Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

DERIVING REFACTORINGS FOR ASPECTJ

Leonardo Cole Neto

DISSERTAÇÃO DE MESTRADO

Recife
28 de fevereiro de 2005

Universidade Federal de Pernambuco
Centro de Informática

Leonardo Cole Neto

DERIVING REFACTORINGS FOR ASPECTJ

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Prof. Paulo Henrique Monteiro Borba*

Recife
28 de fevereiro de 2005

Cole Neto, Leonardo
Deriving refactorings for aspectJ / Leonardo Cole Neto. – Recife : O Autor, 2005.
viii, 85 folhas : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2005.

Inclui bibliografia e apêndice.

1. Engenharia de software – Refactoring – Técnicas de desenvolvimento. 2. Linguagem de programação (Orientada a aspectos). 3. AspectJ (Linguagem de programação de computador) 4. Leis de programação (Métodos formais). I. Título.

004.415.3 CDU (2.ed.) UFPE
005.16 CDD (22.ed.) BC2005-411

Para minha namorada, Danielly, meus pais e irmãos

AGRADECIMENTOS

Agradeço primeiramente aos meus pais e irmãos, pois sem o carinho e suporte de minha família não poderia ter realizado mais esta jornada.

À Danielly, minha namorada e companheira, que sempre me incentivou e ajudou em momentos onde o cansaço e a falta de estímulo quase predominaram. Agradeço ainda por aturar minha falta de paciência em certas ocasiões.

Ao meu grande amigo Teixeira, pelo companheirismo e ajuda em diversas ocasiões e também por proporcionar muitos momentos divertidos, sem os quais vida no CIn certamente seria mais difícil.

A Paulo Borba, um excelente orientador. Capaz de estimular e contribuir com muitas idéias para sempre melhorar meu trabalho. Sua orientação foi, sem dúvida fundamental para o meu engrandecimento pessoal como pesquisador.

A Vander Alves e Pedro Matos, por compartilharem seu trabalho comigo, deixando-me utilizar o resultado de minha pesquisa como uma pequena base para seu trabalho.

Ao professor Alexandre Mota, pela ajuda com parte da atividade formal contida nesta dissertação.

Ao professor Augusto Sampaio e ao colega Eduardo Piveta, por me ajudarem a iniciar minha produção científica com um trabalho resultante de um esforço paralelo ao mestrado.

A Vander, Rohit e Tiago Massoni, por me ajudarem com excelentes comentários para melhorar consideravelmente a apresentação desta dissertação. Agradeço também por sua amizade durante os últimos anos.

A Ramnivas Laddad, por bons comentários que enriqueceram o conteúdo deste trabalho.

Aos outros membros do Software Productivity Group, por ótimas discussões e trocas de idéias que melhoraram bastante minha visão crítica sobre outros trabalhos.

Finalmente, agradeço a meus amigos Sérgio, Trinta, Tiago Santos e Márcio, por proporcionarem vários momentos de descontração durantes esses dois anos.

RESUMO

Refactoring tem sido muito útil na reestruturação de programas orientados a objetos. Esta técnica pode proporcionar benefícios similares aos programas orientados a aspectos. Além disso, refactoring pode ser uma técnica interessante para introduzir aspectos em uma aplicação existente, orientada a objetos.

No intuito de explorar os benefícios proporcionados pelos refactorings, desenvolvedores de aplicações orientadas a aspectos estão identificando transformações comuns para tais aplicações, em sua maioria, para a linguagem AspectJ, uma linguagem orientada a aspectos de propósito geral que estende Java. No entanto, estas transformações não possuem suporte para garantir que preservam o comportamento externo da aplicação. Tal propriedade garante que as transformações são de fato refactorings.

Este trabalho foca neste problema e introduz leis de programação para AspectJ que podem ser usadas para derivar transformações que preservam comportamento (refactorings) para um subconjunto desta linguagem. Leis de programação definem equivalência entre dois programas, desde que algumas restrições sejam respeitadas. Nosso conjunto de leis não somente define como introduzir ou remover construções de AspectJ, como também como reestruturar aplicações nesta linguagem. Aplicando e compondo as leis, pode-se mostrar que uma transformação qualquer, envolvendo AspectJ, é de fato um refactoring. Leis são apropriadas para isso pois são bem mais simples do que a maioria dos refactorings. Comparando com refactorings, as leis envolvem transformações localizadas e somente uma construção da linguagem por vez, além de seram bi-direcionais. As leis formam uma base para a definição de refactorings com uma certa confiança de que estes preservam comportamento.

Nós avaliamos as leis de duas formas. A primeira utiliza as leis para derivar refactorings já existentes na literatura. Isto ajuda a definir com mais precisão as precondições associadas a estes refactorings, alem de verificar se eles preservam comportamento. A segunda forma de avaliação utiliza as leis e alguns refactorings derivados destas para reestruturar duas aplicações Java. A implementação de interesses transversais nestas aplicações é reestruturada utilizando construções de AspectJ para tornar tal comportamento modular. Isto ilustra que as leis podem também ser úteis para transformar aplicações orientadas a objetos em aplicações orientadas a aspectos.

Palavras-chave: Refactoring, Desenvolvimento de Software Orientado a Aspectos, Separação de Interesses Transversais, Leis de Programação

ABSTRACT

Refactoring has been quite useful for restructuring object-oriented applications. It can bring similar benefits to aspect-oriented applications. Moreover, refactoring might be a useful technique for introducing aspects to an existing object-oriented application.

In order to explore the benefits of refactoring, aspect-oriented developers are identifying common transformations for aspect-oriented programs, mostly in AspectJ, a general purpose aspect-oriented extension to Java. However, they lack support for assuring that the transformations preserve behaviour and are indeed refactorings.

This dissertation focus on that problem and introduces AspectJ programming laws that can be used to derive or create behaviour preserving transformations (refactorings) for a subset of this language. Programming laws define equivalence between two programs, given that some conditions are respected. Our set of laws not only establishes how to introduce or remove AspectJ constructs, but also how to restructure AspectJ applications. By applying and composing those laws, one can show that some transformation involving AspectJ is a refactoring. The laws are suitable for that because they are much simpler than most refactorings. Contrasting with refactorings, they involve only localized program changes, and each one focus on a specific language construct. The laws form a basis for defining refactorings with some confidence that they preserve behaviour.

We evaluate our laws by showing how they can be used to derive several refactorings proposed in the literature. This helps to more precisely specify the preconditions and code changes associated with those refactorings, and gives more confidence that they preserve behaviour.

Besides deriving refactorings, we evaluate our laws by restructuring two Java applications. The implementation of crosscutting concerns in those applications are restructured so that they are modularized with AspectJ constructs. This illustrates that the laws might also be useful for transforming Java applications into AspectJ applications.

Keywords: Refactoring, Aspect-Oriented Software Development, Separation of Concerns, Programming Laws

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The need to develop quality software has increased the use of object-oriented [37, 9] techniques in industry, augmenting levels of reuse, maintainability, productivity and extensibility. Refactoring [19, 40, 44] has been quite useful for restructuring object-oriented applications. It is a technique used for restructuring programs with the purpose of increasing code quality. Refactorings generally represent transformations that may be applied to code given that some restrictions are respected. Those restrictions help to ensure that the refactoring preserves behaviour and maintains program consistency.

However, the object-oriented paradigm has known limitations [41, 42]. For instance, there are problems related to the implementation of concerns that generally spread over many different modules of the system (code scattering); this code may even not be related to the behavior implemented by the module (code tangling). Those are called crosscutting concerns and are often derived from non-functional requirements. Examples of crosscutting concerns are transaction control, security, persistence and distribution.

Although some of these limitations may be avoided through the use of Design Patterns [11, 20], this generally increases complexity by adding more classes, levels of hierarchy and indirection. Aspect-Oriented Software Development [18] is an emerging paradigm that comes to complement object-orientation. It helps developers to solve problems related to modularity that could not be properly addressed by the later. Aspect-orientation proposes techniques to obtain better software modularity in practical situations where object-oriented development and associated design patterns are not appropriate.

Aspect-orientation implements crosscutting concerns as aspects. Each aspect defines pointcuts and advices. The pointcuts denote singular points on the system execution flow where an advice can execute; these points are called join points. The advices specify the code to execute and when it should execute. Hence, aspect-orientation increases modularity providing means to implement crosscutting concerns in a modular structure and apart from the code that implements core functionality.

Despite the benefits of aspect-oriented programming, techniques aiming to solve object-oriented programming issues can also bring benefits to aspect-oriented applications as well. For instance, refactoring can be used to restructure aspect-oriented applications in order to increase code quality. In addition, refactoring might be a useful technique for introducing aspects to an existing object-oriented application. Therefore, it is possible to restructure an object-oriented application to modularize crosscutting concerns with aspect-oriented features.

In order to explore the benefits of refactoring, aspect-oriented developers are identifying common transformations for aspect-oriented programs [31, 32, 23, 28], mostly in AspectJ [29], a general purpose aspect-oriented extension to Java [22]. However, most of the refactorings defined so far lack support for assuring that the transformations preserve behaviour and are indeed refactorings. Current approaches for refactoring usually rely on tests to ensure that they preserve behaviour.

This dissertation focus on that problem and introduces AspectJ programming laws that can be used to derive or create behaviour preserving transformations (refactorings) for a subset of this language. Programming laws [27] define equivalences between two programs, given that some conditions are respected. Programming laws define algebraic rules that can generally be proved correct. Thus, by defining refactorings in terms of aspect-oriented programming laws, one can prove that aspect-oriented refactorings indeed preserve behaviour.

The laws are suitable to that because they are much simpler than most refactorings. Furthermore, the laws involve only localized program changes, and each one focus on a specific language construct. The considered changes are localized because they generally affect one single class or aspect, whereas global refactorings may cause changes to the entire application. Additionally, it is easier to verify whether a simple transformation, dealing with one specific language construct, preserves behaviour. This allows an intuitive understanding of the transformation.

Although refactorings are transformations to increase quality, the laws may decrease quality. This is possible because laws might be applied as part of a bigger strategy (refactoring) that increases quality. Besides, as the laws relate two equivalent programs, they are bi-directional. Consider the code before and after the transformation as two distinct sides of an equality. This equation defines two transformations, one substituting the left hand side for the right hand side and one in the opposite direction. That is, the laws not only can transform code according to its intention (left → right), but can also perform the reverse transformation (right → left).

Our set of laws not only establishes how to introduce or remove AspectJ constructs, but also how to restructure AspectJ applications. The definition of the laws as a bi-directional transformation states how to introduce and how to remove AspectJ constructs at the same time. Furthermore, the laws are regarded as single steps to restructure the code. Moreover, applying several laws (several steps) may completely restructure an AspectJ application. By composing those laws, one can show that an AspectJ transformation[1] is indeed a refactoring. The laws form a basis for defining refactorings with some confidence that they preserve behaviour.

The laws are also useful to guide the development of aspect-oriented refactoring tools. The simplicity of the laws support an easier implementation to provide tool assistance for applying refactorings. Further, a tool implementing the laws can also implement the complex refactorings that can be derived from them. Another characteristic of the laws is that they may reveal details about the language semantics. An experienced developer may learn intrinsic language issues by understanding the laws. Therefore, this deeper understanding of the language semantics may help the developers produce software with less errors. This characteristic may also help teaching and learning the language. The laws can be used to show simple examples and illustrate one language construct at a

[1] Any language transformation involving AspectJ constructs, for instance, Java to AspectJ transformations and AspectJ to AspectJ transformations

time. The laws can also be used to justify a compilation strategy because the application of the laws to remove AspectJ constructs is very similar to the transformation applied by the compiler to weave classes and aspects.

Despite its usefulness, the laws need to be proved correct. We provide a formal argumentation that some laws preserve behaviour. We use an existing aspect-oriented semantics for Method Call Interception (MCI) [33] where it is possible to represent some AspectJ constructs, and some of our laws. We provide an equivalence notion stating in which conditions two MCI programs behave the same. We discuss the soundness of one law by interpreting it according to the MCI semantics and verifying it with our equivalence notion. We also discuss how other laws can analogously be proven. However, we can not prove all laws with this approach. For the remaining laws, we rely on their simplicity and intuitiveness to argue informally about their correctness.

We evaluate the practical utility of our laws by showing how they can be used to derive several refactorings proposed in the literature [31, 23, 28]. This illustrates how the laws are useful and helps to more precisely specify the preconditions and code changes associated with those refactorings. Furthermore, the representation of those refactorings as a composition of our laws gives more confidence that they preserve behaviour.

Besides deriving refactorings, we evaluate our laws by restructuring two object-oriented applications to take advantage of aspect-oriented features. The implementation of crosscutting concerns in those applications are restructured so that they are modularized with aspect-oriented constructs. This illustrates that the laws might also be useful for transforming Java applications into AspectJ applications.

The major contributions of this work are the following:

- Definition of aspect-oriented programming laws that are useful for creating aspect-oriented refactorings and formally deriving existing ones to increase the confidence that they preserve behaviour.

- Definition of an equivalence notion stating that two programs have the same behavior.

- Formal argumentation about the soundness of some laws, according to an existing aspect-oriented semantics, and the defined equivalence notion.

- Derivation from the laws of several existing refactorings proposed in the literature, proving that they preserve behaviour. This also pointed out some limitations to the set of defined laws.

- Usage of the laws and derived refactorings to modularize crosscutting concerns from object-oriented applications.

The remainder of this dissertation is organized as follows:

- **Chapter** 2 discusses the AspectJ language in detail. It also shows limitations and the subset of this language used in this dissertation.

- **Chapter** 3 introduces our laws showing their structure, preconditions and intent. This chapter also provide a formal argumentation to show that some of our laws are sound. Part of the results showed in this chapter were already published [14, 15]. Section 4.1.7 resulted from a collaboration work with Alves and Matos [6, 4]. Alves uses our laws as a formal basis in his Phd thesis, which deals with restructuring product lines.

- **Chapter** 4 then discusses the evaluation of our laws. We use two approaches to evaluate the laws: first we derive several refactorings from our laws, and then we restructure two case studies using our laws and the derived refactorings. Part of the results showed in this chapter were already published [14, 13].

- **Chapter** 5 discusses our conclusions, related work and future work.

- **Appendix** A is a complement to Chapter 3. It shows all laws not presented in the referred chapter.

CHAPTER 2

ASPECTJ

Aspect-oriented languages support the modular definition of concerns which are generally spread throughout the system and tangled with core features. Those are called crosscutting concerns and their separation promotes the construction of a modular system, avoiding code tangling and scattering.

AspectJ [29] is an aspect-oriented [18] extension to Java [22]. Programming with AspectJ involves both aspects and classes to separate concerns. Concepts which are well defined with object-oriented [37, 9] 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 [26]. Thus, an AspectJ application is composed of both classes and aspects. Therefore, each AspectJ aspect defines a functionality that affects different parts of the system.

Aspects may define pointcuts and advices. The pointcuts describe a set of points during the program execution flow where a piece of code should execute. The code to be executed is declared as an advice.

This chapter describes the AspectJ language in detail. We consider AspectJ version 1.2 and focus on the mechanisms necessary to understand further discussion in this dissertation. More details about this language can be found in the AspectJ Programming Guide [46].

2.1 POINTCUT

The pointcuts define sets of points on the system execution flow where a piece of code can execute; these points are called join points. Join points may be method calls, method execution, field access, exception handling and static initialization. AspectJ provides several kinds of join points:

Method call - when a method is called, not including super calls of non-static methods.

Method execution - when the body of code for an actual method executes.

Constructor call - when an object is built and that object's initial constructor is called (i.e., not for super or this constructor calls).

Constructor execution - when the body of code for an actual constructor executes, after its this or super constructor call.

Static initializer execution - when the static initializer for a class executes.

Object pre-initialization - before the object initialization code for a particular class runs. This encompasses the time between the start of its first called constructor and the start of its parent's constructor.

Object initialization - when the object initialization code for a particular class runs. This encompasses the time between the return of its parent's constructor and the return of its first called constructor.

Field reference - when a non-constant field is referenced.

Field set - when a field is assigned to.

Handler execution - when an exception handler executes.

Advice execution - when the body of code for a piece of advice executes.

The difference between `call` and `execution` join points is that the first executes before even evaluating the join point parameters. In addition, the `call` join point executes even if the captured join point does not execute because of a runtime exception for instance.

As we mentioned, pointcuts describe sets of join points. In order to define pointcuts, we use the pointcut designators. For instance, `call`(*Signature*) identifies a method or constructor call; whereas `execution`(*Signature*) identifies a method or constructor execution. Pointcut expressions can also expose context to be used by the advices. It is possible to expose the executing object, the arguments of a method, and, in some cases, the object in which a join point is being called. There are specific pointcut designators to bind the exposed context: `this`, `args` and `target`, respectively. Table 2.1 shows valid examples of pointcut designators. We use a simple application for drawing figure elements to build the examples. This application has a `Display` where `FigureElements` can be painted at some coordinate (`Point`).

It is possible to declare anonymous or named pointcuts. Anonymous pointcuts are expressions used directly with the advice declaration. Moreover, we can declare a named pointcut using the `pointcut` keyword from AspectJ. The following example declares a named pointcut called `vectorConstructorCalls(Vector)`, which captures execution of constructors of class `Vector`. This pointcut also exposes the vector object being created (`this(vector)`). In this case, advices may use the name of the pointcut instead of an anonymous pointcut expression.

```
pointcut vectorConstructorCalls(Vector vector) :
                 execution(* Vector.new(..)) && this(vector) ;
```

2.2 ADVICES

The advices specify the code to execute and when it should be executed. The choices are `before`, `after` or instead (`around`) of the captured join point. The `before` advice executes some commands before the captured join point. Following we show an example that prints a message before any call to the `Vector` constructor.

`execution`(`void` *Point.setX*(`int`))	When the *setX* method body, from class *Point*, with an int parameter, executes
`call`(`void` *Point.setX*(`int`))	When the *setX* method is called
`this`(*Point*)	When the object currently executing (i.e. this) is of type *Point*
`target`(*FigureElement*)	When the target object is of type *FigureElement*
`handler`(*PointOutOfBoundsException*)	When an exception handler executes (the catch block for *PointOutOfBoundsException*)
`args`(`int`)	When the executing or called method has an `int` parameter
`within`(*Display*)	When the executing code belongs to class *Display*
`cflow`(`call`(`void` *Display.paint*()))	When the join point is in the control flow of a call to a *Display*'s no-argument *paint* method

Table 2.1. Pointcut Designator Examples.

```
before(): call(* Vector.new(..)) {
  System.out.println("building a vector");
}
```

The `after` advice has three forms: `after returning` is used to execute the advice code only when the join point executes successfully, it may also expose the returning value; `after throwing` is used to execute an advice only if the captured join point raises an indicated exception, it may also expose the raised exception; the last case is the `after` advice, it executes after the captured join point no matter how it ends. The following example shows an `after returning` advice, which captures calls to the method `remove(int)` of the `Vector` class and exposes its parameter (`index`) and the returning value (`result`) to be used in the advice body. The `args` designator indicates that the `index` parameter is in fact the argument of the captured join point.

```
after(int index) returning(Object result):
    call(Object Vector.remove(int)) &&
    args(index) {...}
```

The `around` advice can execute some commands before and after the join point, using or not a call to `proceed`, which allows executing the join point itself. This way, the join point can be completely overridden if `proceed` is not called. The `proceed` command can also change the values of the context exposed to the advice. For instance, it may change an argument value on a method call. Next, we show an example where we start a timer before method `remove(int)` of the `Vector` class starts executing, and prints the total elapsed time after its end.

```
Object around(int index): execution(Object Vector.remove(int)) && args(index) {
   long time = System.currentTimeMillis();
   Object o = proceed(index);
   System.out.println("Elapsed time (ms): "+ System.currentTimeMillis() - time);
   return o;
}
```

One subtle issue about advices arises when two distinct advices affect the same join point: which one should execute first. The AspectJ semantics indicates that if two advices declared within the same aspect affect the same join point, there are two cases to consider: if either one is `after`, the one declared later has precedence; in every other case, the advice declared first has precedence. Note that this rule implies in a cyclic redundance error when the two considered advices are `after` and `before` respectively. According to the rule, the `after` advice should execute first, but it can not execute ahead of the `before` advice. This yields a compiler error.

2.3 ASPECTS

The aspect is a first class entity introduced by AspectJ. Aspects define pointcuts and advices. Besides pointcuts and advices, an aspect may declare fields and methods similarly to classes. Those are generally auxiliary to the behaviour implemented by the aspect. Moreover, the aspect may also have inter-type declarations, which allow us to change classes adding new fields, methods and changing their hierarchy. An aspect may also turn a checked exception into an unchecked one. This enables us to handle exceptions related to crosscutting concerns within the aspect.

AspectJ also provides inheritance for aspects. An aspect my be abstract and define abstract pointcuts. In addition, another aspect may extend the abstract one and provide the concrete definition for abstract pointcuts. This feature allows the definition of reusable aspects. It is possible for an aspect to extend from a class or implement an interface, but the opposite in not allowed. In order to declare an aspect, we use the `aspect` keyword, similar to a class declaration as we can see from the following example.

```
public aspect MyAspect {...}
```

Aspects are not initialized like classes, AspectJ controls how aspects are instantiated. Aspects have just one instance by default, similarly to the Singleton design pattern [20]. This implies that declared fields are shared among advices for every execution. However, AspectJ provides mechanisms to change the instantiation rule as `perthis`, `pertarget`, `percflow`, and `percflowbelow`. For instance, the `perthis` construct states that one instance of the aspect will be created for each executing object captured by a pointcut expression. The following example shows how this construct is used. In this case, there will be one instance of the aspect for every execution of a join point within a Facade class.

```
public aspect MyAspect perthis(execution(* Facade.*(..))) {...}
```

Another important AspectJ feature is the privileged modifier. It allows the aspect to access private members of classes. It is important to notice that the privileged modifier does not change the set of join points captured by the aspect. This modifier only enables the code inside advices to access private members of classes. The privileged modifier appears after the visibility modifier on the aspect declaration.

We already discussed precedence of advices within the same aspect. However, precedence of advices among aspects must also be considered. AspectJ provides the declare precedence construct, which declares a list of aspects in order of precedence. If some aspect appears earlier than other aspect in some declare precedence list, then all advice in the first aspect has precedence over all advice in the second aspect when they are on the same join point. An aspect also has precedence over its super aspect, unless the opposite is stated in a declare precedence list. If none of those rules are matched, it is undefined which aspect has precedence. Following we show an example of the declare precedence construct.

```
declare precedence: MainAspect, OtherAspect;
```

2.4 INTER-TYPE DECLARATIONS

Besides modifying the dynamics of the program, an aspect can modify its static structure. Static modification includes the mechanisms provided by inter-type declarations: introduction of new fields and methods to existing classes and interfaces. For instance, we may want to specify an identifier for every instance of the Vector class. So it is necessary to add a new integer field (line 1) with an accessor method (line 2).

```
  1: int Vector.id;
  2: int Vector.getID(){ return id; }
```

We may also modify the class hierarchy. We can indicate that one class extends from another, or implements a given interface. Line 1, on the next example, indicates that MyList implements interface List. Line 2 indicates that MyVector extends Vector.

```
  1: declare parents: MyList implements List;
  2: declare parents: MyVector extends Vector;
```

2.5 OTHER CONSTRUCTS

Another way of static modification permits the conversion of a checked exception to an unchecked one. AspectJ may soften an exception. That is, it captures the checked exception and raises an unchecked exception in its place. The unchecked exception type is `SoftException`. It has a method (`getWrappedThrowable`) that returns the softened exception. The following example indicates that methods calling the `read()` method in class `InputStream` do not need to handle the exception `IOException`. In this case, it is necessary to handle the new unchecked exception (`SoftException`) somewhere else, otherwise this exception may cause the program to terminate abnormally.

```
declare soft:IOException:call(* InputStream.read())
```

2.6 ASPECTJ SUBSET

In this dissertation, we consider a subset of AspectJ. This simplifies the definition of transformations and does not compromise our results. However, this may limit the number of refactorings we are able to derive with our laws. First, our language does not have packages, and the use of `this` to access class members is mandatory. Also, the `return` statement can appear at most once inside a method body and has to be the last command. Second, we restrict the aspect-oriented constructs, not considering abstract aspects and supporting only the pointcut designators `call`, `execution`, `args`, `this` and `target`.

Restricting the use of `this` simplifies the preconditions defined for the laws. This can be seen as a global precondition instead of a restriction to the language. Most of the laws dealing with advices require this restriction. This restriction allows an easy mapping from the executing object referenced from `this` to the executing object exposed inside advices with the pointcut designator `this`, as we explain in Chapter 3.

The restrictions applied to the aspect-oriented constructs are limitations to our set of laws. In this dissertation, we do not cover transformations involving creation and maintenance of abstract aspects. It would be necessary to have laws for creating abstract aspects, moving pointcuts and advices among aspects, and changing the aspect hierarchy. Also, we only support the mentioned pointcut designators because we think they may represent the core features of this aspect-oriented language. Extending the set of laws to include other AspectJ constructs would be time demanding but not difficult. Besides, it would not affect the already defined laws. For instance, it would be necessary to define new laws to deal with abstract aspects. This is regarded as future work.

This work also assumes hypothesis that must be satisfied in order to correctly use our laws. Those hypothesis include the following:

- Programs are always sequential. The programs we consider may not be concurrent.
- Programs do not use reflection.

Mechanisms such as concurrency and reflection would increase complexity and make very difficult to reason about aspect-oriented programs. Those mechanisms may break several of the laws presented in Chapter 3. Those hypothesis are also considered for object-oriented programming laws [10].

CHAPTER 3

LAWS

> *As far as the laws of mathematics refer to reality, they are not certain;*
> *and as far as they are certain, they do not refer to reality.*
> —ALBERT EINSTEIN

Sometimes, modifications required by refactorings are difficult to understand as they might perform global changes in an application. We use laws of programming [27] to show that an AspectJ transformation is indeed a refactoring. A refactoring denotes a behaviour preserving transformation that increases code quality. In contrast, a law is bi-directional and it does not always increase code quality; it is part of a bigger strategy that does. Besides, our laws are much simpler than most refactorings because they involve only localized changes, and each one focus on one specific AspectJ construct.

In this chapter we describe our laws, showing their intent, structure, and preconditions. Our laws establish the equivalence of AspectJ programs provided some restrictions are respected. Therefore, the structure of each law consists of three parts: left-side, right-side and preconditions. The first two are templates of the equivalent programs. The third part indicates conditions that must hold to ensure the equivalence between the programs.

For example, the following law has the purpose of introducing or removing the `privileged`[1] modifier, which indicates that the aspect can access private members of classes. Most of our laws assume that the aspect has access to private members of classes. It enables us to relax conditions in order to transform the code. However, we can always use this law to remove the privilege in situations where the code does not access private members. We denote the set of type declarations (classes and aspects) by *ts*. Also, *pcs* and *as* denote a set of pointcut declarations and a list of advice declarations, respectively. Note that there is a law for just introducing a new aspect, Law 22 (*Add Empty Aspect*), which is very simple and can be found in Appendix A.

Law 1 - Make Aspect Privileged

```
ts
aspect A {
   pcs
   as
}
```

=

```
ts
paspect A {
   pcs
   as
}
```

[1] We abstract the declaration `'privileged aspect'` as `paspect` for simplicity.

provided

($\leftarrow$) Advice bodies from *as* do not refer to private members declared in *ts*

Our laws basically represent two transformations, one applying the law from left to right and another in the opposite direction. Each law provides preconditions to ensure that the program is valid after the transformation. Another use of the preconditions is to guarantee that the law preserves behaviour. Some laws, when applied from right to left, correspond, roughly, to the transformations applied by the AspectJ compiler to join (weave) the classes and aspects.

There are different preconditions depending on the direction the law is used. This is represented by arrows, where the symbol ($\leftarrow$) indicates this precondition must hold when applying the law from right to left. Similarly, the symbol ($\rightarrow$) indicates that this precondition must hold when applying the law from left to right. Finally, the symbol ($\leftrightarrow$) indicates that the precondition must hold in both directions.

Revisiting Law 1, we see from the preconditions that we can always make an aspect `privileged`, since it only increases the scope of the code inside advices. It is important to note that the captured join points remain the same, as the pointcut expressions are not affected by the `privileged` modifier. Note that private methods can be captured by pointcuts even if the aspect is not privileged.

Eventually, we may realize that our aspect does not need access to private members of classes any more. Thus, we apply this law from right to left, removing the `privileged` modifier. However, it is necessary that the list of advices (*as*) does not refer to private members declared in *ts*.

3.1 ADVICE LAWS

The next law, when applied from left to right, moves part of a method's body into an advice that is triggered before method execution. Using this law, we can move the beginning of a method's body (*body'*) to an advice that runs before method execution.

We use $\sigma(C.m)$ to denote the signature of method m of class C, including its return type and the list of formal parameters. Moreover, we use *context* to denote the list of advice parameters, including the executing object (mapped to a parameter named *cthis*) and the method's parameters (*ps*). We use *bind*(*context*) to denote the expression of pointcut designators that bind the advice parameters (`this`, `target`, and `args`). The laws always expose the maximum context available. For example, Law 2 can expose the executing object and the formal parameters of the captured method. Considering a method `credit` in an `Account` class, the expanded advice signature looks like the code shown next. In this case, *context* is the parameter list (`Account cthis, float amount`) and *bind*(*context*) is the expression `this(cthis) && args(amount)`.

```
before(Account cthis, float amount) :
   execution(void Account.credit(float)) &&
   this(cthis) && args(amount)
```

Law 2 - Add Before-Execution

```
ts
class C {
  fs
  ms
  T m(ps) {
    body';
    body
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  pcs
  bars
  before(context) :
        execution(σ(C.m)) &&
        bind(context) {
    body'[cthis/this]
  }
  afs
}
```

provided

(→) *body*′ does not declare or use local variables; *body*′ does not call `super`;

(←) *body*′ does not call `return`;

(↔) A has the lowest precedence on the join points involving the signature $\sigma(C.m)$; There is no designator `within` or `withincode` capturing join points inside *body*′;

We also denote the set of field declarations and method declarations by *fs* and *ms*, respectively. We consider a simplified law where we omit visibility modifiers, throws clauses and inheritance constructs. However, we have similar laws that include the variations of those constructs in order to match different code templates. As specified in Section 2.6, the `return` statement can only occur once inside a method and it must be the last command. Hence, the method is supposed to have a locally defined variable to hold the return value, which is returned at the end. Both the variable declaration and the return statement are omitted, as they do not contribute to understanding the law.

Note that the advices can not be considered as a set, since order of declaration dictates precedence of advices. According to the AspectJ semantics, if two advices declared in the same aspect are `after`, the one declared later has precedence; in every other case, the advice declared first has precedence. Thus, we divide the list of advices in two. The first part (*bars*) contains the list of all `before` and `around` advices, while the second part contains only `after` advices (*afs*). This separation ensures that `after` advices always appear at the end of the aspect. It also allows us to define exactly the point where the new advice should be placed to execute in the same order in both sides of the law. Additionally, for advices declared in different aspects, precedence depends on their hierarchy or their order in a `declare precedence` construct (see Chapter 2).

Examining the left hand side of Law 2, we see that *body*$'$ executes after all `before` advices declared for this join point. It also executes after all the `around` advices, intercepting this join point, call `proceed`[2]. This means that the new advice on the right hand side of the law should be the last one to execute, preserving the order in which the code is executed in both sides of the law. Thus, the `before` advice should be placed after the list of `before` and `around` advices, but before the list of `after` advices. Moreover, to ensure that the new advice created with Law 2 is the last one to execute, we have a precondition stating that aspect A has the lowest precedence over other aspects defined in *ts*. This precondition must hold in both directions.

As we move *body*$'$ to an aspect, its visible context changes as well. Hence, it is necessary to constrain the context dependencies in order to guarantee that the law relates valid AspectJ programs. Therefore, we impose conditions on accessing local variables and calls to `super` and `return`. Local variables can generally be removed using object-oriented programming laws [10]. The language restriction to obligate the use of `this` to access class members is important to enable the mapping of accesses to the object referenced by `this` to the object exposed as the executing object on the advice (*cthis*). The mapping is denoted by the expression *body*$'$[*cthis*/`this`], where we substitute all occurrences of *this* with the variable *cthis* in *body*$'$.

Nevertheless, there are other implications that must be considered. Changes to the method execution flow (calls to `return`) are generally not allowed because the advice can not implement it, or it would increase complexity. This precondition is necessary to ensure that the law preserves behaviour. It is also necessary to constrain the use of the `within` and `withincode` designators because those designators are based on the syntactic code location. As we change the code location (*body*$'$) those designators may cease to capture join points inside the code being moved. This is a precondition that applies for every law that changes a piece of code location.

This is the simplest law to introduce an advice. Our laws consider the `execution` and `call` designators, as well as five types of advices: `before`, `after`, `after returning`, `after throwing` and `around`. Thus, combining the pointcut designators and advices, we have a total of 10 laws for introducing advices[3]. Each of those laws uses different advice constructions, thereby requiring different method templates.

The next law shows an advice using the pointcut designator `call`. The advices that use the `call` designator are slightly different from the ones using `execution`. The captured join point must appear inside a method's body and before a call to a second method. Moreover, there is a new parameter that can be exposed from the context, the `target` object. Hence, we expose both `this` and `target` objects, and the method's arguments.

We use the `withincode` operator of AspectJ to restrict the calls to the captured

[2] `Around` advices may skip execution of lower precedence advices, as well as the method itself, if it does not call `proceed`.

[3] This number increases considering variations in visibility modifiers, `throws` clauses and inheritance constructs.

method occurring only inside the originating method (n). Additionally, α preceding a list of parameters represents the list of its values. Most of the preconditions to apply this law are similar to the preconditions of Law 2. However, some of the preconditions are different. We define a new precondition that must hold in both directions. This precondition states that the type of *exp* is O, assuring that the specification of the pointcut is correct. In a similar law considering more than one call to m inside n, there would be an extra condition stating that every occurrence of this call must be preceded by *body*. Another variation in this law may consider the existence of code after the call to method m. This law would expose an object of type O, as the `target`, in addition to the context exposed in Law 2.

Law 3 - Add Before-Call

```
ts
class C {
  fs
  ms
  T n(ps') {
    body;
    exp.m(αps)
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T n(ps') {
    exp.m(αps)
  }
}
paspect A {
  pcs
  before(context) :
        withincode(σ(C.n())) &&
        call(σ(O.m()) &&
        bind(context) {
    body[cthis/this]
  }
  bars
  afs
}
```

provided

- (→) *body* does not declare or use local variables; *body* does not call `super`;
- (←) *body* does not call `return`;
- (↔) A has the highest precedence on the join points involving the signature $\sigma(C.m)$; O is the type of *exp*; There is no designator `within` or `withincode` capturing join points inside *body*′;

The precedence with the `call` designator is different from the precedence already discussed for an `execution` designator. In this case, the newly created advice has to be the first to execute. Note that *body*′ on the left hand side of the law executes before any `after` advice affecting the the considered method call. Thus, the advice is placed on top of the list of `before` and `around` advices (*bars*). Also, there is a precondition stating that *A* has the highest precedence over other aspects defined in *ts*.

Next, we start exploring the `after` advices. The first case is when a piece of code executes after another, independently of how the first one finishes execution. We can implement this behaviour in Java using a `try-finally` block. The `try` block executes, then the `finally` block executes, even if the first part raises an exception. This structure maps to the construction of a simple `after` advice in AspectJ shown in Law 4.

Law 4 - Add After-Execution

```
ts
class C {
  fs
  ms
  T m(ps) {
    try {
      body
    } finally {
      body′
    }
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  pcs
  bars
  afs
  after(context) :
        execution(σ(C.m)) &&
        bind(context) {
    body′[cthis/this]
  }
}
```

provided

(→) *body′* does not declare or use local variables; *body′* does not call `super`;

(↔) *A* has the highest precedence on the join points involving the signature $\sigma(C.m)$; There is no designator `within` or `withincode` capturing join points inside *body′*;

The preconditions to applying this transformation are the same as Law 2 preconditions. In fact, all the laws used to introduce advices have a similar behaviour because they all execute a piece of code near a join point. Again, notice that this transformation applied from right to left is almost the same applied by the AspectJ compiler when weaving an `after` advice.

According to the AspectJ semantics, if two advices declared in the same aspect are `after`, the one declared later has precedence. Thus, we include our new advice as the last `after` advice. This ensures that this advice will be the first `after` advice to execute.

Hereafter, we will show only laws that present issues not discussed so far. All the laws not included in this chapter can be found in Appendix A.

It is also possible to build an `after` advice that is triggered only if the captured join point executes successfully (Law 5). In this case, it is also possible to expose the returning value to be used inside the advice body. This is represented by the expression `returning`(T t), where T is the return type and t is the name of a variable holding the return value.

Law 5 - Add After-Execution Returning Successfully

```
ts
class C {
   fs
   ms
   T m(ps) {
      body;
      body′
   }
}
paspect A {
   pcs
   bars
   afs
}
```

=

```
ts
class C {
   fs
   ms
   T m(ps) {
      body
   }
}
paspect A {
   pcs
   bars
   afs
   after(context) returning(T t) :
         execution(σ(C.m)) &&
         bind(context) {
      body′[cthis/this]
   }
}
```

provided

(→) *body′* does not declare or use local variables; *body′* does not call `super`;

(↔) A has the highest precedence on the join points involving the signature $\sigma(C.m)$; There is no designator `within` or `withincode` capturing join points inside *body′*;

Another variation of the `after` advice is triggered only if the captured join point finishes execution throwing an specific exception (Law 6). Similarly to the previous law, it is possible to expose the thrown exception. The expression `throwing`(E e) indicates the type of the exception that triggers the advice (E) and also declares a variable that can be used inside de advice (e).

The last kind of advice covered by our laws is the `around` advice. As it is more powerful and complex we assume that it is only used when the problem can not be solved with a pair of `before/after` advices. Besides, we provide Law 8, which is meant to transform an `around` advice into a pair of `before/after` advices and vice versa (see Section 3.2).

Law 6 - Add After-Execution Throwing Exception

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    try {
      body
    } catch(E e) {
      body′
      throw e
    }
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    body
  }
}
paspect A {
  pcs
  bars
  afs
  after(context) throwing(E e) :
          execution(σ(C.m)) &&
          bind(context) {
    body′[cthis/this]
  }
}
```

provided

(→) *body′* does not declare or use local variables; *body′* does not call `super`;

(←) *body′* does not call `return`;

(↔) A has the highest precedence on the join points involving the signature $\sigma(C.m)$; There is no designator `within` or `withincode` capturing join points inside *body′*;

The law we show next is supposed to be used when the execution of the method's core logic is conditional, which means that the core method's logic may not execute at all. Note that variations of this law may not include the *body′* and *body″*. Another use of `around` advice is discussed with Law 17.

The ordering of advices in this case is similar to a `before` advice. However, the precedence among different aspects has to be considered differently. In this case, it is not possible to ensure the precedence because it is not possible to ensure that *body′* and *body″* execute in the same order. Similarly to Law 2, *body′* should have a low precedence, whereas *body″* should have a high precedence. This can not be accomplished and thus we provide a precondition stating that only aspect A may affect the join point $\sigma(C.m)$. Hence, it is possible to control the precedence only by controlling the ordering of advices inside aspect A.

Law 7 - Add Around-Execution

```
ts
class C {
  fs
  ms
  T m(ps) {
    body′
    if (cond) {
      body
    }
    body″
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  pcs
  bars
  T around (context) :
        execution(σ(C.m)) &&
        bind(context) {
    body′[cthis/this]
    if (cond) {
       proceed(αcontext)
    }
    body″[cthis/this]
  }
  afs
}
```

provided

(→) *body′*, *body″* and *cond* do not declare or use local variables; and do not call `super`;

(←) *body′* does not call `return`;

(↔) There is no aspect in *ts* affecting the join point $\sigma(C.m)$; There is no designator `within` or `withincode` capturing join points inside *body′*;

3.2 ASPECT RESTRUCTURING

Once an advice is in place, we need to simplify its structure, improving legibility. For this purpose we have Laws 8 (*Around to Before-After*), 9 (*Merge advices*), 11 (*Remove* `target` *parameter*), 12 (*Extract named pointcut*), 27 (*Remove* `this` *parameter*), 28 (*Remove argument parameter*), and 32 (*Use named pointcut*), providing a way to covert among advice types, merge equal advices, remove some context exposure not used and, finally, create and use named pointcuts from the advice expressions.

We use the following law to turn an `around` advice into a pair of `before` and `after` advices and vice versa. A precondition is necessary to ensure that the `around` advice is not using the `proceed` construction to change the parameter values of the captured join point. This behaviour can not be represented with the separated `before` and `after` advices. Also note that the call to `proceed` is mandatory.

Law 8 - Around to Before-After

```
ts
paspect A {
   pcs
   bars
   T around(context) :  exp {
      body
      proceed(αcontext)
      body′
   }
   bars′
   afs
}
```

=

```
ts
paspect A {
   pcs
   bars
   before(context) :  exp{
      body
   }
   bars′
   afs
   after(context) returning :  exp{
      body′
   }
}
```

provided

(↔) *body* does not change the values of the *context* variables.

The ordering of advices is slightly more complex in this case. As we organize our aspect declaring all the `after` advices at the end, all `around` advices have precedence over them. Thus, code after a call to `proceed` executes before any `after` advice declared for the same join point. Hence, similarly to Law 4, we declare the `after` advice at the and of the list. Moreover, we must declare the `before` advice in the same place where the `around` advice was declared. This ensures that the order of execution is not changed.

The next law is responsible for merging advices that execute the same action at different join points. Therefore, it enables us to have an advice capturing several join points. We did not focus on simplifying the resulting expressions, although it would be a valuable contribution. Such simplifications would yield new expressions with wild cards for example. Note that this law deals only with `before` advices, but similar versions of this law deal with other kinds of advice. This law has one precondition that must hold from right to left. It ensures that both advice expressions must bind every exposed parameter in *ps*.

Another precondition must hold for both directions, it states that the sets of join points captured by *exp*1 and *exp*2 are disjoint. If there is a join point in common captured by those expressions, the resulting merged advice may not compile. The AspectJ compiler complies about redundant definitions for pointcut designator such as `this` and `target`. Besides, considering the left hand side of the law, the same *body* would be executed twice for the common join point, whereas for the right hand side it would execute just once.

This kind of precondition (not syntactic) is generally difficult to compute, increasing the complexity of tool intended to provide automation for applying the laws.

Law 9 - Merge Before

```
ts
paspect A {
   pcs
   bars
   before(ps) :  exp1 {
      body
   }
   before(ps) :  exp2 {
      body
   }
   bars'
   afs
}
```

=

```
ts
paspect A {
   pcs
   bars
   before(ps) :  exp1  ||  exp2 {
      body
   }
   bars'
   afs
}
```

provided

(←) *exp*1 and *exp*2 bind all parameters in *ps*.

(↔) The set of join points captured by *exp*1 and *exp*2 are disjoint.

Moreover, Law 9 can only be applied if there is no other advice between the ones we are merging. We must impose this restriction to be sure that the advice precedence is preserved. Suppose there is another advice for *exp*1 or *exp*2 between the advices we are merging. If the resulting advice is placed where the first merged advice was declared, precedence for *exp*2 changes, since the merged advice would execute before another declared advice for *exp*2. The same happens for *exp*1, if we place the resulting advice where the second merged advice one was declared.

In order to cope with the advice ordering problem, we provide Law 10, which is able to invert the order of two advices as long as they do not have any join point in common. If the set of join points captured by the involved advices is disjoint, the precedence rule does not apply. Thus, the involved advices can appear in any order. Although we show this law for `before` advices, any pair of advices `before` or `around` may be inverted the same way. However, considering `after` advices implies considering the list of after advices (*afs*) instead of *bars*. We can apply this law several times to make advices adjacent and, only then, apply Law 9.

Law 10 - Change advice order

```
ts
paspect A {
   pcs
   bars
   before(ps) :  exp { ... }
   before(ps') :  exp' { ... }
   bars'
   afs
}
```

=

```
ts
paspect A {
   pcs
   bars
   before(ps') :  exp' { ... }
   before(ps) :  exp { ... }
   bars'
   afs
}
```

provided

(↔) The set of join points captured by *exp* and *exp′* are disjoint.

There are also other laws that help restructuring the advice in order to improve legibility. For instance, the next law is responsible for removing a `target` parameter of an advice provided that the parameter is not used in the advice body.

Law 11 - Remove Target Parameter

```
ts
paspect A {
   pcs
   bars
   before(T  t, ps) :
            target(t) && exp {
      body
   }
   bars'
   afs
}
```

=

```
ts
paspect A {
   pcs
   bars
   before(ps)  :
            target(T) && exp {
      body
   }
   bars'
   afs
}
```

provided

(→) *t* is not referenced from *body*

Although we can remove the `target` parameter from the context exposed by the advice, the binding designator (in this case the `target`) can not always be removed. Removing the target designator from the pointcut expression implies a generalization. This may cause the advice to capture more join points than before the transformation. Hence, the law only changes the `target` expression to use the object type instead of the parameter. This law also has similar versions for each kind of advice. Nevertheless, there are situations where the `target` designator can be removed. For instance, if the pointcut expression describes a `call` join point including the type for the captured call, the `target` designator would be redundant. We could use a variation of Law 11, which removes completely the `target` designator if the `call` designator already constrains the type of the called object.

Laws 27 and 28 are similar to Law 11, but deal with the pointcut designators `this` and `args` respectively. Both can be found in Appendix A.

The last two laws for restructuring the aspect are related to named pointcuts. Law 12 concerns the creation of a named pointcut based on existing expression used by an advice. This is a simple law and its preconditions ensure that the law relates valid programs. Law 32 is also simple: its purpose is to make an advice use a named pointcut instead of its expression. The definition of Law 32 can also be found in Appendix A.

Law 12 - Extract Named Pointcut

```
ts
paspect A {
   pcs
   bars
   before(ps) :  exp(αps)  {...}
   bars′
   afs
}
```

=

```
ts
paspect A {
   pcs
   pointcut p(ps) :  exp(αps)
   bars
   before(ps) :  p(αps)  {...}
   bars′
   afs
}
```

provided

(→) There is no pointcut named p in pcs

(←) There is no reference to p in ts and A

3.3 EXCEPTION HANDLING

Another useful law is Law 13 which, together with Laws 14, 15, 16, and 17, allows the extraction of exception handling code into an aspect. This law is responsible to turn an exception raised by one join point into a soft exception. The other laws deal with `catch` and `throws` clauses to enable the complete extraction of the exception handling.

Law 13 uses the `declare soft` construct to soften the exception. At the same time, it removes the target exception (E) from the throws clause. The other exceptions raised by the method are denoted by exs. The precondition applied when the law is used from left to right is necessary to guarantee that the softened exception would still be handled, thereby preventing a change in behaviour. Otherwise, the softened exception would bypass its original handling point. Likewise, the precondition when applying the law from right to left guarantees that the code compiles and the exception is handled where necessary. `SoftException` is the type of the unchecked exception used by AspectJ; it wraps the softened exception. There is a similar version of this law, which uses the pointcut designator call.

As we soften an exception as showed in Law 13, the result is that a new unchecked exception is raised instead of the existing one. Hence, every handler of the existing exception ceases to work and the exception would bypass its intended handling point and thus it would not preserve behaviour. It is necessary to copy the existing exception handling code so that the new unchecked exception is handled the same way. In fact, this is a precondition to apply Law 13. To introduce the `catch` for the unchecked exception we have Law 14. We abstract the method *getWrappedThrowable*() from class *SoftException* as *getWT*() for simplicity.

Law 13 - Soften Exception

```
ts
class C {
  fs
  ms
  T m(ps) throws E, exs {
    body
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) throws exs {
    body
  }
}
paspect A {
  declare soft : E :
        execution(σ(C.m));
  pcs
  bars
  afs
}
```

provided

(→) Every catch clause for E in *ts*, *bars* and *afs* have a catch clause for the *SoftException* and a case to handle E when it is the wrapped throwable;

(←) Every catch clause for *SoftException* containing a case to handle E when it is the wrapped throwable is accompanied by a catch for E itself; every call to method m of class C either catches or throws E.

We are adding a new catch clause to handle E even if it is a soft exception, which means that anytime *body* may cease to throw E. When this happens, the compiler would stop saying that E is never thrown from *body*. In order to avoid this, we insert the statement `if (false) throw new` E(). This prevents the compiler error and does not change the behaviour, since this statement would never execute. This law is generally an intermediate step in a bigger transformation. Afterwards we show how this statement can be removed.

It is important to note that this law can only be applied if E is not yet declared as soft. Applying this law, in both directions, if E is already declared soft would imply in a change in behaviour. The code would start or stop handling the *SoftException*, at a point where it was not supposed to. For instance, if E is already declared as soft in the left-right situation, it means that the *SoftException* is handled outside method m. Moving to the next problem, it is necessary to remove the softened exception from the throws clauses of methods that do not raise it any more. We provide Law 15 for that.

Law 14 - Add Catch for Softened Exception

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    try {
      body
    } catch(E e) {
      body′
    } cts
  }
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    try {
      body
      if (false) throw new E()
    } catch(E e) {
      body′
    } catch(SoftException se) {
      if(se.getWT() instof E) {
        E e = (E)se.getWT();
        body′
      } else {
        throw se
      }
    } cts
  }
}
```

provided

(↔) E is not declared as soft in any *ts* join point

Law 15 - Remove Exception from Throws Clause

```
ts
class C {
  fs
  ms
  T m(ps) throws E, es {
    body
  }
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    body
  }
}
```

provided

(→) *body* does not throw E

(←) Every reference to method m of class C either catches or throws E, including a super method, if that is the case

It seems to be an object-oriented refactoring, but we provide Law 15 because we need its preconditions to derive aspect-oriented refactorings.

Next, it is necessary to remove the `catch` blocks for the softened exception where the `try` body does not raise it anymore. Law 16 addresses this issue. Note that the statement `if (false) throw new` $E()$ discussed for Law 14 can be removed using law 16.

Law 16 - Remove Exception Handling

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    try {
      body
      if (false) throw new E()
    } catch(E e) {
      body'
    } catch(SoftException se) {
      if(se.getWT() instof E) {
        E e = (E)se.getWT();
        body'
      } else {
        throw se
      }
    } cts
  }
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    try {
      body
    } catch(SoftException se) {
      if(se.getWT() instof E) {
        E e = (E)se.getWT();
        body'
      } else {
        throw se
      }
    } cts
  }
}
```

provided

(→) *body* does not throw E

Finally, we have Law 17, which moves the handling of the unchecked exception to an aspect. Laws 13, 14, 15, 16, and 17 are generally related by their preconditions, which imposes a certain order on their application. Law 17 also shows another use for the `around` advice. It handles the exception thrown by the `proceed` call, which executes the original method. We denote a list of catch clauses as *cts*. Also, this law has variations where the catch clause for the *SoftException* is not the first one.

The composition of the laws as described is a complete refactoring to *Extract Exception Handling* code (see Section 4.1.4).

Law 17 - Move Exception Handling to Aspect

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    try {
      body
    } catch(SoftException se) {
      body′
    } cts
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    try {
      body
    } cts
  }
}
paspect A {
  pcs
  bars
  T around (context) throws es :
        execution(σ(C.m)) &&
        bind(context) {
    try {
      proceed(αcontext)
    } catch(SoftException se){
      body′
    }
  }
  afs
}
```

provided

(→) *body′* does not declare or use local variables; *body′* does not call `super`;

(←) *body′* does not call `return`;

(↔) There is no designator `within` or `withincode` capturing join points inside *body′*;

3.4 INTER-TYPE DECLARATIONS

The next law, together with Laws 19, 20, 21, 29, 30, and 33, deals with inter-type declarations. This law is responsible for moving one field declaration to an aspect. This is necessary in cases where a class field is part of a crosscutting concern and has to be considered inside the aspect. We have to assure that all of its references had already been moved to the aspect before moving the field. This restriction is necessary for non-public fields because the semantics is not the same as simply declaring the field in the class. Visibility modifiers in inter-type declarations are relative to the aspect.

The precondition of only the aspect referencing the moved field is rather strong. Depending on the field visibility, there are other elements which can refer to the field.

However, our experience shows that despite strong, this precondition covers all of the analyzed cases included on next chapter.

Law 18 - Move Field to Aspect

```
ts
class C {
  fs;  T field
  ms
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
}
paspect A {
  T C.field
  pcs
  bars
  afs
}
```

provided

(→) The field *field* of class C does not appear in *ts* and *ms*.

The following law has the purpose of moving the implementation of a single method into an aspect using an inter-type declaration. According to the AspectJ semantics, visibility modifiers of inter-type declarations are related to the aspect and not to the affected class. Hence, it is possible to declare a private field as a class member and as an inter-type declaration at the same time and using the same name. As a consequence, transforming a member method that uses this field into an inter-type declaration implies that the method now uses the aspect inter-typed field. This leads to a change in behaviour. A precondition is necessary to avoid this problem.

Law 19 - Move Method to Aspect

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    body
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
}
paspect A {
  T C.m(ps) throws es {
    body
  }
  pcs
  bars
  afs
}
```

provided

(↔) A does not introduce any field to C with the same name of a C field used in *body*

Laws 29 and 30 deal with changing the class hierarchy using the `declare parents` construct. Both laws are simple and all their preconditions are implicit on the code templates. Therefore, those laws can always be applied if the templates are matched; there are no explicit preconditions. Their definitions can be found in Appendix A. The last three laws (20, 21, and 33) are related to moving method implementations and fields to an interface, providing default implementation for some methods. Law 20 is capable of moving one inter-typed field from a class to one of its implemented interfaces. The preconditions are necessary to avoid name conflicts.

Law 20 - Move Field Up to Interface

```
ts                                  ts
interface D {...}                   interface D {...}
class C impl D {...}                class C impl D {...}
paspect A {                         paspect A {
   pcs                          =      pcs
   T C.field                           T D.field
   bars                                bars
   afs                                 afs
}                                   }
```

provided

(→) A does not already introduce an attribute named *field* to interface D

(←) A does not introduce any method to interface D that references *field*

Law 33 is defined similarly, but moves a method implementation from a class to one of its implemented interfaces. Its definition can be found in Appendix A. The last law (Law 21) is intended to remove a method implementation from a class, given that this implementation is already introduced to the interface the class implements. There are no preconditions for this law. Note that all types involved are well constrained by the law definition itself.

Law 21 - Remove Method Implementation

```
ts
interface D {
  ms
  T m(ps)
}
class C impl D {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  pcs
  T D.m(ps) {
    body
  }
  bars
  afs
}
```

=

```
ts
interface D {
  ms
  T m(ps)
}
class C impl D {
  fs
  ms
}
paspect A {
  pcs
  T D.m(ps) {
    body
  }
  bars
  afs
}
```

Table 3.1 summarizes all the laws defined in this chapter along with the laws defined in Appendix A.

Table 3.1. Summary of laws

Law	Name	Law	Name
22	Add empty aspect	28	Remove argument parameter
1	Make aspect privileged	13	Soften exception
2	Add before-execution	14	Add catch softened exception
4	Add after-execution	15	Remove exception from throws clause
5	Add after-execution returning successfully	16	Remove exception handling
6	Add after-execution throwing exceptions	17	Move exception handling to aspect
7	Add around-execution	18	Move field to aspect
3	Add before-call	19	Move method to aspect
23	Add after-call	29	Move implements declaration to aspect
24	Add after-call returning successfully	30	Move extends declaration to aspect
25	Add after-call throwing exceptions	31	Extend from super type
26	Add around-call	12	Extract named pointcut
8	Around to before-after	32	Use named pointcut
9	Merge advices	20	Move field introduction up to interface
10	Change advice order	33	Move method introduction up to interface
11	Remove `target` parameter	21	Remove method implementation
27	Remove `this` parameter		

3.5 SOUNDNESS

This section shows that some of our laws are sound and thus the transformation related to those laws preserve behaviour. We use a semantics for an aspect-oriented language [33] in which we can represent some of the laws. This language is not as expressive as AspectJ, but provides mechanisms to define some kinds of AspectJ advices with a well defined semantics. It allows us to explore notions of semantic equivalence between aspect-oriented programs. This increases the confidence that the transformations applied by the laws preserve behaviour. We discuss the manual proof of one law. Since it is an error-prone activity, we regarded as a future work to encode this semantics [33] and our laws in a formal specification language, such as PVS [43], which has a theorem prover. Here we provide a formal argumentation about soundness of the laws.

As discussed before, some hypothesis must be satisfied in order to correctly use our laws. For instance, the programs can not use reflection and can not be concurrent. Those hypothesis are also considered for object-oriented programming laws [10].

A limitation to our current work is a consequence of being able to represent only part of the laws with the chosen semantics. As the chosen language is not as powerful as AspectJ, we can represent Laws 2, 3, 5, 9, 11, and 27. It would be necessary to define another language (or extend the one we used) to prove the remainder of the laws. Nevertheless, we can use this subset of the laws to show that some important refactorings indeed preserve behaviour, for instance, the *Extract Method Calls* [31].

3.5.1 Semantics of Method Call Interception (MCI)

Semantics for aspect-oriented languages is still an emerging field. The aspect-oriented languages used today still do not have an associated formal semantics where it is possible to formally reason about programs. However, there are several approaches [8, 1, 49, 33, 34, 36, 48, 17, 7] that try to solve this problem. In this section we discuss an aspect-oriented semantics based on Method Call Interception (MCI) [33].

The MCI semantics was chosen because it allows us to represent several of the advice types offered by AspectJ, allowing us to reason about programming laws involving those kinds of advice. Moreover, the MCI semantics is described as an extension to an object-oriented one, similarly to the way AspectJ extends Java. Therefore, the MCI semantics provides an easier comprehension of how the semantics change from the object-oriented language to its aspect-oriented extension. This semantics only deals with advices, which we consider as a core concept in aspect-orientation. However, other AspetcJ constructs, such as inter-type declarations, are also important and the proof for laws involving them should consider a different or extended language.

Lämmel starts defining the semantics for a small java-like object-oriented language called μO^2 [33]. He describes an operational semantics and defines the rules for this language. Although Lämmel describes both static and dynamic semantics, we consider only the dynamic semantics because we want to compare behaviour of programs. The static semantics is useful to verify if the programs are well typed. Hence, the static semantics would be necessary to prove that the laws relate valid programs, this is regarded as a future work.

After defining the semantics for μO^2, Lämmel extends this language to incorporate the new construct `superimpose`, which allows the definition of an advice intercepting a method. However, the first definition for the `superimpose` construct is very simple and needs to be extended. He extends this definition in two ways. First he introduces interactivity, allowing advices to expose and use variables from the method's execution context. Second, he extends the language definition including quantitative mechanisms, allowing advices to intercept several methods. The syntax for the resulting aspect-oriented language can be seen in Figure 3.1. The MCI extension starts at the `caller` definition.

prog	=	*cdef** *cn.mn*
cdef	=	`class` *cn* `extends` *cn* {*field** *mdef**}
field	=	*type fn*
mdef	=	*type mn* (*arg**) *body*
type	=	*cn* | `void`
arg	=	*type vn*
body	=	*exp* | `abstract`
cn	=	class names
fn	=	field names
mn	=	method names
vn	=	variable names
exp	=	`null`
	|	`this`
	|	*vn*
	|	`view` *type exp*
	|	*exp.fn*
	|	*exp.vn* = *exp*
	|	*exp.mn* (*exp**)
	|	`super`.*mn* (*exp**)
	|	`let` *vn* : *type* = *exp* `in` *exp*
	|	*exp;exp*
	|	`while` (*exp*) *exp*
	|	`caller`
	|	`callee`
	|	`superimpose` *exp* `on` *eve*
eve	=	*mci loc* | *eve* `within` *loc*
mci	=	`dispatch` | `enter` | `exit`
loc	=	*
	|	`object` *exp*
	|	`class` *cn*
	|	`subclass` *cn*
	|	`method` *mn*
	|	`result` *type*
	|	`argument` *type vn*
	|	*loc* && *loc*
	|	*loc* || *loc*
	|	!*loc*

Figure 3.1. MCI syntax

The `superimpose` construct defines that some code (*exp*) is to be executed on the occurrence of an event (*eve*). Comparing to AspectJ, the *exp* can be regarded as the advice body, and *eve* can be regarded as the pointcut expression. The description of an event defines when and where a method interception occurs. A method can be intercepted at three distinct points (*mci*): `dispatch`, before its arguments evaluation; `enter`, after the arguments evaluation but before the method's execution; and `exit`, after the method's execution. Those *mci* points are analogous to the `before-call`, `before-execution` and `after-returning-execution` from AspectJ. The other component of an event (*loc*) describes the location of the method interception, which is an expression that matches methods based on its name, class, arguments, return type, etc. An event can also be constrained to occur only within another location.

Lämmel defines an operational semantics for this language [33]. He defines several rules to show how an expression should be evaluated. Each rule shows the return value of the evaluated expression and shows how the state changes. Some rules may depend on the execution of other rules to achieve its result. Hence, the evaluation of a program can be represented as a tree showing several evaluation rules.

The domains for a rule consist of a method code table (T), which links method names with its parameters and body. An object store (Σ) that holds references to objects and its field values. This object store also hold the advice registry and will be explained later in this Chapter. There is also a reference to the executing object (θ) and an environment for the program variables (η). The expression $\Pi_i(t)$ denotes the *ith* projection of a tuple t.

Figure 3.2 shows the evaluation rule [33] for the `superimpose` construct. This rule states that evaluating a `superimpose` declaration returns a `null` reference (0 is the meaning of a null expression) and updates the object store (Σ''). The `superimpose` evaluation consists of three steps: first, we evaluate the event expression (3.1), which yields the event description ($\overline{k}$) and an updated object store (Σ'); second, we create the advice, represented by α (3.2); finally, we call the `register` helper function (3.3), which updates Σ', yielding Σ'', by registering the event and advice from the previous evaluations.

$$
\begin{array}{ll}
T, \Sigma, \theta, \eta \vdash \mathit{eve} \Rightarrow \overline{k}, \Sigma' & (3.1)\\
\wedge\ \alpha = ((\Pi_{cn}(\theta), \Pi_{mn}(\theta)), \mathit{exp}) & (3.2)\\
\wedge\ \overline{\texttt{register}}(\Sigma', \overline{k}, \alpha) \Rightarrow \Sigma'' & (3.3)\\
\hline
T, \Sigma, \theta, \eta \vdash \texttt{superimpose}\ \mathit{exp}\ \texttt{on}\ \mathit{eve}\ \Rightarrow 0, \Sigma'' & (3.4)
\end{array}
$$

Figure 3.2. `superimpose` evaluation rule

We do not show all the evaluation rules, more details can be found elsewhere [33]. As mentioned before, one of the reasons to choose the MCI semantics is that it shows an object-oriented semantics and extends it to introduce MCI. This description allows us to see exactly how the semantics change when we introduce aspect-oriented features to the language. As the superimpose construct affects only method calls, the only rule changed during the MCI extension is the method `call` evaluation rule.

Originally a method call is evaluated according to the rule listed in Figure 3.3. First, we evaluate the expression that yields the object on which the method is being called (3.5). Second, we search the environment for the method definition (3.6). Then, it is necessary to evaluate the expressions representing the arguments values (3.7-3.9). Finally, an environment is mounted with the evaluated arguments (3.10) to execute the method's body (3.11).

$$T, \Sigma_0, \theta, \eta \vdash \mathit{exp} \Rightarrow \rho, \Sigma_1 \tag{3.5}$$
$$\wedge\ \Pi_1(T) \bullet (\rho, \mathit{mn}) = ((\mathit{vn}_1, ..., \mathit{vn}_n), \mathit{exp}') \tag{3.6}$$
$$\wedge\ T, \Sigma_1, \theta, \eta \vdash \mathit{exp}_1 \Rightarrow v_1, \Sigma_2 \tag{3.7}$$
$$\wedge\ ... \tag{3.8}$$
$$\wedge\ T, \Sigma_n, \theta, \eta \vdash \mathit{exp}_n \Rightarrow v_n, \Sigma_{n+1} \tag{3.9}$$
$$\wedge\ \eta' =\perp [\mathit{vn}_1 \mapsto v_1, ..., \mathit{vn}_n \mapsto v_n] \tag{3.10}$$
$$\wedge\ T, \Sigma_{n+1}, \eta' \vdash \mathit{exp}' \Rightarrow v, \Sigma_{n+2} \tag{3.11}$$
$$\overline{\quad T, \Sigma_0, \theta, \eta \vdash \mathit{exp}.\mathit{mn}(\mathit{exp}_1, ..., \mathit{exp}_n) \Rightarrow v, \Sigma_{n+2} \quad} \tag{3.12}$$

Figure 3.3. Object-oriented `call` evaluation rule

A general object reference is represented by ρ. Function application is denoted as $f \bullet x$, and the entirely undefined function is denoted as $\perp$. The evaluation of a method call yields its value (v') and an updated object store (Σ'_{n+2}).

With the MCI extension, the `call` rule is changed to verify at certain points, if there is a registered event that should be executed. Figure 3.4 shows the `call` rule with the MCI extension. The lookup for registered events matching this method's execution is done through the helper functions `dispatch` (3.15), `enter` (3.20), and `exit` (3.22).

$$T, \Sigma_0, \theta, \eta \vdash \mathit{exp} \Rightarrow \rho, \Sigma_1 \tag{3.13}$$
$$\wedge\ \Pi_1(T) \bullet (\rho, \mathit{mn}) = ((\mathit{vn}_1, ..., \mathit{vn}_n), \mathit{exp}') \tag{3.14}$$
$$\wedge\ \texttt{dispatch}(T, \Sigma_1, \theta, (\rho, \mathit{mn})) \Rightarrow \Sigma'_1 \tag{3.15}$$
$$\wedge\ T, \Sigma'_1, \theta, \eta \vdash \mathit{exp}_1 \Rightarrow v_1, \Sigma_2 \tag{3.16}$$
$$\wedge\ ... \tag{3.17}$$
$$\wedge\ T, \Sigma_n, \theta, \eta \vdash \mathit{exp}_n \Rightarrow v_n, \Sigma_{n+1} \tag{3.18}$$
$$\wedge\ \eta' =\perp [\mathit{vn}_1 \mapsto v_1, ..., \mathit{vn}_n \mapsto v_n] \tag{3.19}$$
$$\wedge\ \texttt{enter}(T, \Sigma_{n+1}, \theta, (\rho, \mathit{mn}), \eta') \Rightarrow \Sigma'_{n+1} \tag{3.20}$$
$$\wedge\ T, \Sigma'_{n+1}, ((\rho, \mathit{mn}), \perp) \vdash \mathit{exp}' \Rightarrow v, \Sigma_{n+2} \tag{3.21}$$
$$\wedge\ \texttt{exit}(T, \Sigma_{n+2}, \theta, (\rho, \mathit{mn}), \eta', v) \Rightarrow v', \Sigma'_{n+2} \tag{3.22}$$
$$\overline{\quad T, \Sigma_0, \theta, \eta \vdash \mathit{exp}.\mathit{mn}(\mathit{exp}_1, ..., \mathit{exp}_n) \Rightarrow v', \Sigma'_{n+2} \quad} \tag{3.23}$$

Figure 3.4. MCI `call` evaluation rule

An event can be registered using the `superimpose` construct. The lookup functions showed in the previous rule, search the environment to see if the registered event matches the executing method. If there is a match, the registered expression is executed. Note that the `superimpose` must be evaluated before the method call for the advice to take effect. Any method calls made before the `superimpose` evaluation will behave according to the μO^2 rule because the environment will not have a registered event. This feature allows us to dynamically introduce advices, which is not possible in AspectJ.

As we want to map the MCI semantics to AspectJ, we need to constrain the language to ensure that all `superimpose` expressions are evaluated before the program starts executing. This can be achieved by allowing `superimpose` declarations only at the beginning of the main method (method called to initiate the program execution according to the language grammar, see *prog* in Figure 3.1).

It is possible to represent part of the advice types provided by AspectJ using the `superimpose` construct. In fact, we can represent `before-call`, `before-execution` and `after-returning-execution` advices. The first type maps to a `superimpose on dispatch` construct, the other two can be mapped to `superimpose on enter` and `superimpose on exit` constructions, respectively.

Other AspectJ constructs, including pointcuts, inter-type declarations, and other kinds of advice, can not be represented with the MCI semantics. This limitation enables us to reason only about Laws 2, 3, 5, 9, 11, and 27. In Section 3.5.3 we discuss the soundness of Law 2 (*Add Before-Execution*). To enable the proof of the other laws, it would be necessary to extend the presented language, or to define a completely new one. This is regarded as a future work.

3.5.2 MCI Program Equivalence

We want to use the proposed semantics to reason about aspect-oriented programs and verify whether two programs behave the same. Thus, it is necessary to define an equivalence relation between them. This equivalence relation can be difficult to define. For instance, if we choose an equivalence relation that compares two environments (states) resulting from programs execution, it would fail to compare programs that behave the same but use different data structures. Different data structures may result in different environments at the end of a program execution. For example, consider two stack implementations: the first uses an array to represent the stack, and the second uses a linked list. Both implementations may behave as a stack, but their final states are different because their data structures are different. In this case, it would be necessary to isolate input and output variables from the environment and compare only those variables.

As the programming laws we are willing to proof, with the MCI semantics, do not change the data structure, we can establish equivalence by comparing the object stores generated by the evaluation of both programs. Figure 3.5 shows the object store (Σ) domain to evaluate an expression [33]. This domain has three components: a function that associates data locations with their values ($\delta \rightarrow_{fin} v$), a function that associates object references with their types ($\rho \rightarrow_{fin} cn$), and the advice registry ($\overline{\Delta}$). Our equivalence notion only uses the first component of the object store comparing the field values and how they change, as stated by Definition 1. The runtime type information is not relevant to our relation, it is part of the object store to allow the evaluation of expressions like type casts. The advice registry is expected to change because we intend to introduce new `superimpose` commands to the program. Hence, it can not be compared.

$$\boxed{\begin{array}{lcll}
\Sigma & = & \delta \rightarrow_{\mathit{fin}} v & \text{(Object store)} \\
 & \times & \rho \rightarrow_{\mathit{fin}} cn & \text{(Runtime type information)} \\
 & \times & \overline{\Delta} & \text{(Advice registry)} \\
\delta & = & \rho \times \mathit{fn} & \text{(Data locations)} \\
\rho & & & \text{(Object references)}
\end{array}}$$

Figure 3.5. Object Store

Definition 1 (Program Equivalence) *Let P and Q be two MCI programs. P is equivalent to Q ($P \equiv Q$) iff, for all valid input, the fields and their values from the resulting object store of P equals that of Q.*

We are only interested in the first component from the object store, which maps field locations to their values. Thus, after the programs evaluation we can compare the values of their fields and state that two programs behave the same if all their fields and values are equal.

Although we define the equivalence relation for MCI, this notion is independent of programming languages. However, this equivalence relation can only be considered for sequential programs. If the programs are concurrent, the equivalence relation should consider the structure of the evaluation tree as well. Nevertheless, our laws do not deal with those mechanisms.

3.5.3 Soundness of the Add Before-Execution Law

In this section we show that the Law 2 (*Add Before-Execution*) is sound using the semantics we chose. We interpret both sides of the law according to the semantics. Then we compare the resulting environments according to our equivalence notion to see whether the two sides of the law have the same meaning.

Following, we show the Law 2 written in terms of the MCI syntax. Thus, we map the `before-execution` advice from AspectJ to a `superimpose on enter` construct from the MCI language (see Section 3.5.1). Also, we constrain the language allowing only declarations of the `superimpose` construct at the beginning of the main method. Moreover, the MCI language does not have any modular concept similar to an aspect. Thus, the aspect simulation is also accomplished by the use of a main method with `superimpose` declarations at the beginning. As a consequence, changes made to the aspect are represented as changes made to the main method and its superimposes. Note that, similarly to the AspectJ law, we have to substitute the `this` keyword for the `callee` keyword when using *body*′ on the right hand side of the law.

There is also the advice ordering problem discussed in Section 3. According to our understanding from the MCI semantics, advices declared later have precedence, no matter the kind of MCI. Thus, we do not need to separate advices as we do with AspectJ. It is only necessary to declare the new `superimpose on enter`, just before all the other `superimpose` declarations (*sis*) to ensure that the new one is the last to be executed.

If we were dealing with Law 5 (*Add after-execution returning successfully*), the new `superimpose` declaration should be placed after all the existing ones to ensure that the `after` advice should be the first to execute. We assume that the kind of rewriting discussed so far, does not change the semantics of Law 2.

In Section 3.5.1 we showed that there is just one evaluation rule that changes with the MCI extension. Thus, our soundness discussion involves only the `call` rule. A complete proof would involve all the language constructs and use induction on the structure of *mainBody*. The base case would consider each single command that can appear in *mainBody*, while the induction step would consider every composition of those commands. This complete proof is regarded as a future work, here we provide a formal argumentation to show that Law 2 is sound.

Law 2 - Add Before-Execution (MCI)

```
ts
class C ext T {
 fs
 ms
 Type m(ps) {
   body';
   body
 }
}
class M ext T {
 void main() {
   sis;
   mainBody
 }
}
```

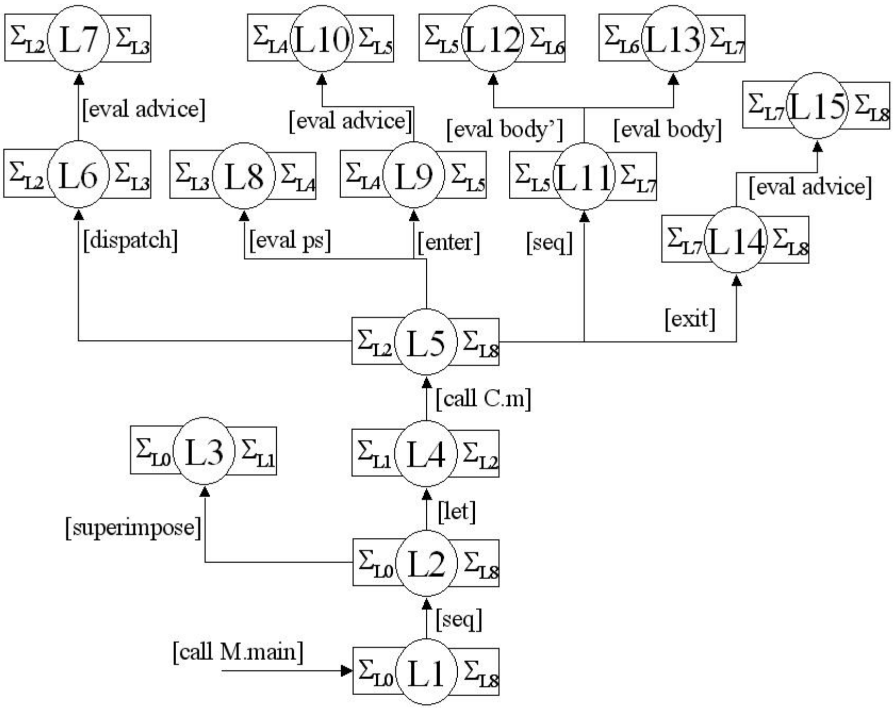
=

```
ts
class C ext T {
 fs
 ms
 Type m(ps) {
   body
 }
}
class M ext T {
 void main() {
   superimpose body'
     on enter
      class C &&
      method m &&
      argument ps;
   sis;
   mainBody
 }
}
```

Note that we did not need the preconditions of Law 2. This is a direct consequence of the differences between the semantics of AspectJ and the MCI semantics. The preconditions relating the use of `super` and `return` do not apply for the MCI version of the law. As the advice from the `superimpose` construct run in the context of the intercepted method (not as another method call), both constructs may appear inside the `superimpose` declaration. The precondition relating precedence of advices is not necessary because the MCI semantics has no entity to represent an aspect.

Proof. (Sketch) Our argumentation is based on a case where the *mainBody* represents a single call to method *m* of class *C* (note that we need to create an object, using the `let` construct, to call a method). This comes directly from the fact that the `superimpose` only affects the method call semantics. Any other simple construction for *mainBody* would trivially preserve behaviour because the other language constructs are not affected by the `superimpose`.

Figure 3.6 shows the evaluation tree for the left hand side of the law, considering that *mainBody* is the command: `let` $c : C =$ `new` C `in` $c.m(ps)$. Every node consists of a program state. The transitions represent applications of transition rules according to the semantics. Thus, each transition is labeled after the applied rule. Also, the left square represent the input object store and the right square represents the output object store for each rule applied. The nodes are numbered according to the execution order, with label L1 being the first.

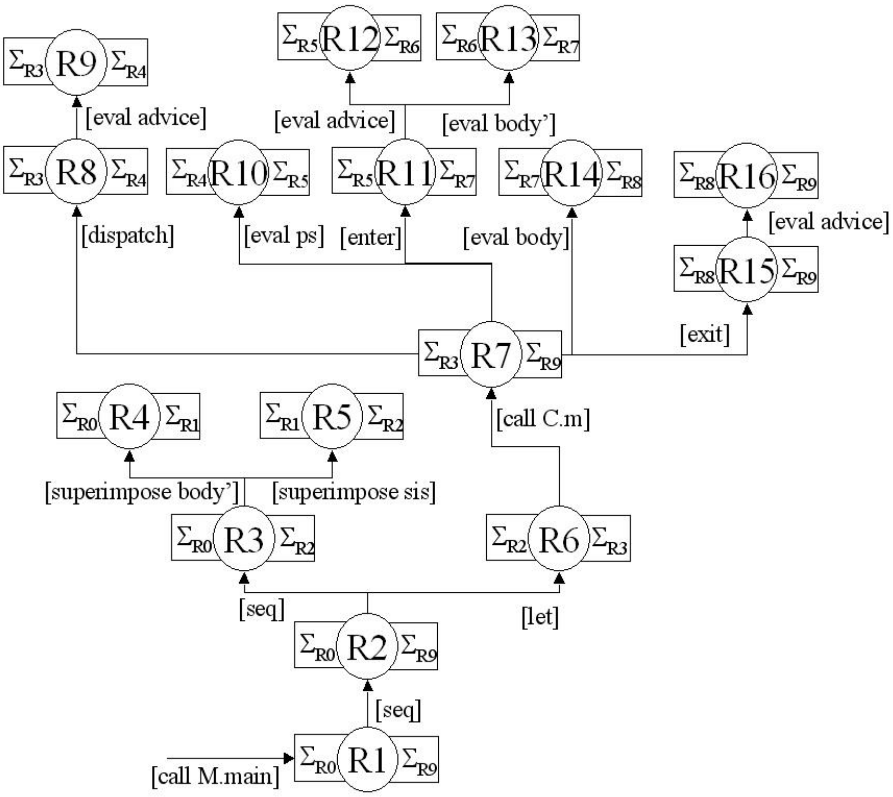


Σ_{L2} L7 Σ_{L3}
Σ_{L4} L10 Σ_{L5}
Σ_{L5} L12 Σ_{L6}
Σ_{L6} L13 Σ_{L7}
[eval advice]
[eval advice]
[eval body']
[eval body]
Σ_{L7} L15 Σ_{L8}
Σ_{L2} L6 Σ_{L3}
Σ_{L3} L8 Σ_{L4}
Σ_{L4} L9 Σ_{L5}
Σ_{L5} L11 Σ_{L7}
[eval advice]
Σ_{L7} L14 Σ_{L8}
[dispatch]
[eval ps]
[enter]
[seq]
[exit]
Σ_{L2} L5 Σ_{L8}
[call C.m]
Σ_{L0} L3 Σ_{L1}
Σ_{L1} L4 Σ_{L2}
[let]
[superimpose]
Σ_{L0} L2 Σ_{L8}
[seq]
[call M.main]
Σ_{L0} L1 Σ_{L8}

Figure 3.6. Evaluation tree for the left hand side.

The left hand side consists in evaluating a sequential composition (L2), which leads to the evaluation of the `superimpose` declarations present in *sis* (L3) and the evaluation of the `let` command (L4). The let updates the store and calls method m of class C (L5). The method call evaluation occurs as showed in Figure 3.4. First, events registered for `dispatch` MCI are executed (L7). Next we evaluate the method's parameters (L8). Then, events registered for `enter` MCI are executed (L10). Following we evaluate the method's body, which is a sequential composition (L11) of body' (L12) and body (L13). Finally, events registered for `exit` MCI are executed (L15). As we want to compare the execution of two programs, we do not expand execution nodes that are equal for both. For instance, the evaluation of *body*, $body'$, *ps*, `dispatch` and `exit` advice nodes are the same for both programs.

Next, Figure 3.7 shows the evaluation tree for the right hand side of the law. In this case, there is a sequential composition(R2) that first evaluates another sequential composition (R3), which includes our new `superimpose` (R4) and the old ones (R5). Then it starts the program similarly to the left hand side. The evaluation of the `superimpose` command updates the registry located on the object store by registering *body′* to be executed when entering the method m with arguments *ps* of class C. As a result, the evaluation of the `enter` helper function (R11) performs a lookup in the registry for events registered for this method and finds that *body′* should be executed (R12). Another difference is that the evaluation of the method's body now includes only *body* (R14).



Σ_{R3} R9 Σ_{R4}
[eval advice]
Σ_{R5} R12 Σ_{R6}
Σ_{R6} R13 Σ_{R7}
[eval advice]
[eval body']
Σ_{R3} R8 Σ_{R4}
Σ_{R4} R10 Σ_{R5}
Σ_{R5} R11 Σ_{R7}
Σ_{R7} R14 Σ_{R8}
Σ_{R8} R16 Σ_{R9}
[eval advice]
[dispatch]
[eval ps]
[enter]
[eval body]
Σ_{R8} R15 Σ_{R9}
[exit]
Σ_{R3} R7 Σ_{R9}
Σ_{R0} R4 Σ_{R1}
Σ_{R1} R5 Σ_{R2}
[call C.m]
[superimpose body']
[superimpose sis]
Σ_{R0} R3 Σ_{R2}
Σ_{R2} R6 Σ_{R3}
[seq]
[let]
Σ_{R0} R2 Σ_{R9}
[seq]
[call M.main]
Σ_{R0} R1 Σ_{R9}

Figure 3.7. Evaluation tree for the right hand side.

According to the equivalence notion established in Section 3.5.2, we are interested on the nodes that may update the first component of the object store (field values). First, the `let` command may update the object store by adding a new object and the values of its fields. The second way to update the field values in the object store is through an assignment. Assignments can appear in any expression and thus, we look for the nodes able to evaluate expressions.

On the left hand side, the nodes related to the evaluation of expressions are: `let` (L4), `dispatch` (L7), *ps* (L8), `enter` (L10), *body′* (L12), *body* (L13), and `exit` (L15). Similarly, the nodes we are interested on the right hand side are: `let` (R6), `dispatch` (R9), *ps* (R10), `enter` (R12), *body′* (R13), *body* (R14), and `exit` (R16).

Analyzing the equivalent nodes from both programs (i.e. L4 and R6, L7 and R9, etc) we can see they are syntactically equal, and thus have an equivalent evaluation. The only factor that may result in different field values at the end of the program execution is the order in which the nodes are evaluated. In both Figures 3.6 and 3.7, the number inside the node represent the order of evaluation, which is the same in both programs. During the evaluation, the field values are supposed to be equal after the evaluation of nodes L13 and R14, because the remaining subtrees are equal for both programs after evaluating those nodes. Thus, according to our equivalence notion, and considering that the programs are sequential, we can conclude that the programs have the same behaviour. ■

3.5.4 Soundness of Other Laws

This proof could be similarly extended for Laws 3 (*Add before-call*), and 5 (*Add after returning successfully*), as they only differ by the kind of advice (MCI) used. Law 3 would use the `superimpose on dispatch` construct and Law 5 would use the `superimpose on exit` construct. For this reason, we consider that these two laws are also sound.

The right hand side of Law 3 would generate an evaluation tree where *body*′ is evaluated before some other method call. This means that *body*′ is evaluated even before the arguments of the method to be called. The evaluation tree for the left hand side would place *body*′ above the `dispatch` node, ensuring that it is also evaluated before the arguments of the considered method.

The proof for Law 5 is almost equal to the proof for Law 2. The only difference is that on the evaluation tree for the left hand side, *body*′ appears after *body*, and on the right hand side, *body*′ appears above the `exit` node. This also ensures that *body*′ is evaluated after *body* in both sides of the law.

However, Laws 9, 11, and 27 should be considered differently. The proof for Law 9 would rely on the composition of MCI locations (|| operand on event locations) to ensure that a registered event matches two or more join points. As the only difference between the left hand side and right hand side is the `superimpose` declarations (consequently the registry), both evaluation trees would be equal. According to the MCI semantics, the evaluation of the || operator is the same as evaluating its first operand an then its second operand. Both evaluations register the same piece of code to execute at different events.

The proof for Laws 11 and 27 would rely on removing the `callee` and `caller` constructs respectively. In the MCI semantics, these constructs only bind variables to be used by the advice, they do not constrain the types as occurs with `this` and `target` in AspectJ. As type restrictions are apart from variable binding, we can remove the variable binding given that the variable is not used inside the advice.

We do not discuss the remaining laws formally. However, in this chapter we described the laws informally based on intuition. As most laws are very simple and intuitive, since each one deals with one construct at a time, we provided informal arguments describing why the two sides of the laws are equivalent. Hence, we generally described how to map an AspectJ construct to its corresponding Java implementation. Moreover, some laws when applied from right to left, perform a transformation very similar to the transformation applied by the AspectJ compiler to weave aspects and classes.

CHAPTER 4

EVALUATION

In this chapter we evaluate our laws using two approaches. First, we use our laws to derive some refactorings already defined in literature [23, 28, 32, 31]. This is useful to evaluate our laws and to show that those refactorings indeed preserve behaviour.

Second, we use our laws and some of the derived refactorings to restructure two Java applications, modularizing crosscutting concerns with AspectJ. We used our laws to ensure that the restructuring process did not change behaviour. For brevity, we omit the direction that each law is used assuming that all laws are applied from left to right.

4.1 DERIVING ASPECTJ REFACTORINGS

Several authors consider refactorings for aspect-oriented languages. Some of them [23, 28, 32, 31] show refactorings to transform Java programs into AspectJ programs, yielding results related to ours. However, they focus on describing large and global refactorings. Here we show that some of those refactorings can be derived from our laws. This is important to evaluate the laws, and show how they can be useful. Our intent is not to define new refactorings, but to provide some basis so that refactorings can be defined with some confidence that they preserve behaviour. Also, once a refactoring is in place, a developer uses it directly and does not need to be aware of the laws.

Some of the refactorings in the literature are basic and thus their derivation is not represented as a sequential composition of our laws. Instead, we represent them as a single law chosen from a limited set. For instance, we can use one of the laws related to creating a new advice (Laws 2, 4, 5, 6, 7, 3, 23, 24, 25, and 26) to accomplish the *Extract Advice* [23, 28] refactoring. Analogously, we use the Law 18 or 19 to accomplish the *Extract Introduction* [23] refactoring. Those refactorings define different transformations for distinct kinds of advice or inter-type declarations.

As most of the refactorings considered here create a new aspect, we assume that all of them use Law 22 and Law 1 to create a new empty aspect and make it `privileged`. Hence, the subsequent derivations do not show the application of these two laws. In addition, we only formalize the *Extract Pointcut* refactoring because it is simple enough to provide a readable law. The other refactorings would be difficult to read and understand, therefore we explain them by means of examples.

4.1.1 Extract Pointcut

The *Extract Pointcut* [28] refactoring does not deal with transformations from Java to AspectJ. This refactoring describes a transformation from AspectJ to AspectJ with the intent of increasing code quality. It creates a named pointcut from expressions used by advices and make the advices use the newly created pointcut, promoting code reuse. This refactoring also increase legibility by representing expressions with names that explains their intent.

Moreover, we formalized this refactoring as a law, providing its reverse refactoring as well. This happens because of the equivalence notion that allows the application of the refactoring on both directions. However, we only discuss its application from left to right.

Refactoring 1 shows the the proposed refactoring [28]. It transforms an anonymous pointcut (exp) used by a number of advices into a named pointcut (p) and changes all advices that use the same anonymous pointcut to reference the newly named one. Although the representation of the refactoring only deals with two advices, this refactoring can be extended to work with any number of advices. The used notation follows the one described in Chapter 3.

Refactoring 1 *Extract Pointcut*

$$
\boxed{\begin{array}{l}
\texttt{ts} \\
\texttt{aspect}\ A\ \{ \\
\quad \mathit{pcs} \\
\quad \mathit{bars} \\
\quad \texttt{before}(\mathit{ps}):\ \ \mathit{exp}\ \{ \\
\quad\quad \mathit{body} \\
\quad \} \\
\quad \mathit{bars}' \\
\quad \mathit{afs} \\
\quad \texttt{after}(\mathit{ps}):\ \ \mathit{exp}\ \{ \\
\quad\quad \mathit{body}' \\
\quad \} \\
\quad \mathit{afs}' \\
\}
\end{array}}
=
\boxed{\begin{array}{l}
\texttt{ts} \\
\texttt{aspect}\ A\ \{ \\
\quad \mathit{pcs} \\
\quad \texttt{pointcut}\ p(\mathit{ps}):\ \ \mathit{exp} \\
\quad \mathit{bars} \\
\quad \texttt{before}(\mathit{ps}):\ \ p(\alpha \mathit{ps})\ \{ \\
\quad\quad \mathit{body} \\
\quad \} \\
\quad \mathit{bars}' \\
\quad \mathit{afs} \\
\quad \texttt{after}(\mathit{ps}):\ \ p(\alpha \mathit{ps})\ \{ \\
\quad\quad \mathit{body}' \\
\quad \} \\
\quad \mathit{afs}' \\
\}
\end{array}}
$$

provided

($\rightarrow$) There is no pointcut named p in pcs;

($\leftarrow$) There is no reference to p in ts and A.

The derivation of this refactoring is simple. We start by applying Law 12 which creates a new named pointcut from exp. Then, it is necessary to apply Law 32 on every advice that uses the same expression represented by the new named pointcut. This law is responsible for changing an existing advice to use a named pointcut that declares the same expression used by the advice. We show a summary of the applied laws in Figure 4.1.

We can intuitively perceive that the advices after the transformation still captures the same set of join points as before. It is due to the fact that a named pointcut serves only to improve reuse of the pointcut expression and legibility inside the aspect. Thus, the involved advices point to the same expression they used before and captures the same

$$\boxed{\text{Law } 12 \quad \rightarrow \quad \overset{\curvearrowright}{\text{Law } 32}}$$

Figure 4.1. Extract Pointcut

set of join points. The final result should be exactly the same of the proposed refactoring [28] which implies that this refactoring can be considered to preserve behaviour regarding the equivalence notion provided by our laws and provided that the preconditions are respected.

We also derived the preconditions of the refactoring from the preconditions of the two involved laws. However, this is not always as easy as conjoining the preconditions of involved laws [44]. Sometimes preconditions of laws can not be satisfied from the beginning of the refactoring. They will be satisfied during the transformation, by applying other laws. In this case, just conjoining the preconditions of involved laws, would rise a precondition that can never be satisfied.

In order to illustrate this problem, consider a refactoring where we use Laws 22 and 1 to create an aspect and make it privileged. Then, we use Law 2 to create an advice `before-execution`. Law 2 has an implicit precondition based on the code template, it assumes the existence of a privileged aspect. Thus, this precondition is false at start. However it will be true after the execution of Laws 22 and 1.

As Law 32 has no preconditions, the preconditions for Refactoring 1 are the preconditions of Law 12 because none of those preconditions are satisfied by the previous application of another law.

4.1.2 Extract Method Calls

This refactoring intends to modularize calls to a method appearing in several other methods. This situation happens quite often. Suppose we use the object-oriented refactoring *Extract Method* [19] to extract code that was duplicated. Now that we restructured the code, we have another problem: there are several repeated calls to the extracted method. This problem may characterize a crosscutting concern.

One solution to the problem is to use aspect-orientation to modularize the method calls. In AspectJ, the concrete solution would be to create an aspect and define an advice that call the method at proper time [31]. The example shown is the same of the refactoring author [31]. It is part of a bank system that checks for user access on every method of the `Account` class.

```
public class Account {
  float balance;
  void credit(float amount) {
    Access.check(new BankPermission("account"));
    balance = balance + amount;
  }
  void debit(float amount) throws ... {
    Access.check(new BankPermission("account"));
    // verify balance and realizes the debit
  }}
```

In order to show the derivation we start from the code showed above, applying our laws to extract the calls to method Access.check. We assume the existence of a privileged aspect named PermissionCheckAspect. If this aspect do not exist, we can always use Laws 22 and 1 to create and make the aspect privileged.

We start by choosing the proper law concerned with advice execution based on where the method call appears. If it appears at the beginning of another method, we use Law 2. Analogously, if it appears at the end of another method we may consider Laws 4, 5, and 6 depending on each case. If none of those laws can capture the place where the method call is located, we should apply object-oriented refactorings to make the method call fit the template of one of the mentioned laws. In our example, we chose Law 2 because the call to method Access.check appears at the beginning of methods credit and debit. Further, we applied Law 2 once for each method, moving the referred method call to an aspect. The resulting aspect is showed next.

```
paspect PermissionCheckAspect {
  before(Account c, float amount) :
         execution(void Account.credit(float)) &&
         this(c) && args(amount){
    Access.check(new BankPermission("account"));
  }
  before(Account c, float amount) :
         execution(void Account.debit(float)) &&
         this(c) && args(amount){
    Access.check(new BankPermission("account"));
  }
}
```

Next, we must simplify our resulting aspect because it has repeated advices with the same action. Therefore we apply Law 9 that is responsible to merge the similar advices promoting a better reuse and legibility.

```
paspect PermissionCheckAspect {
  before(Account c, float amount) :
         (execution(void Account.credit(float)) &&
           this(c) && args(amount)) ||
         (execution(void Account.debit(float)) &&
           this(c) && args(amount)){
    Access.check(new BankPermission("account"));
  }
}
```

Then we use Laws 27 (Remove This Parameter) and 28 (Remove Argument Parameter) to remove the unused account and amount parameters.

```
paspect PermissionCheckAspect {
  before() :
         (execution(void Account.credit(float)) &&
           this(Account) && args(float)) ||
         (execution(void Account.debit(float)) &&
           this(Account) && args(float)){
    Access.check(new BankPermission("account"));
  }
}
```

Finally, we use the already discussed *Extract Pointcut* refactoring to transform anonymous pointcuts into named ones. In the following fragment, we show the resulting aspect and Figure 4.2 shows a summary of the applied laws.

```
paspect PermissionCheckAspect {
  pointcut accountPermission():
           (execution(void Account.credit(float)) &&
             this(Account) && args(float)) ||
           (execution(void Account.debit(float)) &&
             this(Account) && args(float));
  before(): accountPermission() {
    Access.check(new BankPermission("account"));
  }
}
```

Note that the code can be further simplified by reducing the pointcut expression. Although this is not our focus to provide such simplification, it would be a valid contribution. Simplifying the pointcut expressions would enable us to use other AspectJ features as wild cards for example. Additionally, a simplification would remove redundancies such as the repeated expression `this(Account) && args(float)`.

$$\boxed{\overset{\curvearrowright}{\text{Law } 2} \rightarrow \overset{\curvearrowright}{\text{Law } 9} \rightarrow \overset{\curvearrowright}{\text{Law } 27} \rightarrow \overset{\curvearrowright}{\text{Law } 28} \rightarrow \text{Extract Pointcut}}$$

Figure 4.2. Extract Method Calls

We showed the complete derivation of this refactoring, step by step. Hereafter we are not providing this kind of detail. We describe the derivation of other refactorings grouping the application of some laws in a single step.

4.1.2.1 Replace Override With Advice Sometimes we need to add some extra functionality to a class. One way to implement it is using inheritance and overriding similar to the implementation of the Decorator pattern [20]. This approach creates a new class inheriting from the class we want to 'decorate'. Next, it overrides the methods adding some behaviour and then calling the corresponding method on the super class. The *Replace Override With Advice* refactoring [31] is intended to transform the usual object-oriented implementation of this design pattern into a corresponding aspect-oriented implementation.

As argued by Laddad [31], this refactoring is a special case of the *Extract Method Calls* refactoring. Note that we can use the *Extract Method Calls* refactoring to move all the behaviour implemented by the Decorator class into an aspect. Therefore, the only code remaining on the subclass would be a call to the corresponding method on the super class. A final step is necessary to remove those overriding methods - as they do not add behaviour anymore. This can easily be achieved with object-oriented programming laws [10].

4.1.2.2 Extract Contract Enforcement Contract enforcement is generally implemented as pre and post-conditions for methods. It is possible to use Java assertions to implement the verification for those conditions. Moreover, if several methods have the same preconditions, those assertions can appear repeated, which means a crosscutting behaviour. The *Extract Contract Enforcement* [31] refactoring has the purpose of moving those assertions to an aspect. This refactoring is also a special case of the *Extract Method Calls* refactoring, where the extracted piece of code is generally an assertion. Our derivation of the *Extract Method Calls* can be used to achieve this refactoring because our laws can move any piece of code from a method to an advice, including assertions.

4.1.3 Extract Worker Object Creation

A worker object [32] is a class that encapsulates a method. An instance of this class is generally created only to be passed as an argument to a method that performs some operations and eventually call the worker object method. This situation is common when executing methods asynchronously, performing authorization using Java Authentication and Authorization Service (JAAS) API, implementing thread safety in Swing/AWT applications, and so on. Generally this is done by creating anonymous classes on demand, or by creating a considerable number of standard classes.

The *Extract Worker Object Creation* [31] refactoring is intended to modularize the worker object creation and simplify its usage logic. The following example shows an `ATM` class that uses the JASS authorization scheme by passing a worker object to `Subject.doAsPrivileged`. For simplicity, we omit the parameters of methods as they do not contribute to understand the example. More details can be found elsewhere [31].

```
public class ATM {
  ...
  public float getBalance(...) throws BankingException {
    PrivilegedAction worker = new PrivilegedAction() {
      public Object run() {
        // check privilege and getBalance action
      }
    };
    Float balance = (Float)Subject.doAsPrivileged(...);
    return balance.floatValue();
  }
  public void credit(...) throws BankingException {
    PrivilegedExcAction worker = new PrivilegedExcAction() {
      public Object run() throws Exception {
        // check privilege and credit action
      }
    };
```

```
      try {
        Subject.doAsPrivileged(...);
      } catch (PrivilegedActionException ex) {
        throw new BankingException(ex);
      }
    }
    ...
  }
```

As we see, the code is difficult to understand and the method's core logic is tangled within the anonymous classes. Following we show the refactored code. The authorization concern is now modularized in an aspect and the ATM class is much simpler.

```
public class ATM {
  ...
  public float getBalance(...) throws BankingException {
    // check privilege and getBalance action
  }
  public void credit(...)throws BankingException {
    // check privilege and credit action
  }
  ...
}
public aspect AuthorizationRouterAspect {
  pointcut authOperations(ATM atm)
       : execution(public * ATM.*(..)) &&
         this(atm) && within(ATM);
  Object around(final ATM atm) throws BankingException
       : authOperations(atm) {
    PrivilegedExcAction action = new PrivilegedExcAction() {
      public Object run() throws Exception {
        return proceed(atm);
      }
    };
    try {
      return Subject.doAsPrivileged(...);
    } catch (PrivilegedActionException ex) {
      return new BankingException(ex);
    }
  }
}
```

The example before the refactoring uses two distinct worker objects, one that completes execution without raising an exception and one that may raise an exception. The resulting aspect, after the refactoring, has only one advice that uses the second version of the worker object in both cases. Hence, the resulting aspect generalizes the use of the worker object to always be able to raise an exception, including the new worker object and exception handling code in the method that was not prepared to handle this exception before.

We did not derive this refactoring because a complex object-oriented transformation would be necessary. It consists of two steps: add a `try-catch` block for the

Subject.doAsPrivileged call; and then change the type of the PrivilegedAction to PrivilegedExcAction. Figuring out this transformation would help uncover the refactoring preconditions. For instance, if the PrivilegedAction was assigned to an attribute, it would not be possible to change its type, since some other part of the program could use type casts or tests. This is a complex transformation and its complete precondition would be difficult to discover. Assuming we applied this transformation, we would use a variation of Law 7 on the two methods, moving the worker object creation to an aspect. This would enable us to merge the two resulting advices using Law 9, and apply the *Extract Pointcut* to conclude the refactoring.

4.1.4 Extract Exception Handling

Exception handling sometimes can be considered a crosscutting concern because it may be repetitive. Moreover, it can handle exceptions that are not part of a method's logic, but are part of a crosscutting concern. The *Extract Exception Handling* [31] refactoring provides a modular implementation of exception handling using aspects.

Laddad [31] shows an example based on the *Business Delegate* [2, 3] pattern. Almost every method in a business delegate class catches exceptions thrown by the underlying implementation and re-throws an application-specific exception. Following we show part of the LibraryDelegate class from Laddad's example. This class implements the *Business Delegate* pattern.

```
public class LibraryDelegate {
    ...
    private void init() throws LibraryException {
        try {
            // remote home initialization;
            session = home.create();
        } catch (RemoteException ex) {
            throw new LibraryException(ex);
        }
        // other similar exception handling
    }
    public void addBook(BookTO book) throws LibraryException {
        try {
            session.addBook(book);
        } catch (RemoteException ex) {
            throw new LibraryException(ex);
        }
    }
    // other methods with identical exception handling code
}
```

The refactoring consists in softening the crosscutting exceptions with the declare soft construct of AspectJ. This construct wraps the checked exception into a SoftException, which is an unchecked exception (see Chapter 2). Then, the refactoring moves the exception handling code to an advice that intercepts the methods where the exceptions were softened. The resulting code is showed next.

```
public class LibraryDelegate {
    ...
    private void init() throws LibraryException {
        // remote home initialization;
        session = home.create();
    }
    public void addBook(BookTO book) throws LibraryException {
        session.addBook(book);
    }
    ...
}
aspect LibraryExceptionHandling {
    declare soft : RemoteException
        : call(* *.*(..) throws RemoteException) &&
          within(LibraryDelegate);
    \\ declare soft for other crosscutting exceptions

    after() throwing(SoftException ex) throws LibraryException
        : execution(* LibraryDelegate.*(..) throws LibraryException)
          && within(LibraryDelegate) {
        throw new LibraryException(ex.getWrappedThrowable());
    }
}
```

The *Extract Exception Handling* [31] revealed a number of issues. The example for the refactoring proposed by Laddad [31] is specific to cases where the handling code only wraps and re-throws another exception (as proposed by the *Business Delegate* pattern). Thereby, our solution as a composition of laws is more general allowing the extraction of different handling code for the same exception. Another weakness of the proposed refactoring is that it does not mention its preconditions. As a composition of our laws, we can derive the preconditions from the preconditions of each law involved. The derivation of this refactoring uses Laws 13, 14, 15, 16, and 17 as explained in Chapter 3. Figure 4.3 shows the sequence of laws necessary to achieve this refactoring.

$$\boxed{\overset{\curvearrowright}{\text{Law } 14} \longrightarrow \overset{\curvearrowright}{\text{Law } 13} \longrightarrow \overset{\curvearrowright}{\text{Law } 15} \longrightarrow \overset{\curvearrowright}{\text{Law } 16} \longrightarrow \overset{\curvearrowright}{\text{Law } 17}}$$

Figure 4.3. Extract Exception Handling

The preconditions for this refactoring are the preconditions of Law 17 conjoined with the preconditions of Law 14. This happens because the preconditions for the intermediate laws are always satisfied by the previous law in the sequence of laws that compose the refactoring.

However, generality implies less legibility on the final program due to the more complex code. Part of this complexity could be removed with laws for: merging `declare soft` constructs and simplifying pointcut expressions. Nevertheless, such simplifications were not our focus and are regarded as a future work. The resulting aspect using the above composition of laws can be seen next. Note that we also used Laws 27 and 28 for removing unused exposed context (this and args) and then Law 9 to merge the resulting advices.

```
aspect LibraryExceptionHandling {
    declare soft : RemoteException
        : execution(void LibraryDelegate.init());
    declare soft : RemoteException
        : execution(void LibraryDelegate.addBook(BookTO));
    // declare soft for RemoteException on other methods
    // declare soft for other exceptions

    void around() throws LibraryException
        : (execution(void LibraryDelegate.init()) &&
               this(LibraryDelegate)) ||
          (execution(void LibraryDelegate.addBook(BookTO)) &&
               this(LibraryDelegate) && args(BookTO))
           // expressions for other methods {
        try {
          proceed();
        } catch (SoftException se) {
          if (se.getWT() instanceof RemoteException) {
            RemoteException ex = (RemoteException)ex.getWT();
            throw new LibraryException(ex);
          }
          \\ if clauses for other crosscutting exceptions
        }
    }
}
```

4.1.5 Extract Interface Implementation

There are situations in which several classes implementing the same interface provide the same definition for some methods. A solution to that problem would be to turn the interface into an abstract class and make the abstract class provide such implementations. However, this is not always possible because some of the subclasses may already extend from another class. The *Extract Interface Implementation* [31] refactoring is intended to solve this problem by providing default implementations for interface methods. Following we show part of Laddad's example. It is based on the ATM application, and consists of an interface (ServiceCenter) and an implementing class (ATM).

```
public interface ServiceCenter {
    public String getId();
    public void setId(String id);
    public String getAddress();
    public void setAddress(String address);
}
public class ATM extends Teller implements ServiceCenter {
    private String id;
    private String address;
    ...
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    ...
}
```

Suppose there are other classes than `ATM`, which implement the `ServiceCenter` interface exactly the same way, using the same fields. This characterizes the problem described. The proposed solution uses inter-type declarations to move the necessary fields and method implementations to the interface. The resulting aspect and the ATM class are shown next.

```
public class ATM extends Teller implements ServiceCenter {
    ...
}
public aspect IMPL {
    private String ServiceCenter.id;
    private String ServiceCenter.address;
    public String ServiceCenter.getId() {
        return id;
    }
    public void ServiceCenter.setId(String id) {
        this.id = id;
    }
    public String ServiceCenter.getAddress() {
        return address;
    }
    public void ServiceCenter.setAddress(String address) {
        this.address = address;
    }
}
```

This code has been adapted from the original example. In Laddad's example, the resulting aspect is abstract and is declared as an inner class inside the `ServiceCenter` interface. In this case, there is no specific reason for the aspect to be abstract. Also, the use of a nested aspect to implement the refactoring is considered a matter of style. Thus, we consider those modifications not relevant to accomplish the refactoring results.

In order to derive this refactoring, we start moving the fields (Law 18) and methods (Law 19) - related to the interface implementation - to the aspect. Next we use Laws 20 and 33 to move the inter-type declarations from its base class to the implementing interface. We finish the refactoring by using Law 21 to remove the method implementations from other classes implementing the target interface, if they have the same implementation. The composition of laws used to derive this refactoring is showed in Figure 4.4.

$$\boxed{\overset{\curvearrowright}{\text{Law } 18} \longrightarrow \overset{\curvearrowright}{\text{Law } 19} \longrightarrow \overset{\curvearrowright}{\text{Law } 20} \longrightarrow \overset{\curvearrowright}{\text{Law } 33} \longrightarrow \overset{\curvearrowright}{\text{Law } 21}}$$

Figure 4.4. Extract Interface Implementation

A final step not included in Figure 4.4, would consider an object-oriented transformation, removing the unused fields from the implementing classes. The result of applying the *Extract Interface Implementation* [31] refactoring as a composition of our laws is the same of the proposed refactoring itself. Therefore, the composition of laws illustrates that the refactoring preserves behaviour provided that the preconditions from each law used to represent the refactoring are respected.

4.1.6 Extract Concurrency Control

Concurrency control is a good example of crosscutting concern. The code related to this concern is usually spread throughout several methods and tangled with several classes. Hence, concurrency is a good candidate to be modularized with aspects. The *Extract Concurrency Control* [31] refactoring has the purpose of aiding on this task. Following we show the banking example using the read-write lock pattern [32]. The `Account` class now uses the read-write lock pattern to control concurrent access to its fields and operations. The methods that only read the balance field (`getBalance` and `toString`) acquire a read lock, whereas the other methods acquire a write lock, all before their execution. Besides acquiring the lock, the methods must handle exceptions related to the concurrency control.

```
public class Account {
    // constructors, lock and other fields, etc.
    public void credit(float amount) {
        try {
            lock.writeLock().acquire();
            // business logic for credit operation
        } catch (InterruptedException ex) {
            throw new InterruptedRuntimeException(ex);
        } finally {
            lock.writeLock().release();
        }
    }
    public float getBalance() {
        try {
            lock.readLock().acquire();
            // business logic for getting the current balance
        } catch (InterruptedException ex) {
            throw new InterruptedRuntimeException(ex);
        } finally {
            lock.readLock().release();
        }}
    // other methods with similar concurrency control
}
```

The proposed solution creates an abstract aspect (`ReadWriteLockSynchronizationAspect`) that uses `before` and `after` advices to acquire and release the lock, respectively. Moreover the aspect also handles the exceptions related to the concurrency control. The aspect defines two abstract pointcuts `readOperations` and `writeOperations`. Therefore, it is only necessary to declare a new aspect (`ConcurrencyControlAspect`) extending from the abstract one and concretize the pointcut definitions. The definition of those two aspects is shown next.

```
public abstract aspect ReadWriteLockSynchronizationAspect perthis(
    readOperations() || writeOperations()) {

    declare soft : InterruptedException : call(void Sync.acquire())
        && within(ReadWriteLockSynchronizationAspect);
    public abstract pointcut readOperations();
    public abstract pointcut writeOperations();
    // inter-type declaration for lock
    before() : readOperations() {
        lock.readLock().acquire();
    }
    after() : readOperations() {
        lock.readLock().release();
    }
    before() : writeOperations() {
        lock.writeLock().acquire();
    }
    after() : writeOperations() {
        lock.writeLock().release();
    }
    after() throwing(SoftException ex)
        throws InterruptedRuntimeException :
               readOperations() || writeOperations() {
        throw new InterruptedRuntimeException(ex);
    }
}
public aspect ConcurrencyControlAspect
        extends SimpleSynchronizationAspect {
    public pointcut readOperations()
        : (execution(* Account.get*(..)) ||
           execution(* Account.toString(..)))
            && within(Account);
    public pointcut writeOperations()
        : (execution(* Account.*(..)) &&
           !readOperations())
            && within(Account);
}
```

Again we adapted the Laddad's example. In this case, we only included the `declare soft` construct inside the abstract aspect. The original source for the example uses a separate aspect to soften the exception.

We are not able to derive this refactoring exactly as it is presented because we did not deal with abstract aspects and the `perthis` construct. However, it is possible to derive a simpler version of this refactoring without those features. We start applying the *Extract Exception Handling* refactoring to move all the exception handling, related to the interrupted exception, to the aspect. Then, we use Law 18 to move the `lock` definition into the aspect. The use of inter-type declarations, in this case, behave similarly to the `perthis` construct for aspects with declared fields. Now we use Law 2 to move the lock acquiring code to `before` advices and Law 4 to move the lock releasing code to `after` advices. At this point, we have several advices with the same body, but capturing different join points. Thus, we use Law 9 to merge all the `before` advices and a similar version of this law to merge all the `after` advices. We can also use the *Extract Pointcut* refactoring to create named pointcuts for the advice expressions. The resulting aspect is shown next.

```
aspect ConcurrencyControlAspect {
    declare soft : InterruptedException
        : execution(void Account.credit(float));
    declare soft : InterruptedException
        : execution(float Acount.getBalance());
    // declare soft for other methods
    public pointcut readOperations()
        : (execution(void Account.credit(float)) &&
               this(Account)) || ... toString
    public pointcut writeOperations()
        : (execution(float Account.getBalance()) &&
               this(Account)) || ... other methods

    Object around() throws InterruptedException
        : (execution(void Account.credit(float)) &&
               this(Account) && args(float)) ||
          (execution(float Account.getBalance()) &&
               this(Account))
               // expression for other methods {
        try {
          return proceed();
        } catch (SoftException se) {
          if (se.getWT() instanceof InterruptedException) {
            InterruptedException ex = (InterruptedException)ex.getWT();
            throw new InterruptedException(ex);
          }
        }
    }
    before() : readOperations() {
        lock.readLock().acquire();
    }
    before() : writeOperations() {
        lock.writeLock().acquire();
    }
    after() : writeOperations() {
        lock.writeLock().release();
    }
    after() : readOperations() {
        lock.readLock().release();
    }
}
```

The *Extract Concurrency Control* [31] showed a limitation of our laws. As we did not deal with abstract aspects, the resulting code on the proposed refactoring is more reusable, as it uses an abstract aspect which provides an structure easily applicable in other cases. However, the transformation used to generate the abstract portion of the aspects can also be applied to the result we obtained using our laws. Besides, our result accomplishes the refactoring intension and provides better confidence that the transformation preserves behaviour. We intend to extend our set of laws to include the abstract constructs in the future.

4.1.7 Refactorings to Evolve Product Lines

A software product line (PL) consists of a set of products developed from the same set of artifacts and targeted at a specific domain [12]. Evolution of product lines is not easy. We consider an approach [6, 4] that initially extracts variation from an existing application and then reactively adapts the newly created product line to encompass another product variant. Both the extractive and the reactive tasks are supported by refactorings. This approach is evaluated in the context of an industrial-strength mobile game product line. In this section we show some of the refactorings that can be used to evolve a product line according to this approach. We use our programming laws to show that those refactorings preserve behaviour.

The adopted approach [6, 4] first bootstraps the product line and then evolves it with a reactive approach. Initially, there is only one product in the product line; this first implementation has been refactored in order to expose some variation. Next, the product line scope is extended to encompass another product: that is, the product line reacts to accommodate the new variant. During this step, not only refactorings are performed -maintaining the existing product- but also a product line extension that adds the new variant. At this point the product line may react to further extension or may be refactored.

This strategy relies on refactorings to accommodate the necessary changes to incorporate variations. During a case study building a product line for a mobile game [6, 4], some existing refactorings [19] were always applied together with *ad-hoc* aspect-oriented transformations, depending on the kind of variation being considered. Those grouped refactorings were considered as major transformations and represent the refactorings we are looking forward to prove. Moreover, the representation of those refactorings follows the notation we use to represent our laws.

The following refactoring is meant to be used when the considered variation is part of a method's body. The proposed solution extracts the variation into its own method. Then, it uses an AspectJ inter-type declaration to introduce the variation method. Note that, two different products relying on the considered variation, can now be implemented as two different aspects introducing the same method, but with different implementations.

This way, we can assembly different products, by weaving different aspects with a system core. Refactoring 2 already contains preconditions. However, those preconditions were defined during the derivation process and their origin is explained latter.

To derive Refactoring 2, we first need to apply an object-oriented refactoring. Deriving the refactoring from left to right, we apply *Extract Method* [19]. This refactoring creates a new method in class C called *newm* with proper parameters and return type, which executes the piece of code labeled as *body*′. The *Extract Method* can only be applied if the extracted code does not change more than one local variable, or else the extracted method would need multiple return values. In the opposite direction, we use *Inline Method* [19], which can only be applied if method *newm* is not polymorphic. The object-oriented refactorings can be proven to be sound using object-oriented programming laws [10].

Refactoring 2 Extract Method to Aspect

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
    body'
    body''
  }
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
    newm(αps');
    body''
  }
}
privileged aspect A {
  T' C.newm(ps') {
    body'
  }
  pcs
  bars
  afs
}
```

provided

(→) *body*′ does not change more than one local variable

(←) Method *newm* is not polymorphic

(↔) A does not introduce any field to C with the same name of a C field used in *body*′

Notice that the scenario after the method extraction is the left side of Law 19. If the target aspect already exists, we can apply this law to end the transformation. Otherwise, it would be necessary to use Laws 22 and 1 to create a new aspect and make it privileged. At this point we complete the derivation of Refactoring 2.

We considered two more refactorings to evolve product lines: *Extract Resource to Aspect* and *Extract Aspect Commonality*. We derived both refactorings as a composition of our laws and thus provided confidence that they preserve behaviour. We omit these derivations because they are similar to the process showed for Refactoring 2 and their demonstration would be tedious and repetitive.

4.1.8 Other Refactorings

Some of the analyzed refactorings could not be derived from our laws. This does not mean they do not preserve behaviour. In fact, it is generally a limitation of our set of laws. For instance, the *Extract Lazy Initialization* [31] refactoring, which is meant to modularize verifications of field initialization before using it, can not be derived from our laws because it relies on the `get` pointcut. As discussed in Chapter 3, we only cover the pointcut designators `call`, `execution`, `this`, `args` and `target`. We intend to extend our set of laws to cover the remaining pointcut designators, this is regarded as a future work.

The *Replace Argument Trickle by Wormhole* [31] refactoring also could not be derived from our laws because we did not deal with the `cflow` operator. This refactoring is intended to solve the problem of passing arguments only because they are needed in methods deep down the method chain. This parameter is not used by most of the methods to where it is passed. The proposed solution [31] removes the unused parameters form the method signatures and uses the `cflow` operator to expose the parameter when necessary. Our laws may expose the `caller`, `callee` and arguments of methods. It is possible to use `cflow` operator to enhance this context and expose an extra object in which the current join point is under the control flow. For this reason, this solution only applies when the method that uses the wormhole argument is under this argument's execution flow. That is, the wormhole object (object exposed with the `cflow`) originated the call to the method captured by the join point.

Note that most of the refactorings are called *Extract*. This is not a rule but an expected coincidence, since the mechanisms provided by AspectJ are generally used to extract some behaviour into an aspect. However, some of the laws used to derive the refactorings do not extract code. In fact, there are laws that insert code, for instance Law 14. In addition, some of the used laws just restructure the aspect to achieve better reuse and legibility.

4.2 REFACTORING TO ASPECTJ

This section shows a case study in which we use our laws and the refactorings derived in the Section 4.1 to restructure two distinct applications. Both applications were previously restructured to modularize crosscutting concerns using *ad-hoc* transformations. We use our laws to justify that the *ad-hoc* transformations preserve behaviour. This is another way to evaluate the laws. In the first case study, we discuss the concurrency crosscutting concern and in the second we discuss distribution. In both cases we successfully achieve the benefits of aspect-orientation.

4.2.1 Mobile Server

The Mobile Server is a commercial application that provides replication and synchronization of data that might be used off-line in different platforms (including mobile devices). It keeps information regarding changes made by users on each platform, solves conflicts with modifications made elsewhere and then propagates the resulting changes to all replicas.

In this system, one important part is the *Concurrency Manager*, which is responsible for coordination of data repository (a database with useful information) accesses. Thus its services are used by several modules, decreasing code legibility and making maintenance and extension harder. Figure 4.5 shows the components of the Mobile Server. The ones that access the repository need concurrency control.

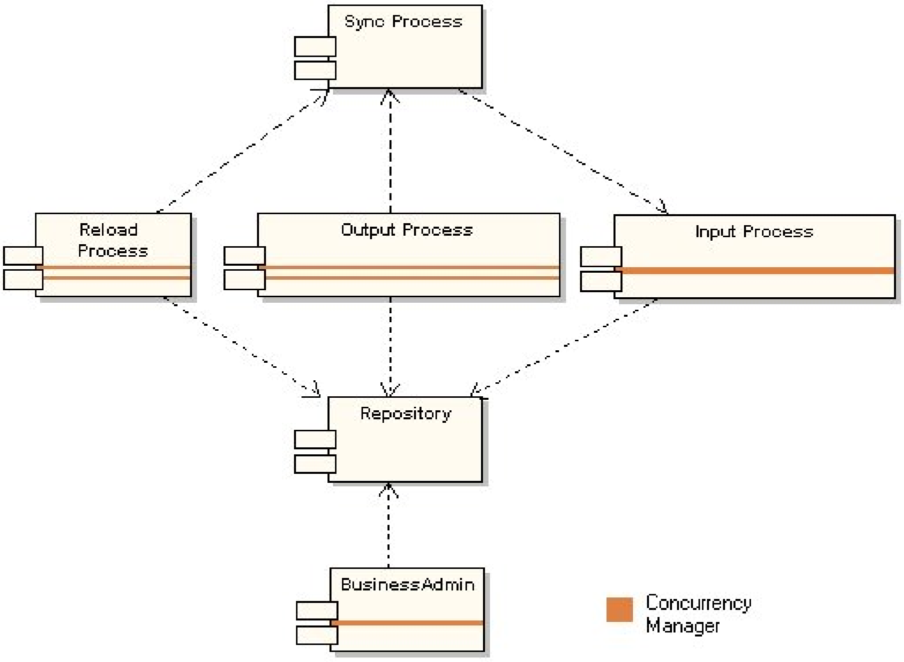


Figure 4.5. Mobile Server Before Refactoring.

A copy of the database is available in each platform allowing users to access the system off-line. As a result, the system needs to provide synchronization mechanisms between the central database and its local copies. There are two processes that carry out this responsibility. The first one is the *Input Processor*, which analyzes changes made on each local database and incorporates these changes in the centralized database. The second one is the *Output Processor*, which analyzes the database to collect changes that will be applied to the local databases. Those two processes respectively consume and produce files that are used by the *Synchronization* process, which is responsible to receive local changes from the device and send global changes from the system. The *Business Admin* process configures the database tables. The *Reload* process is used only in case a local database is lost (new device or device crash). In this case, it sends the complete database copy to the device.

This case study focus on separating the code related to the *Concurrency Manager* (CM) from all the other parts of the system, using aspects to provide the separation that could not be adequately achieved with object-oriented techniques. Once the crosscutting concern is identified, our strategy consists in two steps. First, we use object-oriented refactorings [19] for restructuring the code to enable the laws application (satisfy pre-conditions). Second, we apply a sequence of laws (refactoring). We start by removing the concurrency control from the *Output* and *Reload* processes. The following method `process` is part of both processes.

```
void process() {
  CM.beginExecution(this.id,this.tableNames,this.user);
  CM.sort(this.tableNames);
  for (int i=0; i<this.tableNames.length;i++) {
    CM.getNextLock(this.id,this.user);
    // different logic for each process;
    CM.releaseTable(this.id,this.getTableNames()[i],this.user);
  }
  CM.endExecution(user,user);
}
```

At first, we use object-oriented refactorings. We apply *Extract Method* to eliminate the use of local variables, and *Encapsulate Field* to ensure that the fields (`id`, `tableNames` and `user`) are used by its accessor methods. At this point we apply *Extract Method* once again, to provide the required join points to be used by the advices. It was necessary to extract the `processTable` method, which contains the logic applied to a single table. Now this new method is the only difference between the processes. So we applied *Pull Up Method* on two methods (both called `process`), isolating the concurrency control on an abstract super class. The resulting code is shown next.

```
public abstract class APThread {
  void process() {
    CM.beginExecution(this.getID(),this.getTableNames(),this.getUser());
    CM.sort(this.getTableNames());
    for (int i=0; i<this.getTableNames().length;i++) {
      CM.getNextLock(this.getID(),this.getUser());
      this.processTable(this.getTableNames()[i]);
      CM.releaseTable(this.getID(),this.getTableNames()[i],
                      this.getUser());
    }
    CM.endExecution(this.getID(),this.getUser());
  }
}
```

The process starts indicating to the CM the tables that will be required (`beginExecution`), the manager provides the order in which the process can use the tables (`sort`), finally the process waits for its turn to use each table (`getNextLock`). For each used table, the process notifies the manager to release it (`releaseTable`) and, after processing, the manager is notified to release all the remaining tables (`endExecution`). All those calls to CM are tangled with the process business code. Our goal is to modularize that code.

Once the system is restructured, we can start applying the laws. We use Laws 22 and 1 to create an aspect and make it privileged. Then we apply Laws 2 and 5 to create a new advice before and after the `process` method execution, moving the calls to the CM (`beginExecution` and `endExecution`) to the aspect. We also applied Law 8 to create a simple `around` advice from the pair of `before/after` advices. Next we apply Laws 3, 24 and 8 to introduce a new advice around a call to the extracted method `processTable`, moving the remaining calls to the manager.

```
public abstract class APThread {
  public void process() {
    for (int i=0; i<getTableNames().length;i++) {
      processTable(this.getTableNames()[i]);
    }
  }
}
```

The above code shows the resulting method without the concurrency control. The following code shows the aspect that is responsible to making the call to CM when necessary. We applied Law 11 to the second advice in order to remove the `target` parameter since this parameter was not used. We can now move the methods that are used only by the concurrency control code using Law 19. The only method moved was `getID`, which returns a constant of the CM class. Then we can start restructuring the aspect. To that matter, we use the *Extract Pointcut* refactoring that creates named pointcuts from the advice expressions and makes the advices refer to these pointcuts. This basically finish refactoring the *Output* and *Reload* processes.

```
privileged aspect CMAspect {
  void around(APThread c): execution(void APThread.process()) &&
                           this(c){
    CM.beginExecution(c.getID(),c.getTableNames(),c.getUser());
    CM.sort(c.getTableNames());
    proceed(c);
    CM.endExecution(c.getID(),c.getUser());
  }
  void around(APThread c, String table):
                           call(void APThread.processTable(String)) &&
                           this(c) && target(APThread) && args(table){
    CM.getNextLock(c.getID(),c.getUser());
    proceed(c, table);
    CM.releaseTable(c.getID(),table,c.getUser());
  }
}
```

The next process analyzed was the *Business Admin* process. This process is responsible for configuring the database managing the replicated tables. We started preparing the code to be refactored as we did before. In this case, we used *Replace Temp with Query* and *Extract Method* to eliminate local variables.

As this process uses only one table for each operation, it does not have a loop similar to the one showed in the previous process. Thus, we need only one advice that is responsible to make all the necessary calls to CM. This `around` advice would result from applying Laws 2, 5 and 8. The rest of the refactoring was exactly the same showed to the output and reload process.

The last affected module is the Input process which is responsible for processing the information received from the devices, solving conflicts and propagating this information to the centralized database. We used object-oriented refactorings to remove local variables and to provide the necessary join points to be used by the aspects. It was also necessary to change the way the CM was accessed. It was originally accessed through a field. However, it can be directly accessed (through static methods), without the need for a field.

```
public class IPThread {
  public void process(..) {
    CM.beginExec(..);
    CM.sort(..);
    try {
      \\(for loop similar to other cases)
    } finally {
      CM.endExec(..);
    }
  }
}
```

The prepared code ended with a structure slightly different from the other processes shown above. The notification of the end of execution appears inside a `finally` clause. This happens because the processing exceptions were not handled inside the method affected by the concurrency control. Hence, we cannot use an around advice as we did before, we must use before and after advices. The first notifies the beginning of the process and the second notifies its end. So, we use Law 2 to introduce the before advice and Law 4 to introduce the after advice. The remainder of the refactoring is identical to the refactoring of the *Output* and *Reload* processes.

Now that we have refactored out all the concurrency control code, there is still one last issue: exception handling. Therefore, we use the *Extract Exception Handling* refactoring, which moves exception handling code to an aspect. This refactoring is achieved from the sequence of laws shown in Figure 4.3. We used this refactoring on the exceptions related to the concurrency control. The resulting system is showed in Figure 4.6.

4.2.1.1 Discussion As most of the laws were not proven to be sound, we need another way to verify if the behaviour was preserved. Therefore, we built a test suite to exercise the concurrent actions on the *Repository*. A good test suit still does not guarantee correctness. However, tests are considered a good practice to verify if a refactoring preserves behaviour [19]. Our test suite exercises the *Repository*, creating situations where concurrency problems would arise. The execution of the test suite did not reveal any error. The system state was always coherent during the test execution and the operations were performed as expected.

We also monitored the system performance during the execution of the test suite before and after the restructuring. The refactored version showed a decrease in perfor-

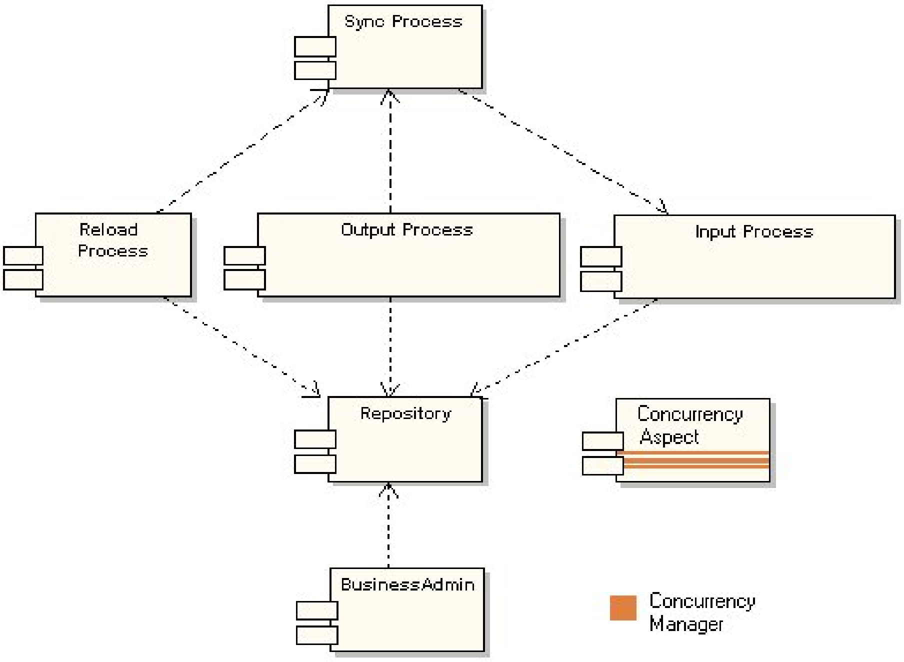


Figure 4.6. Mobile Server After Refactoring.

mance. However, the performance bottleneck is the access to the database. Thus, this difference was not relevant.

There were also benefits yielded by the use of AOP. For instance, the code separated from the concurrency control is cleaner and more legible, increasing systems maintainability. Unfortunately, we could not use the same advice for several join points in the Mobile Server and, as a consequence, there was not a relevant reduction on the number of lines of code. The aspects only affected four classes and the code extracted generated two aspects, the first responsible for the concurrency control and the second responsible for the exception handling.

4.2.2 Health Watcher

The Health Watcher is a real web based system intended to improve the quality of the services provided by health care institutions. By allowing people to register several kinds of health complaints, such as complaints against restaurants and food shops, health care institutions can promptly investigate the complaints and take the required actions. The system has a web-based user interface for registering complaints and performing several other associated operations.

In order to achieve modularity and extensibility, a layered architecture and associated design patterns [20, 5, 35] were used in the Java implementation of the system. This layer architecture helps to separate data management, business, communication (distribution), and presentation (user interface) concerns. The system also uses the Facade [20] design pattern to provide a single access point to the system business rules.

This structure leads to less tangled code – such as when business code interlaces with distribution code – but does not completely avoid it. This is the case of the code specifying the classes that have to be serializable for allowing the remote communication of its objects. The exception handling code is also scattered throughout the system.

We refactored this system to separate all code related to distribution from the other parts of the system. The system used a common implementation with Java RMI (Remote Method Invocation) [38] to make the system facade remotely available. It is a simple client/server implementation using RMI, where the server is the remote facade and the clients are the servlets that implements the user interface.

This implementation consists of an interface implemented by the facade class (HWFacade). This interface (IHWFacade) extends from the Remote interface and all of its methods must rise RemoteException. Moreover, it is necessary to make all the classes, used as arguments on the facade methods (i.e. Symptom), implement the Serializable interface. The last involved problem is about registering the remote facade on the naming service (server side) and retrieving it (client side).

We start separating the distribution code for the server side. Our first task is to move the serializable implementations to the aspect. In this case, the system only uses the Serializable interface to tag the classes that will be transported through the network. Thus, it is only necessary to move the interface declaration from the classes to the aspect. This is easily achieved with Law 29. We have to apply this law to every class that implements the Serializable interface. If there were transient fields or other behaviour related to serialization, we would use other laws to achieve the modularization.

Next it is necessary to move the remote implementation from the facade to the aspect. We use Law 30 to move the declaration of the Remote interface to the aspect. The following code shows part of the resulting aspect after this steps.

```
paspect ServerSideHW {
  declare parents: IHWFacade extends Remote;
  declare parents: Symptom implements Serializable;
  (implements declarations to other classes)
}
```

The final step on the server side is to move the code which registers the facade on the naming service to the aspect. We use Law 19 to move the main method on the facade class to the aspect, ending the separation of the distribution code on the server side.

```
paspect ServerSideHW {
  public static void HWFacade.main(String[]  args) {
    try {
      HWFacade facade = HWFacade.getInstance();
      UnicastRemoteObject.exportObject(facade);
      Naming.rebind("/HW",facade);
    } catch (Exception ex) { ... }
  }
}
```

The client side is a bit more complex. All of the client classes (servlets) look for the facade on the naming service to start using it. Another consequence of distribution is the `RemoteException` that is thrown by all methods in the remote facade, forcing the client classes to handle it. The first part can be achieved using Law 19 to move the facade initialization from the servlets to the aspect.

```
paspect ClientSideHW {
  public void InsertComplaint.initRemoteHW() {
    try {
      Object o = Naming.lookup(..);
      remoteHW = (IHWFacade) o;
    } catch (Exception e) { ... }
  }
}
```

At the end we use the *Extract Exception Handling* refactoring to move all the exception handling related to the `RemoteException` to the aspect.

We can still improve our result applying the *Extract Interface Implementation* refactoring showed in Section 4.1 to move the facade instance and initialization to the servlets superclass, eliminating the repeated code showed on the aspect which introduces the facade initialization on every servlet.

Although we could separate almost all the distribution code, distribution code is still part of the resulting program, since the facade interface still throws `RemoteException`. This remnant part could not be removed by our *Extract Exception Handling* because we do not have access to the stub implementation. The refactoring would remove the exception from the throws clause on the stub and then remove it on the interface. This was possible when we applied this refactoring to the Mobile Server because we had access to all the implementations of the affected interface.

4.2.2.1 Discussion We showed that the business code separated from the crosscutting concerns is cleaner and more legible, increasing the systems maintainability. Moreover, the aspects increased the systems modularity since the scattered code is now localized inside aspects. The new implementation also reduced the number of lines of code, due to the fact that the aspects have advices controlling several different join points. The code on the advices was repeated in every captured point. This reduction is more visible in cases where advices captures more join points. For instance, the client code to recover the remote facade was repeated in eighteen servlets.

In terms of affected classes, the Mobile Server case study was less representative. However, the Health Watcher case study affected 18 servlets on the client side plus 14 classes on the server side (the facade and serializable classes). In this case, we created three aspects, one responsible for the client side effects, other responsible for the server side effects. The last aspect is responsible for the necessary exception handling.

Another important consequence on the Health Watcher case study is that it can be generalized, since the distribution implementation of this application is commonly used.

Hence, we can generalize the steps used on this case study and derive a refactoring called *Extract Distribution* from the composition of the laws used here.

CHAPTER 5

CONCLUSIONS

We propose the use of programming laws for helping developers to deal with the problem of defining behaviour preserving transformations for AspectJ. Those transformations help to better modularize Java programs by using AspectJ constructs. Moreover, they are useful to restructure AspectJ programs because they represent simple transformations that grouped together create a refactoring. Therefore, the created refactorings can be used to restructure AspectJ programs increasing code quality.

We derive large and global refactorings from laws that are simple and localized. The refactorings derived are global because they usually affect many classes and aspect at once. Besides, our laws are localized because they generally change a singe class or aspect at a time. Our approach gives confidence that a transformation preserves behaviour because we intuitively show that each law preserves behaviour, and thus a composition of those laws also preserves behaviour.

We showed that the laws can be proved sound according to a formal semantics. We show that in detail for Law 2 (*Add Before-Execution*). For that, we use an operational semantics for Method Call Interception (MCI)[33], which can represent some of the laws but not all of them. The MCI language only supports the representation of `before-call` `before-execution` and `after-returning-execution` advices, including context exposition and expressions for matching method calls. Further, we only discussed soundness for the laws involving those constructs (Laws 2 – *Add Before-Execution*, 3 – *Add Before-Call*, 5 – *Add After-Execution Returning Successfully*, 9 – *Merge Advices*, 11 – *Remove Target Parameter*, and 27 – *Remove This Parameter*). To enable the proof of the remaining laws, we should define a completely new language along with its semantics, including all the AspectJ constructs covered by the laws. Another solution would be to extend an existing language (such as MCI) to incorporate the missing constructs.

In order to prove the laws we defined an equivalence relation stating when two programs behave in compatible ways. As MCI provides an operational semantics [33], we can represent evaluation of programs as a tree represented by the applied evaluation rules. Each rule describes the input state of the program, the expression to be evaluated, the returned result from the evaluation and an updated state. Our equivalence notion compares all the object fields, in the resulting program state, to state if two programs have the same behaviour. Thus, the proof is based on the evaluation of both programs according to the MCI semantics, and the analysis of the resulting evaluation trees, from both sides of the law, comparing the values of object fields included in the program state description.

We evaluated the laws by deriving refactorings already proposed in the literature. The derivation of those refactorings showed limitations of the set of laws we use. As mentioned in Section 4.1, the proposed *Extract Concurrency Control* [31] refactoring results in a more reusable code than we achieve with our laws. It makes use of abstract aspects providing a base to be reused in other applications. Although we do not deal with abstract aspects, as well as `get` and `set` pointcuts, we see no further difficulties on defining laws establishing properties of those constructs. Those laws would allow us to achieve a reusable solution with abstract aspects. They would also allow the derivation of refactorings not discussed here, such as *Extract Lazy Initialization* [31], which uses the `get` designator to initialize fields on demand. Additionally, the derivation of the *Extract Worker Object Creation* [31] refactoring showed the necessity for creating new object-oriented transformations. We could not derive this refactoring because it would be necessary to define a complex object-oriented transformation to prepare the code for the laws. This transformation would involve changing the type of a local variable, this may be difficult to prove. Also the associated preconditions to this transformation may not be trivial.

At last, we used our laws and the derived refactorings to transform two commercial applications separating a crosscutting concern with aspects. On the first case study, we successfully separated concurrency control from the core logic of the system. On the second case we considered to isolate distribution with aspects and once again we succeeded, despite the remnant exception on the remote interface. The second application uses a common implementation of distribution with Java RMI [38]. The remote class must implement an interface that extends from the `Remote` interface from RMI and every method must throw `RemoteException`. We could not completely remove this exception from the implemented interface, because our *Extract Exception Handling* refactoring would need access to the stub generated by the RMI compiler, and this is not possible.

Even though our set of laws is not complete in the sense it does not represent every feature of AspectJ, it is representative enough to derive several complex refactorings and to completely restructure common implementations of concurrency and distribution concerns.

The major contributions of this work are the following:

- Definition of aspect-oriented programming laws that are useful for creating aspect-oriented refactorings and formally deriving existing ones to increase the confidence that they preserve behaviour. Moreover, the laws are useful for guiding the implementation of aspect-oriented refactoring tools, for helping developers to better understand the language semantics, for justifying a compilation strategy, and for helping the teaching and learning process for AspectJ.

- Definition of an equivalence notion stating that two programs have the same behavior.

- Formal argumentation about the soundness of some laws, according to an existing aspect-oriented semantics, and the defined equivalence notion.

- Derivation from the laws of several existing refactorings proposed in the literature, proving that they preserve behaviour. This also pointed out some limitations to the set of defined laws.

- Usage of the laws and derived refactorings to modularize crosscutting concerns from object-oriented applications.

5.1 RELATED WORK

In this section we describe related work, singling out the relation to our work and the main differences to our approach.

Refactoring

The behaviour preserving property of refactorings is not easily proved. Opdyke [40] started showing that preconditions to apply the transformation would help on this task. Afterwards, based on the preconditions of the individual refactorings, Roberts [44] studied the composition of basic refactorings and the derivation of a single precondition to the derived refactoring. However, Opdyke and Roberts do not formally prove that the transformations preserve behaviour. Recently, Kniesel and Koch [30] specialized Roberts concept, deriving the composite precondition based on the weakest precondition of each individual refactoring. We use the concepts of preconditions to define our laws as behaviour preserving transformations. Further, our laws are intended to be composed, generating useful behaviour preserving refactorings.

Programming Laws

Algebraic laws for other paradigms, such as for imperative languages [27], have been addressed before. A similar work proposes some basic laws for ROOL [10], which is a refinement object-oriented language, as mentioned earlier. This related work is very similar to ours since they propose basic programming laws that are used to derive refactorings. In fact, we consider our work complementary to object-oriented programming laws. As discussed in Chapter 4, aspect-oriented refactorings generally depends on object-oriented transformations that can be proven with object-oriented programming laws. The laws proposed for ROOL and its relative completeness were formally proved, whereas we regard this proof for aspect-oriented programming laws as a future work.

Aspect-Oriented Refactoring

The second part of Hanenberg, Oberschulte and Unland's [23] research regards refactorings to AspectJ. In fact, they propose some new refactorings from Java to AspectJ. However, they only discuss the refactoring as a whole and the conditions to apply the refactoring. Our approach discusses those kinds of refactorings as basic laws of programming in order to simplify their understanding and proof. We also derived the proposed refactorings using our laws, showing that they preserve behaviour.

Iwamoto and Zhao [28] also show examples of refactorings from Java to AspectJ. However, there is no argumentation about necessary conditions to apply the refactorings to ensure that they preserve behaviour. We used the suggested refactorings and derived them as a composition of our laws. Hence, we were able to state in which conditions we can apply the refactorings as well.

There is a related work [31] that discusses aspect-oriented refactorings and the problems related to applying object-oriented refactorings in the presence of aspects. It proposes several complex and interesting refactorings and shows clear and easy to understand examples. We derived most of the proposed refactorings as discussed in Section 4.1.

Finally, another recent work [39] proposes a catalog of aspect-oriented refactorings. The refactorings are grouped by categories and are described similarly to the way Fowler [19] describes object-oriented ones. The proposed refactorings involve new refactorings in addition to the ones already proposed in the literature [31, 23, 28]. They rely on tests to verify if the refactorings do not change the program's behaviour. It would be interesting to derive the proposed refactorings using our laws, showing that they indeed preserve behaviour.

Aspect-Oriented Semantics

We use an existing operational semantics for Method Call Interception [33, 34] to represent aspect-oriented programming laws and reason about them. It seemed appropriate to choose this semantics because of its simplicity, its model of extending an object-oriented language, and its capacity to represent several types of advices from AspectJ.

However, there are other approaches for reasoning about aspect-oriented programs. It would be difficult to represent the laws using most of them. Douence et. al. [17] define a domain-specific language, along with its semantics, to define crosscuts based on execution monitoring. His system is based on events, similar to the Observer [20] pattern. Andrews [7] presents process algebras as a formal basis for aspect-oriented languages. He uses a subset of CSP tailored to this purpose, representing join points as synchronization sets. He also defines an equivalence notion between imperative programs and uses it to show the correctness of his weaving process. Wand et. al. [48] define a semantic model for dynamic join points. This is not appropriate to our purpose because we need a semantics in which we could represent AspectJ features. Xu et. al. [49] use a reduction strategy to transform aspect-oriented programs to implicit invocation. This transformation allows them to reason about the programs using already defined semantics for implicit invocation. Aldrich [1] discusses the problem of modular reasoning about aspect-oriented programs. He defines an aspect-oriented language and associated semantics where modular reasoning is possible. Finally, Barzilay et. al. [8] examine call and execution semantics in AspectJ and their interaction with inheritance.

We use the MCI semantics because it is much simpler than most of those approaches and it extends the semantics of an object-oriented language just as AspectJ extends Java, providing an easier understanding of how the semantics change from the object-oriented program to its aspect-oriented extension.

Restructuring Applications with Aspects

A previous work [45] used the same case study presented in Section 4.2.2. Although it presents specific guidelines on how to implement persistence and distribution as aspects by restructuring a pure Java system to an aspect-oriented one, it does not demonstrate that those guidelines are behaviour preserving. In fact, they are not refactorings, they introduce new behaviour. On the other hand our approach uses laws of programming in order to define refactorings, which are behaviour preserving. Here we restructure the system, therefore the system is supposed to be distributed.

Object-Oriented Refactoring with Aspects

Several authors discuss refactoring with AspectJ. Some of them [23, 28] address the problem of applying general object-oriented refactorings when using AspectJ. The problem arises from the fact that object-oriented refactoring usually changes the structure of join points of the program and thus changes how the aspects affect classes.

Hanenberg, Oberschulte and Unland [23] propose some preconditions to apply an object-oriented refactoring in the presence of aspects. Those conditions guarantee a mapping of join points during refactoring, which is necessary for preserving behaviour. They also propose modifications to refactorings such as *Extract Class* [19] in order to make them aspect-aware and therefore respect the preconditions. Analogously, Iwamoto and Zhao [28] propose modifications to existing refactorings in order to make them aspect-aware. However, they only show some examples and give some guidelines on how to avoid the aspect effects on the object-oriented refactorings. While their focus is on investigating how object-oriented refactorings are affected by aspects, our focus is on aspect-oriented transformations used to define or prove aspect-oriented refactorings.

Aspect-Oriented Refactoring Tools

Another related work [24] discusses a tool implemented to support the task of refactoring an aspect-oriented system. Their approach consists in developing a tool to be integrated with the Eclipse IDE. It is designed to involve the developer in a dialog to build the refactoring based on the concern description. The dialog is used to help on the necessary design and implementation decisions during the refactoring process. They have two approaches to achieve that. The first uses a concern graph to describe and implement the refactoring. The second, chooses a target design pattern from the GoF [20] and restructure it using aspects according to a previous work [25]. The implementations of those design patterns where also evaluated in a quantitative approach [21], helping to state which patterns are better implemented with aspects or purely with object-oriented techniques.

We also intend to implement our laws, providing tool assistance and automation. Our approach will extend JaTS [16], a Java transformation language, allowing it to represent AspectJ transformations.

This implementation would allow us to define simple code transformations corresponding to the execution of a law and apply this transformation to any piece of code that matches the template. However, this system does not provide a way to verify preconditions. The construction of such a verification system is also regarded as future work.

The use of tool support to apply our laws would allow us to define refactorings in terms of our laws and automate the application of those refactorings. This is a very important feature not only to developers intending to restructure aspect-oriented applications, but also to developers intending to introduce aspect-oriented feature to existing object-oriented applications.

5.2 FUTURE WORK

In this section we suggest some directions for future work.

Extend the Set of Laws

One limitation to our work is that we only deal with the pointcut designators `call`, `execution`, `args`, `this` and `target`. Moreover, it is necessary to extend the set of defined laws to incorporate constructs such as abstract aspects, `get` and `set` pointcuts, `cflow`, `cflowbelow`, etc.

It is also necessary to define more laws to restructure and simplify the aspects. For instance, laws for simplifying pointcut expressions are necessary to produce expressions with wild cards. For now, our laws always use a complete signature to describe join points and the composition of those expressions is not further simplified (See Law 9).

Although we deal with the `declare parents` and `declare soft` constructs, we do not provide laws to merge or simplify them. Merging laws would be similar to the laws for merging advices. In addition, this merging laws could also take advantage of laws for simplifying pointcut expressions.

Formal Proof and Completeness

We provided a formal argumentation to show that Law 2 (*Add Before-Execution*) is sound. However, it is necessary to provide a detailed formal proof. This proof would use induction on the structure of the main method (See Section 3.5). Such a proof is long and complex, and thus is beyond the scope of this dissertation. It is also necessary to prove all the other defined laws to ensure that they preserve behaviour. For now we rely on the simplicity of the laws to give an informal argumentation based on intuition. We discuss the manual proof of one law. Since it is an error-prone activity, it would be interesting to encode the used semantics [33] and our laws in a formal specification language, such as PVS [43], which has a theorem prover.

Another limitation to our proof is that we consider only dynamic semantics, and thus we can only show that a given law preserves behaviour. It is also necessary to consider static semantics. This would ensure that the laws relate well-formed and well-typed programs [47]. Our laws define preconditions to ensure that both, static and dynamic semantics, are preserved.

Once we extend the set of laws, it is necessary to prove that this set is complete. We could show that the set of laws is relatively complete. One way to prove that is to show that this set of laws is sufficient to reduce an arbitrary program to a normal form, similarly to a formal approach to object-oriented programming laws [10]. All those formalisms would help to enforce the validation and evaluation of the laws.

Deriving More Refactorings

We showed that our laws are useful to create new refactorings (see Section 4.2.2.1) and derive existing ones, with some confidence that they preserve behaviour. We also derived several refactorings already proposed in the literature. However, there are other interesting refactorings that need to be proved.

For instance, Monteiro and Fernandes [39] propose a catalog of aspect-oriented refactorings. It would be interesting to derive the proposed refactorings using our laws. Another work [25] provides the implementation of the Gof patterns [20] using AspectJ. It would be interesting to use our laws to show that the object-oriented implementation of the patterns is equivalent to its aspect-oriented implementations.

A recent work [50] reports the refactoring of a middleware system to modularize features such as client-side invocation, portable interceptors, and dynamic types. As a future work, we intend to systematize the refactoring applied using our laws. Thus, providing some confidence that the refactored middleware preserves behaviour.

APPENDIX A

LAWS

Law 22 - Add empty aspect

```
ts
```

=

```
ts
aspect A {
}
```

provided

(→) *ts* does not declare any class or aspect named A

(←) A is not referenced from *ts*.

Law 23 - Add After-Call

```
ts
class C {
  fs
  ms
  T n(ps') {
    try {
      exp.m(ps)
    } finally {
      body
    }
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T n(ps') {
    exp.m(ps)
  }
}
paspect A {
  pcs
  bars
  afs
  after(context) :
          withincode(σ(C.n())) &&
          call(σ(O.m()) &&
          bind(context) {
    body[cthis/this]
  }
}
```

provided

(→) *body* does not declare or use local variables; *body* does not call `super`;

(↔) A has the highest precedence on the join points involving the signature $\sigma(C.m)$; O is the type of *exp*; There is no designator `within` or `withincode` capturing join points inside *body'*;

Law 24 - Add After-Call Returning Successfully

```
ts
class C {
  fs
  ms
  T n(ps') {
    exp.m(ps);
    body
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T n(ps') {
    exp.m(ps)
  }
}
paspect A {
  pcs
  bars
  afs
  after(context) returning(T' t) :
        withincode(σ(C.n())) &&
        call(σ(O.m())) &&
        bind(context) {
    body[cthis/this]
  }
}
```

provided

(→) *body* does not declare or use local variables; *body* does not call `super`; T' is the return type of method m

(↔) A has the highest precedence on the join points involving the signature $\sigma(C.m)$; O is the type of *exp*; There is no designator `within` or `withincode` capturing join points inside *body*′;

Law 25 - Add After-Call Throwing Exception

```
ts
class C {
  fs
  ms
  T n(ps') throws es {
    try {
      exp.m(ps)
    } catch(E e) {
      body
      throw e
    }
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T n(ps') throws es {
    exp.m(ps)
  }
}
paspect A {
  pcs
  bars
  afs
  after(context) throwing(E e) :
        withincode(σ(C.n())) &&
        call(σ(O.m())) &&
        bind(context) {
    body[cthis/this]
  }
}
```

provided

(→) *body* does not declare or use local variables; *body* does not call `super`;

(←) *body* does not call `return`;

(↔) *A* has the highest precedence on the join points involving the signature $\sigma(C.m)$; *O* is the type of *exp*; There is no designator `within` or `withincode` capturing join points inside *body*′;

Law 26 - Add Around-Call

```
ts
class C {
  fs
  ms
  T n(ps') {
    body
    if (cond) {
      exp.m(ps)
    }
    body'
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T n(ps') {
    exp.m(ps)
  }
}
paspect A {
  pcs
  T' around (context) :
          withincode(σ(C.n())) &&
          call(σ(O.m())) &&
          bind(context) {
    body[cthis/this]
    if (cond) {
      proceed(αcontext)
    }
    body'[cthis/this]
  }
  bars
  afs
}
```

provided

(→) *body*, *body'* and *cond* do not declare or use local variables; and do not call `super`;

(←) *body* does not call `return`;

(↔) There is no aspect in *ts* affecting the join point $\sigma(C.m)$; O is the type of *exp*; There is no designator `within` or `withincode` capturing join points inside *body'*;

Law 27 - Remove this Parameter

```
ts
paspect A {
  pcs
  bars
  before(T t, ps) :
          this(t) && exp {
    body
  }
  bars'
  afs
}
```

=

```
ts
paspect A {
  pcs
  bars
  before(ps) :
          this(T) && exp {
    body
  }
  bars'
  afs
}
```

provided

(→) t is not referenced from *body*

Law 28 - Remove Argument Parameter

```
ts
paspect A {
  pcs
  bars
  before(P1 p1, ..., Pi pi, ...,
                  Pn pn, ps) :
          args(p1, ..., pi, ..., pn)
          && exp {
    body
  }
  bars'
  afs
}
```

=

```
ts
paspect A {
  pcs
  bars
  before(P1 p1, ..., Pn pn, ps) :
          args(p1, ...,  Pi, ..., pn)
          && exp {
    body
  }
  bars'
  afs
}
```

provided

(→) p_i is not referenced from *body*

Law 29 - Move Implements Declaration to Aspect

```
ts
class C impl D {
  fs
  ms
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
}
paspect A {
  declare parents : C impl D
  pcs
  bars
  afs
}
```

Law 30 - Move Extends Declaration to Aspect

```
ts
class C ext D {
  fs
  ms
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
}
paspect A {
  declare parents : C ext D
  pcs
  bars
  afs
}
```

Law 31 - Extend From Super Type

```
ts
class C′ ext C {...}
class D ext C′ {...}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C′ ext C {...}
class D ext C {...}
paspect A {
  declare parents : D ext C′
  pcs
  bars
  afs
}
```

Law 32 - Use Named Pointcut

ts		ts
paspect A {		paspect A {
pcs		pcs
pointcut $p(ps):\ exp(\alpha ps)$		pointcut $p(ps):\ exp(\alpha ps)$
$bars$	=	$bars$
before$(ps):\ exp(\alpha ps)$ {...}		before$(ps):\ p(\alpha ps)$ {...}
$bars'$		$bars'$
afs		afs
}		}

Law 33 - Move Method Up to Interface

ts		ts
interface D {...}		interface D {...}
class C impl D {...}		class C impl D {...}
paspect A {		paspect A {
pcs		pcs
$T\ C.m(ps)$ {	=	$T\ D.m(ps)$ {
$body$		$body$
}		}
$bars$		$bars$
afs		afs
}		}

provided

(→) Neither A or any other aspect in ts introduce a method named m to interface D

(←) Method m is not referenced in any implementation of interface D other then C

BIBLIOGRAPHY

[1] Jonathan Aldrich. Open Modules: A proposal for Modular Reasoning In Aspect-Oriented Programming. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, March 2004.

[2] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2001.

[3] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., 2003.

[4] Vander Alves. Identifying variations in mobile devices. In *GPCE'04 Young Researchers Workshop*, Vancouver, Canada, 2004. Available at: http://serl.cs.colorado.edu/~rutherfo/gpce_yrw04/program/alves.pdf.

[5] Vander Alves and Paulo Borba. Distributed Adapters Pattern: A Design Pattern for Object–Oriented Distributed Applications. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001.

[6] Vander Alves, Pedro Matos Jr., and Paulo Borba. An incremental aspect-oriented product line method for j2me game development. In *OOPSLA'04 Workshop on Managing Variability Consistently in Design and Code*, Vancouver, Canada, 2004. Available at: http://www.kircher-schwanninger.de/workshops/MVCDC/Submissions/Alves_Matos_Borba.zip.

[7] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, pages 187–209. Springer-Verlag, Sept 2001.

[8] Ohad Barzilay, Yishai Feldman, Shmuel Tyszberowicz, and Amiram Yehudai. Call and Execution Semantics in AspectJ. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, March 2004.

[9] Grady Booch. *Object–Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[10] Paulo Borba, Augusto Sampaio, Ana Lucia Cavalcanti, and Marcio Cornelio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, January 2004.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern–Oriented Software Architecture.* John Wiley & Sons, 1996.

[12] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns.* Addison-Wesley, 2002.

[13] Leonardo Cole and Paulo Borba. Using programming laws to modularize concurrency in a replicated database application. In *1st Brazilian Workshop on Aspect-Oriented Software Development - WBSOA'04 - SBES'04*, October 2004. At: http://www.cin.ufpe.br/~lcn/publications/WASP2004_Cole_Borba.pdf.

[14] Leonardo Cole and Paulo Borba. Deriving Refactorings for AspectJ. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, March 2005. ACM Press.

[15] Leonardo Cole, Paulo Borba, and Alexandre Mota. Proving aspect-oriented programming laws. In *FOAL 2005 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2005*, Technical Report. Department of Computer Science, Iowa State University, March 2005.

[16] Marcelo d'Amorim, Clóvis Nogueira, Gustavo Santos, Adeline Souza, and Paulo Borba. Integrating Code Generation and Refactoring. In *Workshop on Generative Programming, ECOOP02*, Malaga, Spain, June 2002. Springer Verlag.

[17] Remi Douence, Olivier Motelet, and Mario Sudholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, pages 170–186. Springer-Verlag, Sept 2001.

[18] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[19] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison–Wesley, 1999.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison–Wesley, 1994.

[21] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantitative study. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, March 2005. ACM Press. To appear.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison–Wesley, second edition, 2000.

[23] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35, Erfurt, Germany, September 2003.

[24] Jan Hannemann, Thomas Fritz, and Gail C. Murphy. Refactoring to aspects: an interactive approach. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, Anaheim, California, USA, October 2003.

[25] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and AspectJ. In *OOPSLA'2002*, pages 161–173, Seattle, Washington, USA, November 2002.

[26] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *3rd International Conference on Aspect-Oriented Software Development,AOSD'2004*, Lancaster, UK, March 2004.

[27] Charles Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[28] Masanori Iwamoto and Jianjun Zhao. Refactoring aspect-oriented programs. In Faisal Akkawi, Omar Aldawud, Grady Booch, Siobhán Clarke, Jeff Gray, Bill Harrison, Mohamed Kandé, Dominik Stein, Peri Tarr, and Aida Zakaria, editors, *The 4th AOSD Modeling With UML Workshop*, 2003.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[30] Günter Kniesel and Helge Koch. Static composition of refactorings. In Ralf Lämmel, editor, *Science of Computer Programming*, Special issue on "Program Transformation". Elsevier Science, 2004.

[31] Ramnivas Laddad. Aspect-Oriented Refactoring Series. TheServerSide.com, December 2003.

[32] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Company, 2003.

[33] Ralf Lämmel. A Semantical Approach to Method-Call Interception. In Gregor Kiczales, editor, *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, April 2002. ACM Press.

[34] Ralf Lämmel and Christian Stenzel. Semantics-Directed Implementation of Method-Call Interception. 46 pages; Accepted for publication in IEEE Proceedings Software; Special Issue on Unanticipated Software Evolution, November 2003.

[35] Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: Persistent Data Collections pattern. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, pages 161 – 173, Rio de Janeiro, Brazil, October 2001. UFRJ Magazine: Special Issue on Software Patterns.

[36] Kidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 17–26. Department of Computer Science, Iowa State University, April 2002.

[37] Bertrand Meyer. *Object–Oriented Software Construction.* Prentice–Hall, second edition, 1997.

[38] Sun Microsystems. Java Remote Methos Iinvocation (RMI). At: http://java.sun.com/products/jdk/1.2/docs/guide/rmi, 2001.

[39] Miguel Monteiro and João Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, March 2005. ACM Press.

[40] William Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Urbana-Champaign, IL, USA, 1992.

[41] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject–oriented composition. *TAPOS*, 2(3):179–202, 1996. Special Issue on Subjectivity in OO Systems.

[42] Harold Ossher and Peri Tarr. Using subject–oriented programming to overcome common problems in object–oriented software development/evolution. In *International Conference on Software Engineering, ICSE'99*, pages 698–688. ACM, 1999.

[43] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS language reference version 2.3. In *SRI International*, Technical Report, 1999.

[44] Donald Roberts. *Practical Analysis for Refactoring.* PhD thesis, Urbana-Champaign, IL, USA, 1999.

[45] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the OOPSLA '02 conference on Object Oriented Programming Systems Languages and Applications*, pages 174 – 190. ACM Press, November 2002.

[46] AspectJ Team. *AspectJ Programming Guide.* World Wide Web, http://www.eclipse.org/aspectj/.

[47] Frank Tip, Adam Kieżun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 13–26, Anaheim, CA, USA, November 6–8, 2003.

[48] Mitchell Wand, Gregor Kiczales, and Chris Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 1–8. Department of Computer Science, Iowa State University, April 2002.

[49] Jia Xu, Hridesh Rajan, and Kevin Sullivan. Aspect Reasoning by Reduction to Implicit Invocation. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, March 2004.

[50] Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 13–26, Vancouver, British Columbia, Canada, October 24–28, 2004.

Dissertação de Mestrado apresentada por **Leonardo Cole Neto** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Deriving Refactorings for AspectJ"**, orientada pelo **Prof. Paulo Henrique Monteiro Borba** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

Christina von Flach G. Chavez

Profa. Christina von Flach Garcia Chavez
Departamento de Ciência da Computação / UFBA

Prof. Paulo Henrique Monteiro Borba
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 28 de fevereiro de 2005.

Prof. JAELSON FREIRE BRELAZ DE CASTRO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.