had not considered **Persistence**, **Distribution** and **Concurrency** in our analysis, the **uc_asp** implementation would had superior results for all analysed metrics. However, while considering all important concerns regarding the HW implementation, we achieve the same relative improvement on the overall system's modularity, as obtained in the LS case study.

Table 4.11 Health Watcher - Degree of Tangling

	# Components	Tangled	Not Tangled	% Not Tangled	ADOT
uc_oo	107	78	29	72%	0.158
uc_asp	124	15	109	88%	0.064

4.2.5.2 Quantifying Complexity of Source Code

In this section, we discuss how modularized use case implementations impact positively or negatively on the complexity of source code by quantifying its effects on coupling, cohesion, and size metrics. Also in this study, the measures were gathered according to the system perspective; that is, they represent the total of metric values associated with all classes and aspects in the system implementation.

Table 4.12 and Table 4.13 present the absolute values for all collected metrics for the Use Case OO (**uc_oo**) and Use Case as Aspect (**uc_asp**) implementations, respectively.

Table 4.12 HealthWatcher - Use Case Implementations OO

Release	CBC	CBC/VS	LCO	LCO/VS	DIT	LOC	VS
Base	261	2.66	775	7.91	94	5525	98
CR1	354	3.16	924	8.25	124	6629	112
CR2	354	3.16	924	8.25	124	6629	112
CR3	337	3.09	886	8.12	115	6415	109
CR4	364	3.13	925	7.97	115	7289	116

Table 4.13 HealthWatcher - Use Case as Aspect

Release	CBC	CBC/VS	LCO	LCO/VS	DIT	LOC	VS
Base	327	3.05	678	6.33	92	5466	107
CR1	462	3.61	678	5.29	122	6583	128
CR2	462	3.61	678	5.29	122	6579	128
CR3	426	3.49	656	5.38	113	6203	122
CR4	463	3.53	717	5.47	113	7146	131

Figure 4.14 plots the absolute results for Vocabulary Size (VS - number of components) and Lines of Code (LOC) throughout the releases. The VS graphic shows that the **uc_asp** implementation tend to be worse in all releases. This happens because of the same effect

(a) Vocabulary Size

(b) Lines of Code

Figure 4.14 HealthWatcher - Vocabulary Size and Lines of Code

seen in the LS case study: the introduction of aspects that are used to modularized the use cases. For each use case in the system, we need to define, at least, one new aspect containing the specific code related its implementation.

Regarding the Lines of Code (LOC) values, we confirm the tendency observed in the LS case study. Because of the heterogeneous nature of use case implementations, the LOC values do not present a considered decrease in the **uc_asp** implementation. We have similar values for both solutions throughout all releases.

Now, let's analyze Coupling Between Components (CBC) and Lack of Cohesion over Operations (LCO) metrics.

Figure 4.15(a) shows that there is a considered difference in favor of the **uc_oo** implementation with respect to the absolute value of the coupling metric (CBC). It happens mainly because, although many of the aspects reduce the coupling of system classes by modularizing their respective crosscutting concerns, they still need to hold references to the classes in which they introduce some state or behavior. Due to the class hierarchy of the HW system, a use case implementation has methods spread over, at least, 6 components; therefore, a use case aspect needs to introduce these methods at their respective classes, contributing to a significant increase on the coupling metric. This way, despite having more components (classes and aspects), demonstrated by the VS metric, we can observe that the **uc_asp** implementation has produced, on average (CBC/VS), more coupled classes and aspects. An important observation here is that we find the same effect (increase coupling) as the one obtained while breaking down a class into 2 others classes (main and auxiliary, for example) or adding an interface between 2 previous directly related classes. Although these actions contributes to increase the coupling values, they are considered good design decisions in order to provide better code maintenance.

In the LS case study, the **uc_asp** implementation presented superior result for the av-

erage coupling (CBC/VS) metric. We can explain this fact by looking at the architecture of both systems. A deeper class hierarchy of the HW system generates, on average, 3 more introductions on each use case aspect; thus, increasing the coupling values.

(a) Coupling

(b) Cohesion

Figure 4.15 HealthWatcher - Coupling and Cohesion

Figure 4.15(b) shows the superiority of the **uc_asp** implementation with respect to the cohesion metric (LCO). The **uc_asp** implementation is, on average, 27% superior in the absolute value and also 45% superior in the average values per component (LCO/VS). The production of components with a higher cohesion was an expected effect in the **uc_asp** implementation; it can be explained by the fact that when we modularize use case code into an aspect, we separate methods that access different fields (mainly because they address different use cases code) to distinct aspects. This action contributes to increase the cohesion of the **uc_asp** implementation (decrease the value of LCO). Although use case aspects encapsulate crosscutting behavior applied to different components, these behaviors can be directly related to each other producing lower LCO values, as explained in the LS case study.

Whereas the LCO metric, we find even better outcomes for the HW case study when compared with the LS case study. This is also justified by the increased complexity of the HW system architecture which produces components, in the **uc_oo** implementation, with higher LCO values; so, when modularized in the **uc_asp** implementation, a greater improvement in the LCO values is achieved for the HW system.

Figure 4.16 displays the graphics for the average cohesion and coupling per component. Observing the curves, **uc_asp** implementation has better result for cohesion while the **uc_oo** implementation presents superior outcome for coupling.

Figure 4.17 shows that the Depth of Inheritance Tree (DIT) metric has similar results for both implementations and present also uniform behavior among all releases. This

Figure 4.16 Health Watcher - Average Coupling and Cohesion

outcome confirms the results obtained in the LS case study.

Figure 4.17 HealtWatcher - Deph of Iheritance Tree

Summarizing our findings regarding complexity of source code, we conclude that throughout all releases, the `uc_asp` implementation presents better results with respect to lines of code (LOC) and cohesion (LCO) while the `uc_oo` implementation presents better results for coupling (CBC) and vocabulary size (VS) metrics.

4.2.5.3 Quantifying Changeability

Regarding the quantification of change effects, we obtained the same outcomes as the LS case study.

Table 4.14 shows the absolute values for the changeability metrics. We can confirm that the `uc_asp` implementation conforms more strictly to the Open-Closed principle (when we consider the total obtained in all releases). In this case, it requires more new components and less modifications in existing components to implement a change. Up to 14% fewer components are changed in the `uc_asp` implementation throughout the

releases, as seen in Table 4.15.

Table 4.14 HealthWatcher - Changeability Absolute Values

Release	Changed Comp.		Added Comp.		Added LOC	
	uc_oo	uc_asp	uc_oo	uc_asp	uc_oo	uc_asp
CR1	17	4	14	21	1104	1151
CR2	8	2	0	0	31	33
CR3	7	0	0	0	0	0
CR4	3	18	7	7	874	782
Total	35	24	21	28	2009	1966

Propagation of Use Case Scoped changes is superior in the Use Case as Aspect implementation. When considering the change requests (CR1, CR2, and CR3) where the **uc_asp** is superior, they require fewer changes to components (in terms of modified components) and more added components. Considering the number of components manipulated at each change, the **uc_asp** implementation is, on average, 43% superior than the **uc_oo** implementation. This happens because these CR's are applied directly to use case implementations which are implemented, in the **uc_asp** implementation, as separate modules. While in the **uc_oo** implementation new fields and methods are added directly to the system's core classes in order to implement the same changes.

Table 4.15 HealthWatcher - Changeability Percentage

Releases	% Unchanged Comp		% Changed Comp.	
	uc_oo	uc_asp	uc_oo	uc_asp
CR1	81.9	96.1	18.1	3.9
CR2	92.6	98.4	7.4	1.6
CR3	93.5	100.0	6.5	0.0
CR4	97.1	85.2	2.9	14.8
Total	65.1	79.7	34.9	20.3

Propagation of System Scoped Changes is superior in the Use Case OO implementation. When considering the change request where the **uc_oo** implementation is superior (CR4), the inverse behavior is observed; that is, the **uc_oo** implementation now require fewer changes to components and the same number of components to be added in the system. This happens because when implementing a system-scoped change in the **uc_asp** implementation, code related to the change is scattered across the use cases

modules. In this case, the CR4 deals with the modification of the persistent mechanism which impacts all use cases in the system. In the **uc_oo** implementation, this change is restricted to **Repository** elements which are confined in the **Data** layer. On the contrary, in the **uc_asp** implementation, there are persistence code in all use case aspects, resulting in a more invasive change.

These outcomes reinforce the conclusions obtained in the LS case study. Propagation of change effects is extremely dependent on the type of change. A system under a use case decomposition, generally, absorbs better the effects of use case scoped changes; having a more invasive process of change when subjected to broader scoped changes (system-scoped). This observation can guide the software architects by providing criteria to support the decision of which form of decomposition fit best to a specific system context.

4.2.5.4 Quantifying Traceability, Pluggability and Parallel Development

Regarding Traceability, Pluggability and Parallel Development we ratify all the results obtained in the LS case study.

Table 4.16 presents the metric values that quantify how costly it is to plug/unplug a use case on both implementations of the system. We collect the results after the application of CR3 (Removal of 1 use case in the system). Looking at the results, we can find that is much easier to plug/unplug a use case in the **uc_asp** implementation of the system. We need less changes in components, and also fewer lines of code need to be commented to eliminate all code related to a use case from the system.

Table 4.16 Health Watcher - Pluggability

Changed Comp		Blocks with Dead LOC		#Total Dead LOC	
uc_oo	uc_asp	uc_oo	uc_asp	OO	AO
7	1	7	0	42	0

Table 4.17 presents the results for parallel development metrics. It shows that the **uc_asp** implementation achieves better results based on the lower values of the CI and NCA metrics throughout the releases.

Table 4.17 HealthWatcher - Parallel Development

Release	Core Influence (CI))		#adjustment in core	
	uc_oo	uc_ao	uc_oo	uc_ao
Base	1	0.52	0	0
CR1	1	0.44	0	5
CR2	1	0.44	0	0
CR3	1	0.46	0	0
CR4	1	0.42	7	7

Table 4.18 shows the result for traceability metrics. As we can see, we need less components to implement a use case in the **uc_asp** implementation of the system. This fact decreases the complexity on the creation and maintenance of traceability links from requirement specification to code assets.

Table 4.18 HealthWatcher - Traceability

Concern Name	uc_oo		uc_asp	
	cdc	#sf	cdc	#sf
SearchDisease	9	9	3	3
UpdateHealthUnit	8	8	3	3
SearchSpecialitiesByHealthUnit	9	9	3	3
UpdateMedicalSpeciality	10	10	4	4
InsertDiseaseType	8	8	2	2
InsertNewEmployee	8	8	2	2
InsertNewSymptom	8	8	2	2
UpdateSymptom	10	10	4	4
UpdateEmployee	9	9	3	3
InsertNewHealthUnit	8	8	2	2
InsertNewComplaint	12	12	4	4
UpdateComplaint	9	9	3	3
SearchComplaint	8	8	2	2
InsertNewMedicalSpeciality	8	8	2	2
SearchHealthUnitBySpecialities	9	9	3	3
Login	6	6	2	2

4.3 THREATS TO VALIDITY

In this section, we discuss the threats that can harm the validity of our results.

- **Target System** - Our findings are based on just one real system, this is a limiting factor. Nevertheless, even using a toy application as our first case study, we could confirm most of the results with the HelathWatcher case study. Besides that, others factors increase the reliability of our findings: the HealthWatcher is a real information system and was designed with modularity and changeability principles as main driving criteria; it has been extensively used as the subject of studies comparing AO and OO design, because of that, has been considered as a benchmark system for AOP studies. As future work, we plan to involve others systems to collect more evidence of our findings.
- **Techniques/Language** - Our studies use two AOP-based techniques aiming at modularizing use case implementations. Although the techniques are not tied to any specific aspect language, we have just explored AspectJ (on applying the Use Case as Aspect technique) and CaesarJ (on applying the Use Cases as Collaboration technique). This can interlace the results with the language and not with the technique itself. Trying to diminish this effect, we just use basic language resources, implementing the technique in the simplest possible way. Doing that, one can swap to other equivalent aspect language without having to change the concept behind each technique; and by so, expect equivalent results even using a different language.
- **Concerns Selection and Assignment** - Concerns metrics can be unreliable because of the subjectivity inherent in the concern selection and assignment tasks. This can limit the consistency and repeatability of the measurements. Trying to reduce this effect, we use the concern selection and assignment guideline proposed by Eaddy et al. [EZS⁺08] discussed in Section 3.3.1.1. This way, we tie the scope of concerns selection and assignment increasing the chances of producing more repeatable and aligned results.
- **Metrics** - Questions can always be raised arguing if we use a good representative set of internal software attributes affecting the system modularity. However, looking at previous works, we were able to identify relevant attributes that are frequently used on quantitative case studies: coupling, cohesion, size (VS, LOC), and SoC. We also included others metrics that are more specifically related to our goals, like the ones used to quantify traceability and pluggability of use cases implementations. One can add others indicators to adjust the criteria to particular settings on further

case studies. Some of the metrics used in this work has often been questioned, such as, cohesion and coupling [?]. We understand the issues raised on the application of such metrics, mainly, in the AO context. To tackle this problem, our results and conclusions are gathered from a set of metrics rather than just based on one specific.

- **Change Requests** - As we observed, the type of change has strong influence on the studies results and this is a kind of bias that must be taken into account. We were not able to contemplate all kinds and combinations of changes due to time and scope limit of our work. For example, we did not apply changes that deal with refactoring or design patterns aiming at improving the system design. In order to minimize this effect, we had to prioritize changes that would allow us to make an analysis from different points of views. Therefore, we select changes from different scopes and nature regarding the way they affects use case implementations.

4.4 CONCLUSIONS AND DISCUSSION

In this section, we highlight the main outcomes gathered after the analysis of both studies.

1. A use case implementation is a concern that will, likely, be related with others concerns, constituting what we call a *container concern*. For example, in the HW case study, use cases concerns are linked with persistence and distribution. Thus, isolating use case implementations in the HW system degraded the modularization of the persistence and distribution concerns. We find evidence that the isolation of container concerns affect negatively the modularization of concerns that have a dependence relation with them; producing higher degree of scattering of previously well-modularized concerns. For this reason, we reinforce that we just have a fair evaluation on modularity gain if we analyze the results considering a set of concerns that covers most of the code and represents a rationale of the implementation, and not just a specific set of them.
2. Impact on maintenance and evolution tasks has strong relation with types of change. For example, if a system is under constantly addition of new functional requirements (perfective and use case-scoped change), isolating use case implementations brings significant gain on modularity. Although, considering changes in the application environment (adaptive and system-scoped change), modularizing use cases can scatter code of others concerns (e.g., persistence) often leading to a more invasive process of change.

3. As might be expected, modularized use case implementations lead to a system where is easier to plug/unplug a specific use case and where is easier to track use case code creating direct links from requirement specification to use case code assets.
4. The choice of which decomposition must be dominant over others should be guided by the types of change that tend to happen more in a specific context. To achieve good modularity, an important question should be positively answered: is the actual modularity useful to the types of change most likely to happen in the system?

From these observations, we find that modularity is a relative concept. This happens because modularity analysis depends on a set of factors with inherent subjective characteristics. In our studies, for example, it depends not only on the chosen system, metrics and the applied technique, but, also on the selected concerns and types of change; Modifying one of these variables could lead to completely different result regarding the real modularity gain. Indeed, we conclude that it is hard (or even impossible) to find a design (using language constructs that physically separate concerns) where all considered concerns are modularized and do not produce negative side effects to others when exposed to maintenance and evolution tasks. Depending on a combination of factors, such as, the set of concerns under evaluation, the types of change, the application architecture and, even, the developers goals, one design (decomposition) should be chosen over others.

CHAPTER 5

CONCLUSIONS

In this work, we discuss two AOP-based techniques used to modularize use case implementations. We apply different constructs of two major AOP-languages (AspectJ and CaesarJ) to explore how a system behaves regarding modularity issues when its functional use cases are decomposed into separate modules.

Aiming to achieve a systematic evaluation, we develop a GQM model to guide the selection of meaningful metrics. We use these metrics to quantify the impact of modularized use case implementations on quality softwares attributes, which are directly related to important modularity factors (that are most relevant in the context of our work), such as, complexity of source code (comprehensibility), changeability, traceability, and support for parallel development.

We run two cases studies based on maintenance and evolution tasks in order to answer our research questions as highlighted below:

- Q1. Does modularity of use case implementations reduce modularity of non use case concerns?
- Q2. Does modularity of use case implementations improve overall maintenance and evolution?
- Q3. Which modularity dimensions benefit from modularized use case implementations?

In the first study, we apply the Use Case as Aspect technique (discussed in Section 3.2.1) and the Use Case as Pluggable Collaboration technique (discussed in Section 3.2.2) to isolate all use case implementations in the target system. We opted to use a toy application in order to consolidate the techniques, the assessment and evaluation process. In the second study, we use the HealthWather system (considered a benchmark system for AOP studies) and apply the Use Case as Aspect technique.

Based on the studies results we answer our research questions and conclude the following:

1. Modularity of use case implementations affects negatively the modularity of concerns directly related to it. For example, if persistence is a concern present in

the system, if you modularize use cases code you will spread the persistence code throughout the use case modules.

2. Modularity is a relative concept. We are not able to say that by modularizing use case implementations we improve overall maintenance and evolution. Modularity gains depends on a series of factors beyond the chosen system, the used technique and metrics. Among them, we can emphasize the *type of change factor*. For example, if a system suffers more requirements changes than environment (e.g. the way data is persisted in the system), it will benefit more if its use case implementations are modularized.
3. Considering that use cases drive the whole development process (use case driven development context), traceability and pluggability are the modularity dimensions that always benefits when systems have their use case implementations modularized and consolidated as the main form of decomposition.

5.1 RELATED WORK

The investigation described in this dissertation is mainly related to use case implementations modularity techniques and empirical studies in aspect-orientation.

5.1.1 Use case Implementations Modularity Techniques

Jacobson [Jac03] presented the Use Case Slice technique where he used AOP to separate use cases in Design and Implementation. He made a connection between AOP and Use Cases. He defined two types of use cases: (i) peer use case— is the use case that exists by itself, that is, it does not depend on others use cases; and extension use case—(ii), is a use case which depends on a base use case, extending or generalizing its behavior. He argued that aspect oriented programming provides the missing link in order to keep the modularity of use cases in the design and implementation. He used static constructs, like intertype declarations, to implement peers use cases and dynamic constructs, like pointcuts and advices, to implement extension use cases. The Use Case as Aspect technique is based on this work, but with the following differences: (i) we do not make distinction of use case types, namely peers and extensions;(ii) we just apply the technique in the implementation level (code); and (iii) we just apply the technique on functional use cases; we do not create use case modules to nonfunctional use cases like persistence, for example.

Bhole et al. [BL08] used AspectJ and its DemeterJ extension, called DAJ [SL] to maintain the use case structure from requirements to implementation. They used the

same concept introduced by Jacobson relating static AOP constructs to the implementation of peers use cases and dynamically constructs to the implementation of extension use cases. They presented a simple experiment where they evaluate use case modularity in the context of AOP. They conclude, based on a qualitative evaluation, that modularizing use case implementations improve the overall system modularity. Their observations differs from ours as we state that modularity is relative to different point of views.

Herman et al. [HHM04] presented the concepts for a process which can be applied from elicitation to implementation based on constructs of an aspect language called Object/Team [Her07]. This process consistently refine use case aspects. They demonstrated with an example how the use cases are transformed from requirement to implementation. They use collaborations with roles and classes to modularize use case implementations. It is an approach similar to the one used in the Use Case as Pluggable Collaboration technique. Our technique differs from this one as we focus only at the implementation level and do not use roles while implementing the use case collaborations.

5.1.2 Empirical Studies in Aspect-Orientation

Inspired by the need to consolidate AOP as an efficient mechanism to modularize cross-cutting concerns, an increasing number of qualitative and quantitative assessments has been performed in the last five years or so. Most of them concentrate their efforts on the AspectJ language and do not consider use case implementations as the focus of analysis.

Some case studies analyze how AOP promotes superior modularity and separation of concerns in the implementation of traditional crosscutting concerns, such as, distribution [SLB02, SBL06] and persistence [RC03, SBL06]. However, they focus on how AspectJ mechanisms are used to separate these usual crosscutting concerns; they do not quantify the positive and negative effects of AO techniques in the presence of widely scoped changes. They just focus on their specific concerns of interest and do not analyze the impact of such modularization on other important concerns of the target systems.

Garcia et al. [SGC⁺03] defined a metric suite contemplating Separation of Concerns (SoC) and adapted CK metrics. They apply the suite to evaluate how AOP mechanisms provide better separation of specific concerns like design patterns [GSF⁺06], and exception handling [CFCF⁺06]. They did not evaluate the impact of these specific modularization on others systems concerns as well.

Kulesza et al. [KSG⁺06] evaluated the scalability of the AspectJ implementation by performing some functional changes (addition of use cases) to the HealthWatcher system (the same target system used in our work). They use the metric suite defined by Garcia et al. In the evaluation of separation of concerns, they just considered persistence, distribu-

tion and concurrency as concerns. They analyze reuse, scalability and stability of aspect modules showing some advantages and disadvantages of using aspects. Differently from our work, they did not consider use cases as concerns and they did not correlate their results with types of changes (as they only use one type: addition of new functionalities) or other factors.

Greenwood et al. [GBF⁺] presented a quantitative case study that also evolves the HealthWatcher system. They used the same metric suite defined by Garcia and colleagues contemplating SoC, coupling, cohesion and size metrics. They evaluate AspectJ and CaesarJ implementations assessing, what they call, the design stability of OO and AO implementations. Similar to our approach, their evaluation are based on system changes that are typically performed during software maintenance tasks. They ranged from successive refactorings to introduction of new functionalities. The study include an analysis of the application in terms of modularity, change propagation, concern interaction, and identification of ripple-effects. They identified positive and negative effects of using aspects, but they also do not consider use case implementations as concerns.

In this work [VCFS], Valente et al. argued that quantification is one of the most important parameter to consider regarding modularity benefits in the use of aspects. They run three case studies and provide qualitatively and quantitatively evaluation based on a suite of separation of concerns metrics. They state that the most favorable uses of aspects happen when their code relies extensively on quantified statements. They conclude that on having quantification, aspects better contribute to separation of concerns, since the previous duplicated and tangled code related to the implementation of a crosscutting concern is now confined in separate aspects. In our work, we cannot assume that better separation of concerns imply just on having quantification. For example, we show that although a use case implementation do not benefit from quantification (because of its heterogeneous nature), in some situations, we improve the overall system's modularity by modularizing it. So, in our context, quantification, by itself, does not indicate if the modularization of a specific concern will fail or succeed.

5.2 FUTURE WORK

We have evaluated the AOP-based techniques in a single real web-based system: the HealthWatcher. An interesting future work would be to execute others case studies in the same domain, and also in applications from different domains, in order to collect more evidence about our findings and gain confidence that the results presented in this dissertation can be generalized.

An interesting approach to software modularity has been gaining a lot of attention

recently: tools providing views of systems concerns. The concerns can be arranged to enable different views (different forms of decomposition) to meet a variety of developers needs. In contrast with our linguistic (programmatic) approach, a tool-based solution is the one in which a tool or IDE provides use case modularity in a virtual manner, that is, the programming environment is responsible for enabling views of the modularization unit.

Hon et al. [HK06], introduced the idea of fluid AOP which is a technique that supports modularization of crosscutting concerns in IDE, it provides temporary views that modularizes current concerns. These views (aspects) can be generated each time it is needed, or can be stored as a meta object.

Virtual Separation of Concerns/Features tools [CCWY03, Kas07, RB10] follows the idea of fluid AOP and have been calling attention as a promising approach to enable this new kind (virtual) of modularity. Most of their recent success is due to the fact they can provide virtual separation of features, which seems to be a useful approach to the Software Product Line context.

Likewise, Program Navigation Analysis Tools [RM] is another category of great interest that could be used to enable modularization of use case implementations. A great number of approaches have been proposed to monitor and analyze the actions of developers as they perform software development tasks [KM05, RM03]. The results of such analysis often represent a subset of the code of interest to a developer.

Therefore, another possible work is to investigate how these new approaches to modularity behave when used to modularize use case implementations. It will be interesting ,as future work, to repeat our experiments using one of these tools and compare with our linguistic (AOP) based solution.

Finally, another interesting work is to do a rigorous evaluation on the metrics used in this work. Also, based on the evaluation results, we can propose new ones and also repeat the studies applying the new set of metrics thus comparing with our actual results.

BIBLIOGRAPHY

- [AGMO06] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006.
- [aop] Aop metrics tool. <http://aopmetrics.tigris.org/>.
- [AWB⁺93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. *Object-Based Distributed Programming*, pages 152–184, 1993.
- [Azu01] M. Azuma. SQuaRE: the next generation of the ISO/IEC 9126 and 14598 international standards series on software product quality. In *ESCOM (European Software Control and Metrics conference)*, pages 337–346, 2001.
- [BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311. ACM, 1990.
- [BCR94] V.R. Basili, G. Caldiera, and H.D. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 1:528–532, 1994.
- [BL08] M. Bhole and K. Lieberherr. Use Case Modularity using Aspect Oriented Programming. *Relation*, 10(1.57):9687, 2008.
- [BMR⁺08] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns, volume 1*. Wiley India Pvt. Ltd., 2008.
- [BR00] K.H. Bennett and V.T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the conference on The future of Software engineering*, page 87. ACM, 2000.
- [BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

- [BS03] K. Bittner and I. Spence. *Use case modeling*. Addison-Wesley Professional, 2003.
- [CCWY03] M.C. Chu-Carroll, J. Wright, and A.T.T. Ying. Visual separation of concerns through multidimensional program storage. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 188–197. ACM, 2003.
- [CFCF⁺06] F. Castor Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C.M.F. Rubira. Exceptions and aspects: the devil is in the details. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, November*, pages 05–11. Citeseer, 2006.
- [CKM94] SR Chidamber, CF Kemerer, and C. MIT. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.
- [CL02] I. Crnkovic and M.P.H. Larsson. *Building reliable component-based software systems*. Artech House Publishers, 2002.
- [con] Concerntagger eclipse plugin. <http://www.cs.columbia.edu/~eaddy/concern-tagger/>.
- [CT04] M. Ceccato and P. Tonella. Measuring the effects of software aspectization. In *Workshop on Aspect Reverse Engineering*. Citeseer, 2004.
- [Dij69] E.W. Dijkstra. Notes on structured programming. 1969.
- [EAM07] M. Eaddy, A. Aho, and G.C. Murphy. Identifying, assigning, and quantifying crosscutting concerns. In *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, page 2. IEEE Computer Society, 2007.
- [Ede93] D.V. Edelstein. Report on the IEEE STD 1219–1993-standard for software maintenance. *ACM SIGSOFT Software Engineering Notes*, 18(4):95, 1993.
- [Erl00] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
- [EZS⁺08] M. Eaddy, T. Zimmermann, K.D. Sherwood, V. Garg, G.C. Murphy, N. Nagappan, and A.V. Aho. Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering*, pages 497–515, 2008.

- [GBF⁺] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, et al. On the impact of aspectual decompositions on design stability: An empirical study. *ECOOP 2007-Object-Oriented Programming*, pages 176–200.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.
- [GJ05] I. Godil and H.A. Jacobsen. Horizontal decomposition of Prevayler. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, page 100. IBM Press, 2005.
- [GSC⁺04] A. Garcia, C. Sant'Anna, C. Chavez, V.T. da Silva, C.J.P. de Lucena, and A. von Staa. Separation of concerns in multi-agent systems: An empirical study. *Software Engineering for Multi-Agent Systems II*, pages 343–344, 2004.
- [GSF⁺06] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. *Transactions on Aspect-Oriented Software Development I*, pages 36–74, 2006.
- [HC01] G.T. Heineman and W.T. Councill. *Component-based software engineering: putting the pieces together*. Addison-Wesley USA, 2001.
- [Her07] S. Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.
- [Her09] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264, 2009.
- [HHM04] S. Herrmann, C. Hundt, and K. Mehner. Mapping use case level aspects to object teams/java. In *OOPSLA Workshop on Early Aspects*, 2004.
- [HK06] T. Hon and G. Kiczales. Fluid aop join point models. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, page 713. ACM, 2006.
- [Jac03] I. Jacobson. Use cases and aspects-working seamlessly together. *Journal of Object Technology*, 2(4):7–28, 2003.

- [JB00] I. Jacobson and S. Bylund. *The road to the unified software development process*. Cambridge Univ Pr, 2000.
- [JN04] I. Jacobson and P.W. Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.
- [Kas07] C. Kastner. CIDE: Decomposing legacy applications into features. In *Proceedings of the 11th International Software Product Line Conference (SPLC), second volume (Demonstration)*, pages 149–150. Citeseer, 2007.
- [KHH⁺01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *ECOOP 2001-Object-Oriented Programming*, pages 327–354, 2001.
- [KLL⁺97] G. Kiczales, C.V. Lopes, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *In Proceedings European Conference on Object-Oriented Programming*, 1997.
- [KLL⁺02] G.J. Kiczales, J.O. Lamping, C.V. Lopes, J.J. Hugunin, E.A. Hilsdale, and C. Boyapati. Aspect-oriented programming, October 15 2002. US Patent 6,467,086.
- [KM05] M. Kersten and G.C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168. ACM, 2005.
- [KSG⁺06] U. Kulesza, C. Sant’Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *22nd IEEE International Conference on Software Maintenance, 2006. ICSM’06*, pages 223–233, 2006.
- [Lie96] K.J. Lieberherr. *Adaptive object-oriented software: the demeter method*. PWS Boston, 1996.
- [LZ74] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59. ACM, 1974.
- [Mey88] B Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

- [MMP89] O.L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406. ACM, 1989.
- [MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.
- [OT02] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, pages 293–324, 2002.
- [Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1058, 1972.
- [PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [PCW84] DL Parnas, PC Clements, and DM Weiss. The modular structure of complex systems. In *Proceedings of the 7th international conference on Software engineering*, page 417. IEEE Press, 1984.
- [PI87] R.S. Pressman and D. Ince. *Software engineering: a practitioner's approach*. McGraw-Hill New York, 1987.
- [Pol57] G. Polya. *How to solve it* Princeton, 1957.
- [RB10] Pacheco H Teixeira L Ribeiro, M and P Borba. Emergent feature modularization. In *Proceedings of Onward! 2010*. ACM, 2010.
- [RC03] A. Rashid and R. Chitchyan. Persistence as an Aspect. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, page 129. ACM, 2003.
- [RM] M.P. Robillard and G.C. Murphy. Program navigation analysis to support task-aware software development environments. In *Proceedings of the ICSE Workshop on Directions in Software Engineering Environments*, pages 83–88. Citeseer.

- [RM03] M.P. Robillard and G.C. Murphy. Automatically inferring concern code from program investigation activities. 2003.
- [SBL06] S. Soares, P. Borba, and E. Laureano. Distribution and persistence as aspects. *Software Practice and Experience*, 36(7):711, 2006.
- [SGC⁺03] C. Sant’Anna, A. Garcia, C. Chavez, C. Lucena, and A. Von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings of Brazilian Symposium on Software Engineering*, pages 19–34, 2003.
- [SL] J. Sung and K. Lieberherr. DAJ: A Case Study of Extending AspectJ. Technical report, Northeastern University Technical Report NUCCS-02-16, 2002. Available at: <http://www.ccs.neu.edu/research/demeter/biblio/-DAJ1.html>.
- [SLB02] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. *ACM SIGPLAN Notices*, 37(11):174–190, 2002.
- [Som06] I. Sommerville. *Software Engineering*. Addison-Wesley, 2006.
- [VCFS] M.T. Valente, C. Couto, J. Faria, and S. Soares. On the benefits of quantification in AspectJ systems. *Journal of the Brazilian Computer Society*, pages 1–14.
- [VdLSR07] F. Van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering*. Springer, 2007.