Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2001

Using Aspect-Oriented Programming for Connecting and Configuring Decoupled Business Rules in Object-Oriented Applications

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: María Agustina Cibrán

Promotor: Prof. Viviane Jonckers (Vrije Universiteit Brussel)
Advisor: Maja D'Hondt (Vrije Universiteit Brussel)

Contents

Abstract

This thesis addresses the problem of decoupling business rules from the core software application. By business rules we identify constraints or statements of the kind "if then" that specify part of business knowledge.

The problem with business rules is that they are usually represented implicitly, scattered across many places in the code and even tangled with code that addresses other concerns. Moreover, they tend to change faster than the rest of the application to accommodate changes in business considerations. As a consequence, because of the way business rules are represented, it is hard to localize and change them, resulting in invasive changes to the code. Thus, the whole software becomes difficult to understand, maintain, evolve and reuse.

To deal with these problems separation of concerns is fundamental. The goal of this dissertation is to decouple business rules in order to achieve a high degree of flexibility by allowing easy plugging and unplugging of rules into the software applications depending on the business needs. We especially focus on connecting decoupled business rules with the rest of the application. It is out of the scope of this dissertation to focus on the representation of the business rules.

The first contribution of this thesis is the categorization of business rules followed by a detailed analysis of the requirements for a mechanism for decoupling business rules and their subsequent connection with the rest of the application. As many of these requirements coincide with the ideas behind Aspect-Oriented Programming, an aspect-oriented approach is used, using AspectJ as the concrete aspect-oriented programming language to perform the weaving of the rules into the applications.

As a second contribution we present a detailed description of the necessary ingredients for the decoupling, configuration and application of business rules. Generic solutions are provided and illustrated with examples from a case study.

Keywords: business rules, Aspect-Oriented Software Development (AOSD), Aspect-Oriented Programming (AOP), Separation of Concerns (SoC), AspectJ, decoupling, weaving.

Acknowledgements

I would like to thank to all the people that helped me in the development of this thesis.

Thanks to Prof. Viviane Jonckers for supervising this work and for her helpful advice.

Thanks to Maja D'Hondt for her constant help and guidance, for sharing her ideas with me and also for her support when I needed it.

Thanks to Gustavo Rossi for having given me the opportunity to do this master.

Thanks to Isabel, for having organized everything for us during our staying in Brussels.

Thanks in general to all the members of SSEL and PROG lab for having received us and for their patience and support during these months.

Being away from Argentina was not always easy for me. I would like to thank to all the people that accompanied me during this year.

Thanks to all the friends and relatives that supported me from Argentina. Specially thanks to mamá and papá because they were always near supporting me besides the physical distance. Thanks also to Fede for his advice when I needed it.

Thanks to Emmanuel, not only for his unconditional help and for having worked with me in many projects but also for his friendship and support.

Thanks to Kim Mens for his friendship, sense of humor, help and kindness.

Thank to all the other EMOOSErs, for having spent so nice moments together in Nantes.

Thanks to the new friends I made during my staying in Brussels, specially to Sebas, Boris, Miro, Maja, Isabel, Tielke and Raul for having shared so nice moments with me.

Finally, thanks to all the people I met during this year because they will make me remember this period with very nice memories.

Chapter 1

Introduction

1.1 Motivation

Software applications developed for a certain domain contain and use a lot of knowledge about that domain. The complexity to manage that domain knowledge is becoming more important as the complexity of software applications is increasing. By Domain Knowledge we understand a conceptual model containing concepts and relations between them. This model also defines constraints on the concepts and relations and rules that state how to infer new concepts and relations [SAA+99]. In the context of this thesis we refer to the concepts and relations as the business objects in the core application and to the constraints and rules as business rules.

Nowadays, due to current software engineering methodologies, business rules are expressed implicitly, strongly coupled and even tangled in the application code. Thus, changes in the business considerations imply invasively changing the code, making it hard to understand, maintain, evolve and reuse.

One problem that arises in real-world domains is the fact that business needs tend to change rapidly and thus business rules should reflect that change. However, it is desirable to easily change the business rules in a controlled and localized way, avoiding propagation of the adaptations to the core application. Moreover, it is also desirable to be able to adapt the core application without having to invasively change the business rules.

Another problem nowadays is that generally the software engineer plays two roles in the software development, the domain expert and the technology expert, dealing with different aspects of the software at the same time. This fact violates the principle of separation of concerns [Dij76] [Par72] [HL95]. Separation of concerns is a key principle of software engineering. It refers to the ability to identify, encapsulate, and manipulate only those parts of the software that are relevant to a particular concept, goal or purpose [OT01].

To conclude, it is clear that these aspects of a software application, i.e., the business rules on the one hand and the business objects on the other hand, should be expressed as decoupled as possible to allow software understanding, maintenance, evolution, adaptation and reuse.

1.2 Problem context

Even if there is no precise definition of what a business rule is, we follow the idea of considering it as a statement that defines or constraints some aspect of the business. It is intended to assert business structure or to control the behavior of the business [bus00].

Complexity in dealing with business rules is mainly based on the following considerations:

- *redundancy:* business rules inherently are global assertions about the model that involve multiple entities of it. Sometimes the same business rule is specified using different elements spread over many places in the model. However, we would like to express the rule as a single unit in only one place, not having to repeat it in many points of the system. For example the business rule stating that *"if the customer is a gold member of the shop then the price of all the products that are new releases is reduced in 5%"* specifies something that involves the products and the customers at the same time. Thus, it leads to the question of where to place this rule to avoid redundancy? in the Customer class? or in the Product class?

- *change management:* it is difficult to maintain and update business rules if they are spread over all the model. As it was identified in [CRFS01] [Arsa], business rules, as part of the domain knowledge information, evolve rapidly to cope with the volatile business considerations. Existent rules might change, new rules related with different concerns might be added, others removed, etc. Thus, it is desirable to keep them as much isolated as possible to facilitate their maintainability and to simplify their organization and management. Therefore, it is better to design rules as different artifacts in order to keep them independently from the rest of the application.

To deal with these characteristics, separation of concerns is fundamental. However, the separation of the concerns needed by business rules is not always easily and straightforwardly achieved. This is mainly caused by the well-known problem of the *"tyranny of the dominant decomposition"* [TOHS99]: modern languages and methodologies permit the separation and encapsulation of only one kind of concern at a time.

We think that the complexity in modelling business rules is mainly motivated by the fact that they represent concerns that cut across the basic functionality of the system. Business rules generally involve many and heterogenous elements defined in the system. Moreover, business rules represent concerns that can involve aspects of all these entities even at the same time. The nature of these problems reminds us to the idea of *"separation of concerns"*.

Due to the lack of support for separation of concerns in traditional software engineering techniques, when decomposing software into modules based on one dominant dimension of concerns, the software that addresses the concerns involved in business rules is not localized: it is *scattered* across many modules and within most of these, it is *tangled* with software addressing other concerns. This makes business rules difficult to reason about and hard to manipulate in a clean way.

Moreover, current software engineering techniques might not be the most suitable for expressing volatile business decisions. For instance when modelling a business using traditional Object-Oriented techniques, object behavior alone is not the most expressive way to describe business rules. Business policies may change independently of the business objects in the problem domain, or the functionality of end-user applications may evolve, while business rules remain valid.

Even though many approaches have been trying to express business rules at the analysis and design level, like for instance using OCL (Object Constraint Language) [OMG01] for expressing the rules, not much attention has been given to the mapping between the rules at a conceptual level and their concrete implementation in a software application. This is one of the reasons why in this dissertation we will focus on the implementation of business rules rather than in their representation.

Aspect-Oriented Software Development technologies have been successfully used to identify and capture concerns that cut across the basic functionality of the system and express them in a separate way. Many approaches have been developed, Aspect-Oriented Programming being one of the most accepted and used.

We will investigate the suitability of Aspect-Oriented Software Development techniques for the decoupling of business rules from a software application.

1.3 Focus and hypothesis

In this dissertation we will focus on the management of business rules as part of Domain Knowledge in object oriented applications. One of the main goals of this research is to define business rules independently of the behavior of business objects and decoupled from the way these business objects are used in the applications.

The issues we want to address are:

What are the most often encountered types of rules? How do we handle the proliferation of business rules? How can the addition or modification of business rules be simplified? How do we avoid tangling the business rules with the core application functionality? How do we address scalability (number and complexity of rules), performance, maintainability and extensibility? How do we guarantee software stability in this context?

Our research hypothesis is: *Business Rules, as part of business knowledge, tend to change much faster than the core business application and thus they should be as decoupled as possible. We believe Aspect-Oriented Software Development technologies are suitable for achieving the decoupling of the business rules and to afterwards weave them into the application code to obtain the desired working software application.*

Thus, the main goals of this thesis are:

- study and analyze how well Aspect-Oriented Software Development technologies can deal with the decoupling of business rules from the core application.
- provide generic solutions for the different issues involved in the management of business rules that can be used as methodological guidelines for the decoupling of business rules and their weaving into software applications.

1.4 Approach and contributions

In order to achieve the goals mentioned above, the following tasks had to be carried out:

- Study of the state of the art in Business Rules.
- Study of the state of the art in AOSD techniques, focusing on HyperJ and AspectJ.
- Classification of the different types of business rules.
- Identification of the set of features needed for modelling, designing and implementing business rules decoupled from the rest of the application.
- Focus on separation of concerns, not only decoupling concerns from the core application and the business rules but also stressing the decoupling even further, among the different issues involved in the management and configuration of the rules.
- Experimentation on a case of study using AspectJ (being the most suitable of the two techniques mostly studied and analyzed).
- Identification of generic solutions that involve object oriented abstractions and aspects to be considered as methodological guidelines for the decoupling of business rules from the core application.

- Analysis of the issues that can be easily achieved using AspectJ as the weaving language and the limitations that were encountered.

1.5 Organization of the dissertation

The dissertation is organized as follows: in chapter 2 we will present the state of the art in business rules by describing some of the typical issues involving business rules and some of the approaches addressing related work. In chapter 3 Aspect-Oriented Software Development is introduced and a description of AspectJ as an Aspect-Oriented Programming language is presented. Chapter 4 introduces the case of study used for the experiments in this dissertation. In chapter 5 a detailed analysis of the requirements needed for the management of business rules as well as a classification of business rules are presented. In chapter 6 we present generic solutions to the identified ingredients for the decoupling of business rules and their subsequent configuration, composition and application. These solutions are solved using AspectJ. Finally chapter 7 presents the conclusions of this dissertation and future work.

Chapter 2

Business Rules

Usually a software application implements actions that are regulated by the business considerations and uses business knowledge to guide the execution of these actions [Vae01].

In this chapter we concentrate on business rules as part of business knowledge. We present some examples of business rules, describe their typical problems and list some of the advantages achieved when rules are expressed decoupled from the rest of the application.

2.1 Definition and examples

Today, there is no exact and precise definition of what a business rule is. However in the literature many authors have agreed that a business rule can be defined as a statement that defines or constraints some aspect of the business. It is intended to assert business structure or to control the behavior of the business [bus00].

Typically a software application is driven by business knowledge. By business knowledge we understand concepts of the business, relations between those concepts, business processes, workflows and business rules. Business rules represent a dynamic part of the business knowledge because they specify which actions to take under certain conditions generally by specifying statements of the form "if then". Moreover, they tend to change faster than the rest of the business knowledge to accommodate to business requirements.

We can find many different kinds of business rules. Some business rules assert certain properties about certain business objects in the application. For instance, in the context of an application that models an electronic store we can find different examples of this type of rules:

"customers must be older than 18"

"every purchase must involve a minimum of 5 euros"

"customers must be registered in the shop to be able to perform purchases"

Other business rules might specify business decisions that involve behavioral considerations. These business rules, instead of stating some property that hold, state some active actions or behaviors that should be performed under certain conditions. Examples of this kind of rules are:

"if the customer buys more than 10 products, then apply 10% discount"

"if it is the day of the anniversary of the shop, then apply 20% discount"

"if the CD that is being bought is a classical music one, then apply 15% discount"

"if the customer buys the same product more than 10 times, then apply discount of 10%"

Moreover, as it is also identified in [IBMa], there are business rules that classify certain business objects or entities for a particular business situation. For instance, a customer may be classified into Gold, Silver and Bronze categories based on their spending history or the amount of money they have in their account. Some example rules are the following:

"if the customer has spent less than 50 euros in purchases in the shop, then consider the customer as a bronze member"

"if the customer has spent between 50 and 100 euros in purchases in the shop, then consider the customer as a silver member"

"if the customer has spent more than 100 euros in purchases in the shop, then consider the customer as a gold member"

Many other heterogeneous examples of business rules can be encountered and categorized according different dimensions. In chapter 5 we further analyze the different dimensions into which to classify the business rules and the features needed for their decoupling, configuration and connectivity with the core application.

2.2 Why business rules are important?

Business rules are important because they specify requirements for software applications that support the business [Cen01]. Business rules regulate how a business is run. They determine how the business is structured and organized.

Focusing on business rules to drive software development provides the potential for building better systems that are more responsive to the needs of the businesses they support, easier to modify, and less costly to maintain.

Depending on how the business chooses to implement its business rules, they affect business efficiency, productivity, and profitability.

2.3 Typical problems with business rules

- *Business rules are implicitly represented, scattered and tangled in the application code:*

  As we motivated in the introduction, the main problem that occurs when dealing with business rules is their implicit representation. Rules are usually implemented within the body of various methods that an object implements [Arsa], tangled with code that addresses other concerns.

  Moreover, rules can be scattered through different modules, being difficult to localize and maintain. Updates to frequently changing rules are difficult and error-prone making the overall lifecycle of introducing new business requirements, finding where to change the code and where to add new code to reflect the new requirements unacceptably time-consuming and extremely difficult.

- *Business rules tend to evolve quickly:*

  Changes to business requirements entails changes to the design and implementation of rules. Business rules are expected to change more frequently than the rest of the business objects as they need to reflect changes in the business requirements.

  Moreover, rules may be time-sensitive. This occurs when the rules in a domain are time-constrained; for example, they may pertain to some promotions only valid for a limited period of time. Such domains have a rapidly expanding and changing set of rules that may frequently change, sometimes on a day-to-day basis (see case study in chapter 4).

  Making intrusive changes to production code is unsafe and costly. If business rules are tangled in the code, changes in the business would imply intrusively changing the code, which can lead to unwanted side-effects.

- *Business rules are driven by business considerations:*

  Business rules reflect business requirements and thus they need to be created by and visible to management. The software engineer is the one that should map the rules and their changes to the implementation. Thus, encapsulating and localizing rules will help in the division of these roles.

To conclude with, business rules are expected to change more frequently than the rest of the business objects and thus the impact of these changes will be minimized if the rules are encapsulated and decoupled from the rest of the application. In this way, rules are ready to be plugged in and reused.

Some approaches have been proposed with the intention to decouple business rules from the application code. Section 2.5 describes some of the main approaches.

2.4 Why to decouple business rules?

We can mention the following list of advantages of decoupling business rules from the core application:

- *Explicit representation of business decisions:* By means of decoupling business rules, business decisions would be explicitly represented. Externalizing rules and representing them decoupled from the rest of the application makes it possible to view and understand them independently from the other concerns involved in the software applications.
- *Reuse of business rules across business processes:* Separating rules from the core application makes easier the reuse of a business practice decision in several applications in a consistent way.
- *Reuse of software applications considering different sets of business rules:* Separating rules from the core application makes it possible to reuse the same application with different business rules.
- *Clearer understanding of application behavior:* If the rules are decoupled, the application code is not tangled with the business logic and thus it becomes cleaner and easier to understand.
- *Decreased maintenance and testing costs:* Decoupled rules have a clearly defined scope since they are not tightly coupled to the application code. This makes them easier to modify and quick to test, decreasing costs and improving cycle time.
- *Improved manageability of business practice decisions:* Decoupling allows the control and managements over the definition of who can change rules and under what circumstances.
- *Simplify the identification of conflicting business rules in different parts of the business:* Having the business rules decoupled from the rest of the application helps to check conflicts or interferences between rules, i.e., whether rules being used in two different parts of an application, or even two different applications dealing with different parts of the business, are consistent.

2.5 State of the art in business rules

In this section some approaches in the literature that deal with the problem of decoupling business rules from the applications are addressed and described.

2.5.1 Business Rule Beans

The *Business Rule Beans* (BRBeans) [RDR$^+$] [IBMa] [Lou] approach focuses on the idea of identifying the points of variability in an application where business requirements expressed as business rules need to be applied. It mainly explores the idea of how to insert business rules into a software application rather than how to better express the rules.

The kinds of business rules in this approach can have the form of constraint rules (the ones that state some property that must be satisfied) or derivations (the ones that calculate some single value, generally a non-boolean value).

The points where the externalized rules are fired are called trigger points. A trigger point is a piece of code in an object method that interfaces with the BRBeans runtime to attach and execute business rules dynamically during application execution. This mechanism facilitates the dynamic attachment of externalized rules.

The trigger point selects which rules will be fired based on some criteria called trigger point context. The trigger point context can represent a static business situation (not depending on run-time data, for instance, the context defined by the scope of a class), a dynamic business situation (for instance, depending on the type of customer, information known at run-time) or a structural context (for instance, the context can specify the placement of a pre or post condition of a specific method).

The dynamic situation for the trigger point context is similar to the idea of defining rules that select other rules (approach presented in this dissertation in 5.1.1.2). The difference in our approach is that the selection of rules depending on dynamic information is modelled as another business rule, simplifying the number of concepts we need to manage.

The framework supports the insertion of trigger points into code by hand-coding, through the use of inheritance, or through code generation. The difference between this approach and the one presented in this dissertation is that in our approach, even if the aspects we define for connecting and configuring the business rules need to be hand coded, they are weaved in a non-invasive way into the core application. We avoid to invasively change the code to include the trigger points.

The BRBeans deal with some issues regarding the combination of rules. It is in the trigger point where the results of having applied the business rules are analyzed and the action to take is determined. This implies that the framework is flexible enough to apply several business rules in a given place and that some kind of post-decision is taken after the application of the business rule. However the solutions provided to solve the combination of rules are limited to some specific cases. The ways provided for combination of business rules are basically limited to the return of all results from all rules fired in an array, the return of only the first result or the last one and the return of the logical and/or of the results from all the rules fired. But it is also possible to add user-defined strategies for the combination of the results of a set of rules.

However, only little support for the combination of behavioral rules is provided. It is not possible to combine behaviors that personalize certain concerns in different ways. Thus, this approach does not deal with the conflicts that can arise between behavioral derivation rules. It is not expected to find in a derivation trigger point more than one applicable rule, otherwise it leads to an exception.

One important feature of the BRBeans framework is that it provides some common patterns that describe often encountered situations when producing rule-enabled applications [RDR$^+$]: fixed business context, situational business context, jurisdictional business, structural business context.

2.5.2 CommonRules

This approach emphasizes the clear and clean separation of data (represented by business objects) and logic (represented by rules) and the flexible and seamless connectivity between them [IBMb]. *CommonRules* has a classic inference engine. The rules are provided respecting the form of "if then".

CommonRules puts considerable emphasis to the representation of business rules by means of supporting declarative logic programs as a rule-based knowledge representation. As a knowledge representation language CommonRules uses Courteous Logic Programs, an extended form of declarative logic programs that allows the conflicts to be dealt with by the rule system itself.

Courteous logic programs allow the specification of partially-ordered priorities between rules. These priorities can be used to solve conflicts between rules. The prioritized conflict handling enables modularity and locality in updating, merging, specifying and maintaining rule sets. In this way, changes in rule sets can be simply specified by the addition of new rules, without having to modify previous rules.

In CommonRules it is also possible to ensure mutual exclusion between rules. It is possible to specify for instance, that discounting price by X percent is mutually exclusive with discounting price by Y percent (whenever X and Y are not equal).

Thus, CommonRules provides support for conflict resolution by means of two mechanisms: conditional overriding of prioritized rules and conditional mutual exclusions of conclusions from the main rules.

Moreover, CommonRules allows procedural attachments to business objects to associate predicates with methods in general business objects. More precisely, the procedural attachments provided in situated logic programs include:

- *effectors* that perform actions upon drawing conclusions in rule consequents.
- *sensors* that perform queries during testing of conditions in rule antecedents.

These effectors and sensors are specified by statements that "link" (i.e., associate) them to predicates. These effector and sensor link statements are treated as part of the knowledge representation.

Although with CommonRules it is possible to model rules that describe different business policies and add, remove and change them dynamically without recompiling, there are also some limitations in this approach. For instance regarding the kind of business rules that can be modelled, it is not always possible to add business rules that describe business considerations that were not anticipated from the beginning. For instance, it is not always possible to state rules about pricing if the concept of price was not already present in the original application.

To conclude with, this approach clearly emphasizes in the representation of the business rules. To link the rules with the rest of the application, the programmer has to specify files that map the declarative terms (expressing the rules) with the concrete attributes and objects in the code. Even if the linking of the rules is considered in this approach, the technology used is not as powerful as the capabilities of AspectJ, where we can easily capture dynamic points in the execution of a program to specify where the rules need to be plugged in.

2.5.3 Rule Object Pattern

The *Rule Object Pattern Language* [Arsa] contains patterns as solutions to common problems encountered during modelling, design and implementation of business rules. The aim of the *Rule Object pattern*, one of the most important patterns in the Rule Object Pattern Language, is to make the design of business processes extensible and adaptable, without endangering them with intrusive changes, by reifying the business rules and making them pluggable.

The idea of this pattern is to encapsulate the changing part of the business as Rule Objects with methods for their conditions and actions. As these conditions and actions tend to increase in number and variability, the pattern suggests to separate these out into their own classes. In this way, we can easily plug conditions and actions when the business imposes new requirements. The Rule Object Pattern uses the *Composite* design pattern [GHJV95] to design compound rule objects. The basic structure of the Rule Object Pattern is shown in Figure 2.1.

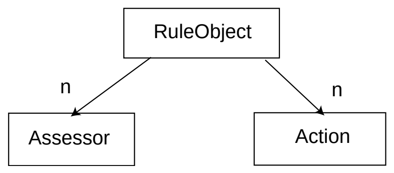


Figure 2.1: Rule Object Pattern

2.5.4 Grammar oriented object design by Arsanjani

The Rule Object Pattern suggests that the business objects should know their own business rules expressed as rule objects and manage them dynamically [Arsa]. This approach does not focus on the connectivity between the rules and the rest of the application. Thus, by only using the rule object pattern we would still have problems like the ownership of rules that involve different business objects, different scopes of rules, and other issues dealt in the approach presented in this dissertation. The rule object pattern focuses on the decoupling of business rules and their representation as first class entities but does not give special attention to the linking of the rules into the applications, main difference with our approach.

The rule object pattern has been used by some other approaches. For instance, it has been used to seamlessly map business models to component-based software architectures [Arsb]. This approach suggests the use of use-case grammars to express the business considerations for a given domain. Business rules are modelled as “pluggable” rules objects that can be adaptively plugged into a component business model without violating the open-closed principle.

Other uses of this pattern can be encountered in following approaches like the Adaptive Object models (see 2.5.5) and approaches where rule objects are used for the personalization of e-commerce applications (see 2.5.6).

2.5.5 Adaptive Object Model

Adaptive Object-Model (AOM) [YBJ] [YJ] architecture provides a dynamic implementation of business rules. In this approach business rules can change dynamically, emphasizing flexibility and run-time configurability.

Adaptive Object-Models provide an alternative to traditional object-oriented design. Contrary to traditional object-oriented design where business requirements are embedded in the system itself typically defined in term of classes, attributes, behaviors, and relationships, an AOM does not model business entities as first class objects. Instead, AOMs provide an architecture style for defining those objects, classes, attributes, behaviors, and relationships. AOM achieves this through the use of patterns: *TypeObjects* for defining classes, *Properties* for attributes, *Strategies* and *RuleObjects* for behaviors, and *Accountability* for defining relationships at runtime. Thus, for expressing business rules the AOMs follows the idea of the *Rule Object pattern* [Arsa] because it reflects the philosophy of the architecture suggested by this kind of systems.

The main characteristic of these systems is that they can easily evolve to reflect business considerations thanks to this flexible architectural style they are based on. In AOM's business information, in particular business rules, instead of being stored in code, it is usually kept in a shared database that the running systems have access to. Thus, new changes in the business considerations can be reflected by the system immediately without having to release a new version of the system.

Other approach based on the same idea as the Adaptive Object-Model is the Table-Driven approach [Per], in which business rules are defined and implemented as meta-data.

2.5.6 Business rules for the personalization of e-commerce applications

Some research has been done on the personalization of complex Web applications [CRFS01] [GRF$^+$01] [RFC] [KRS00]. These approaches focus on the identification of design constructs that can simplify the process of adding personalization features to a Web application. Some of these approaches try to decouple business rules from the Web applications, as this kind of applications is driven by business decisions that are usually expressed as business rules.

The approach presented in [CRFS01] focuses on the analysis of which design structures may help to cope with the increasing complexity of this kind of applications. In this approach different personalization patterns and design structures to obtain seamless extensions of existing software are presented. It emphasizes on behavioral personalization. This approach is based on the decoupling of business rules using design patterns [GHJV95] and the *Rule Object pattern* [Arsa].

Other research on the field is presented in [GRF$^+$01]. In this approach, the *SOUL* meta-programming framework [Wuy01] is used to express personalization business rules. SOUL integrates *Smalltalk* and *Prolog* by letting designers write Prolog rules in the meta-level governing the behavior of the base level. In this way, inheriting the well-known Prolog features, business rules can be expressed in SOUL in a modular and easy way. Like this, the addition and modification of business rules are simplified.

2.5.7 Event-centric Business Rules

In this approach an event-based decomposition of business rules is proposed. It focuses on e-commerce applications providing guidance on how rules may be decomposed into events and how their execution can be simplified by means of an event-based architecture. Some concrete implementation of the event-based architecture is the *ILOG Rules* framework [ILO] [AB]. These solutions are based on the use of a rule engine and include a rule language. ILOG Rules uniquely separates rules and the engine at the object level. This separation allows a rule engine to be connected to multiple rule sets, or a rule set to be used by any number of rule engines. Rules can be dynamically added, modified, or removed from the engine, without shutting down or recompiling the application.

This approach provides some support to deal with conflicts of business rules based on priorities. If several rules meet their conditions at the same time, the rule with the highest priority has precedence to be applied, temporarily forbidding the application of other conflicting or non-conflicting rules. Support for static or dynamic priorities is provided.

2.5.8 Coordination Contracts

This approach provides a clear separation between the computations performed by components and the mechanisms that coordinate their interactions [AFW]. These interactions between components are usually driven by business decisions expressed as business rules.

Generally the regulations that control the interaction between components are defined within the components themselves. This makes them fixed not being able to change them depending on the context. On the contrary, this approach tends to model the rules that determine the way the components need to interact in order to fulfil business requirements externalized from the way these components are implemented. Those regulations are expressed separately as a new abstraction called *Coordination Contracts*.

The aim of this approach is to extract the business rules that govern the interactions between components without intrusively modify the code that implements the components, i.e., preserving the black box property of the components. The definition of the coordination contracts only needs the information specified as part of the interfaces of the components. When the information provided as part of the interfaces is not enough and extra data is needed for the regulation and also when mismatches between the interfaces occur, adaptors are generated automatically for the adaptation of the components interfaces.

Coordination contracts act at the level of instances. This represents an advantage comparing with an approach based on aspects.

2.5.9 Constraint Enforcement

This research focuses on the automatic checking and enforcement of constraints in object-oriented applications. Generally constraints are expressed in the modelling level by means of a declarative language like the Object Constraint Language (OCL) [OMG01]. But it is still the responsibility of the programmer to ensure the checking of the constraints at the implementation level. This approach [Str00] facilitates the task of the programmer by means of automatically generating code that checks the validation of the constraints. This generated code is weaved into the core application using *AspectJ* as the weaving language. Analysis of the constraints is performed to detect places in the code where the constraints must be checked.

2.5.10 Aspect-oriented approach for Crosscutting Constraints

This approach is based on the idea that separation of concerns is fundamental not only at the implementation level but also at the design level. The same problems that result from tangled code at the programming level, also occur in the tangled constraints of the models [GBNT01]. This approach focuses on the modelling problem that arises when constraints that specify global system properties (such as latency, precision, timing) cross-cut the boundaries of the model hierarchy [GBN]. The same constraint is repeatedly applied in many different places of the model with slight variations.

Thus, the idea and motivation of this approach is to describe a common constraint in a modular manner and designate the places and conditions where it is to be applied. This approach uses an aspect-oriented approach by means of the definition of the specification of the constraints and their propagation in a separate module.

2.5.11 Business Logic Extraction

This research focuses on the analysis of the advantages of the extraction of business knowledge from business software applications. This approach is based on the use of *flow logic specifications* and *atomic services* to extract the business logic from the software applications [Vae01].

The atomic services are rule sets that represent the building blocks of encapsulated business knowledge. They are collections of rules that are related and are applicable in similar contexts. They represent atomic chunks of business knowledge potentially applicable in many business contexts.

However, as the logic needed for a certain context is spread out over many atomic services, it is needed to define the ways in which multiple atomic services should be combined and applied in a certain context and domain. This is achieved with the definition of flow logic specifications.

Thus, this approach suggests a software architecture that supports and facilitates extracted business logic as atomic services and flow logic specifications.

Chapter 3

Aspect Oriented Software Development

3.1 Overview

A software application involves many and heterogeneous concerns. By concerns we refer to properties or areas of interest in the system. Typically concerns can range from high-level notions like security and quality of service and low-level notions such as caching and buffering. They can be functional, like business rules or non-functional such as synchronization and transaction management [EFB01].

Separation of concerns (SoC) is fundamental to deal with all these heterogeneous concerns in a software application. Separation of concerns is a key principle of software engineering. It refers to the ability to identify, encapsulate, and manipulate only those parts of the software that are relevant to a particular concept, goal or purpose [OT01].

In many situations separation of concerns is not easy to achieve. Once software systems reach a certain complexity, the modularization constructs provided by current languages and environments fall short. Current software engineering techniques generally provide a dominant decomposition mechanism that is not suitable to capture and represent all kinds of concerns that can be found in a software application. This problem is identified in [TOHS99] as the "*tyranny of the dominant decomposition*": modern languages and methodologies permit the separation and encapsulation of only one kind of concern at a time. Examples of tyrant decompositions are classes in object-oriented paradigm, functions in functional paradigm, and rules in rule-based systems. Therefore, it is impossible to encapsulate and manipulate all the diverse and heterogeneous concerns in only one of the decomposition mechanisms.

Unfortunately, as it has been recognized by the Aspect-Oriented Programming (AOP) community [KLM+97], the “system-wide” concerns found in the implementation of any complex system, often do not fit nicely into the modularization mechanism of the technique used. These concerns are said to *cut across* the natural modularity of the rest of the implementation [KHH+01a]. Common examples of crosscutting concerns are design or architectural constraints, systemic properties or behaviors (e.g., logging and error recovery), and features.

As a consequence, when software is decomposed into modules based on only one dominant dimension of concern, the code that addresses other concerns is not localized. On the contrary, that code is *scattered* across many places in the entire application and usually they are even *tangled* with software addressing other concerns as they cannot be mapped into the dominant decomposition abstractions.

The problem presented so far motivates the need for extra support to achieve separation of cross-cutting concerns. Separation of concerns suggests the possibility to work with the design and implementation of a system in natural units of concerns rather than in units imposed by the tools or languages that are being used. In other words, the idea is to adapt the modularity of a system to reflect the way the software engineer thinks about a problem rather than to adapt the way to think to the way languages and tools do it [KHH+01a].

Aspect-Oriented Software Development (AOSD) appears as a new technology that focuses on the separation of cross-cutting concerns. AOSD allows us to better program software applications by separately specifying the various concerns of a system and some description of their relationships, and then providing mechanisms to weave or compose them together into a coherent program [EFB01].

Firstly, separation of concerns was more oriented towards the implementation, dealing with concerns that can appear tangled in the code. Nowadays the AOSD community recognizes the need of separation of concerns through the whole software development cycle. Cross-cutting concerns may arise at any stage of the software lifecycle, including requirements specification, design, implementation, etc.

To conclude, AOSD helps to avoid tangling of concerns by explicitly capturing, representing and manipulating them as first entities. In this way, the target application code is easier to develop, understand, reason about, maintain and reuse.

3.2 Approaches

Many approaches and techniques that enable Aspect-Oriented Software Development are being subject of recent research. Basically we can identify two different philosophies represented by these approaches. Some of the approaches are more dynamic in the sense that they allow to capture specific points in the execution of a program where the crosscutting concerns must be weaved whereas others are more static or structural, providing support for structural mappings between units of a program. This main difference between the approaches influences our choice of which approach would be suitable for the kind of problems we need to solve when decoupling business rules and composing them with the rest of the application.

Among the most accepted AOSD approaches to deal with cross-cutting concerns at the implementation level we can mention *Aspect-Oriented Programming*, approaches for *multi-dimensional separation of concerns*, *Composition Filters* and *Aspectual Components*. For some of these approaches concrete systems (languages or tools) have been developed whereas for others only research prototypes exist.

Examples of the most developed and used tools are *AspectJ*, the representative example language for the ideas supported by Aspect-Oriented Programming, and *HyperJ*, a system that supports "multi-dimensional" separation and integration of concerns. Both enable the modular implementation of crosscutting concerns in standard Java software. *AspectJ* is a good representative example for the languages that support dynamic join point models whereas *HyperJ* is a good example of the other (more structural) approach.

As we will see along this dissertation, we will have to manage dynamic information when dealing with business rules. For instance, we will need to capture dynamic points in the execution of the program to specify the places where we need to plug the rules. This motivates the need for a sophisticated mechanism to capture these dynamic points. Approaches like *Composition Filters* and *AspectJ* would be suitable in this sense.

In this thesis we choose *AspectJ* as a concrete weaving language that supports a dynamic join point model. *HyperJ* is also studied and analyzed as we think it would be suitable for expressing business rules that imply mappings between concepts. However, it is important to notice that the mechanism to identify weaving points provided by *HyperJ* is not as powerful as the one provided by *AspectJ*.

3.3 Aspect-Oriented Programming as a post-object programming mechanism

Currently, one of the dominant programming paradigms is the Object-Oriented programming (OOP). The idea behind this paradigm is to build a software system by decomposing a problem into objects and then writing the code of those objects. Such objects abstract together behavior and data into a single conceptual entity.

Although Object-Orientation has been used with success in many complex applications keeping them maintainable and comprehensive, it presents some limitations. The hierarchical modularity mechanisms of Object-Oriented languages are very useful but they are inherently unable to modularize all kinds of concerns of interest that exist in a complex software application.

Many mechanisms that look to increase the expressiveness of the OO paradigm have been considered. They are called post-object programming (POP). Aspect-Oriented programming (AOP) is one of those. AOP focuses on mechanisms for simplifying the separation of cross-cutting concerns in software applications. AOP provides a mechanism to explicitly capture crosscutting structure, expressing crosscutting concerns in a modular way.

AOP addresses the problem of the dominant decomposition by implementing the "base" program (addressing the dominant concern) and several aspect programs (each addressing a different cross-cutting concern) separately and then "weaving" them all together automatically into a single executable program.

In this section we will describe the characteristics of AspectJ, one of the most representative examples of AOP languages.

3.4 AspectJ as an example of AOP

AspectJ is a simple general-purpose extension to Java that provides, through the definition of new constructors, support for modular implementation of crosscutting concerns. It enables plug-and-play implementations of crosscutting concerns [KHH$^+$01b].

AspectJ has been successfully used for cleanly modularize implementations of crosscutting concerns such as tracing, contract enforcement, display updating, synchronization, consistency checking, protocol management and others.

In the following sections we will present a description of the features and capabilities provided by *AspectJ*. This description is mostly based on [Asp] [KHH$^+$01b] [KHH$^+$01a].

3.4.1 Join Point Models in AspectJ

The join point model in an AOP language provides the common frame of reference for the structure of crosscutting concerns.

AspectJ provides support for two types of crosscutting implementations defining two join point models: static and dynamic.

3.4.1.1 Dynamic crosscutting

Dynamic crosscutting makes it possible to define additional implementation to run at certain well defined points in the execution of a program. Dynamic crosscutting is based on a small but powerful set of constructs:

- *Join Points* are well defined points in the execution flow of a program where aspects crosscut the normal execution with the crosscutting concerns.
- *Pointcuts* are a means to make reference to a set of join points and to manipulate certain values captured in those join points.
- *Advices* are modules, like methods, that encapsulate the crosscutting implementation upon pointcuts.
- *Aspects* are units of modular crosscutting implementation. It is composed of pointcuts, advices, and ordinary Java member declarations.

3.4.1.2 Static crosscutting

AspectJ also allows us to implement static crosscutting. Static crosscutting allows the definition of new operations on existent types. It affects the static structure of the program. This is achieved using forms called *Introduction.*

An *Introduction* is a member of an aspect, but it defines or modifies a member of another type (class). The main actions we can achieve with *Introduction* are the following ones:

- add fields to an existing class. For example:

  ```
  public int Point.x = 0;
  ```

- add methods to an existing class

  ```
  public int Point.getX() { return x; }
  ```

- extend an existing class with another

  ```
  declare parents: Point extends GeometricObject;
  ```

- implement an interface in an existing class

  ```
  declare parents: Point implements Comparable;
  ```

Introduction is a powerful mechanism for capturing crosscutting concerns because it not only changes the behavior of components in an application, but also changes their relationships.

3.4.2 AspectJ's building blocks

Aspect-Oriented programming languages have three main elements for expressing crosscutting concerns: a join point model, a means of identifying join points, and a means of affecting implementation at join points. These elements are based on a set of constructs that constitutes the building blocks for the specification of crosscutting implementation. These constructs are called: *aspects*, *join points*, *pointcuts* and *advice*.

A brief introduction to the main features of AspectJ for supporting crosscutting implementation is presented in the following sections.

3.4.2.1 Join Points

AspectJ's internal mechanism is based upon the join point model. *Join points* can be considered as nodes in a simple run time object call graph. These nodes include points at which an object receives a method call and points at which a field of an object is referenced. The edges are control flow relations between the nodes. In the dynamic join point model, control passes twice through each join point, once on the way in to the sub-computation rooted at the join point and once on the way back out.

Let's illustrate this concept using a simple figure editor as an example. In the Figure 3.1, large circles represent objects, square boxes represent methods defined upon the class of those objects and small numbered circles represent join points.

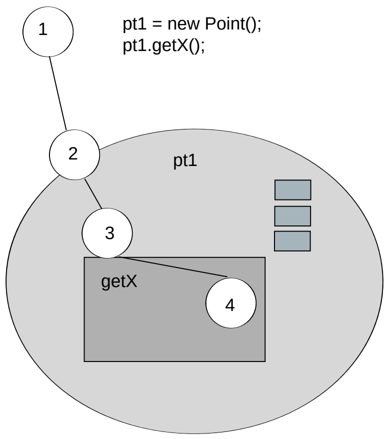


Figure 3.1: Dynamic join point model when the method getX is called

The following types of join points exist in AspectJ. Some of them are illustrated on the example:

- *Method calls*: Points in the execution of a program where a method is called. E.g. (1): A method call join point corresponding to the method `getX()` called on the object pt1.

- *Constructor calls*: Points in the execution of a program where a constructor of a class is called.
- *Method call receptions*: Points in the execution of a program where an object receives a method call. E.g.(2): A method call reception join point at which pt1 receives the call to the method getX().
- *Constructor call receptions*: Points in the execution of a program where a class receives a constructor call.
- *Method executions*: Points in the execution of a program where an individual method is invoked. E.g.(3): A method execution join point at which the particular method getX() defined in the class Point begins executing.
- *Constructor executions*: Points in the execution of a program where an individual constructor is invoked.
- *Field gets*: Points in the execution of a program where a field of a class, object or interface is read. E.g.(4): A field get join point where the field x of the object pt1 is read.
- *Field sets*: Points in the execution of a program where a field of a class, object or interface is set.
- *Exception handler executions*: Points in the execution of a program where the exception handler of a method is invoked.
- *Class initialization*: Points in the execution of a program where the static initializers of a class are run.
- *Object initialization*: Points in the execution of a program where the dynamic initializers for a class are run during object creation.

3.4.2.2 Pointcuts

In *AspectJ*, a *pointcut* is a program element that identifies a particular subset of join points in the program flow and optionally some of the values in the execution context of those join points.

AspectJ includes several primitive pointcut designators. Programmers can combine these primitive pointcut designators using the boolean operators &&, || and ! to specify anonymous or user-defined pointcut designators.

Two kinds of pointcut designators exists in *AspectJ*:

- **Primitive pointcut designators**

  Some primitive pointcuts only identify join points of one kind. For instance, *receptions* only matches method call reception join points.

  Examples of this kind of primitive pointcuts are:

- `call(Signature)`

  Picks out a method or constructor call join point based on the static signature.

- `execution(Signature)`

  Picks out a method or constructor execution join point based on the static signature.

- `get(Signature)`

  Picks out a field get join point based on the static signature.

- `set(Signature)`

  Picks out a field set join point based on the static signature.

Other kind of primitive pointcuts are the ones that match any kind of join points at which a certain property holds. For example *instanceof(Point)* matches all join points at which the currently executing object (the value of this) is an instance of Point or a subclass of it.

- `within(TypePattern)`

  The within pointcut picks out all join points where the code executing is defined in the declaration of one of the types in TypePattern.

- `withincode(Signature)`

  The withincode pointcut picks out all join points where the code executing is defined in the declaration of a particular method or constructor.

- `cflow(Pointcut)`

  Picks out all join points based on whether they occur in the dynamic context of Pointcut, including pointcut's join points themselves.

  For example, `cflow(move())` identifies all join points that occur between receiving calls for the methods specified in the pointcut move() and returning from those calls (either normally or by throwing an exception.) (see definition of pointcut move() in 3.4.2.2)

- `cflowbelow(Pointcut)`

  Picks out all join points in the control flow below the join points picked out by the pointcut.

- `this(TypePattern or Id)`

  Picks out all join points where the currently executing object (the object bound to this) is an instance of a type that respects TypePattern, or of the type of the identifier Id. It will not match any join points from static methods.

– `target(TypePattern or Id)`

  Picks out all join points where the target object (the object on which a call or field operation is applied to) is an instance of a type that respects the TypePattern, or of the type of the identifier Id. It will not match any calls, gets, or sets to static members.

– `args(TypePattern or Id, ...)`

  Picks out all join points where the arguments are instances of a type of the appropriate TypePattern or identifier Id.

– `if(BooleanExpression)`

  The *if* pointcut picks out join points based on a dynamic property. It picks out all join points where the boolean expression received as an argument evaluates to true.

These two kinds of primitive pointcuts can be combined to identify join points in useful ways. For example:

```
!instanceof(FigureElement) &&
 calls (void FigureElement.incXY(int, int))
```

matches all calls to method incXY that do not originate from an object that is an instance of FigureElement or any of its subclasses.

Notice that the examples presented so far correspond to pointcuts that are defined based on explicit enumeration of a set of method signatures. *AspectJ* also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. These two different ways of designate the pointcuts are called *name-based crosscutting* and *property-based crosscutting* respectively.

The simplest way to define property-based crosscutting is by means of the use of wildcards in certain fields of the method signature. For example:

```
call(void FigureElement.get*(..))
```

identifies calls to any method defined on FigureElement, for which the name begins with "get", specifically the factory methods getX and getY; and

```
call(public * FigureElement.* (..))
```

identifies calls to any public method defined in FigureElement.

- **User-defined pointcut designators**

  Programmers can define their own pointcuts by means of the *pointcut* designator. The pointcut designators can identify join points from different classes and thus crosscut classes.

  The declaration:

```
pointcut move(): call(void FigureElement.setXY(int,int)) ||
                 call(void Point.setX(int))              ||
                 call(void Point.setY(int))              ||
                 call(void Line.setP1(Point))            ||
                 call(void Line.setP2(Point));
```

  declares a new pointcut designator called move that identifies any call to methods that move figure elements. User-defined pointcut designators can be used whenever a pointcut designator can appear. For instance:

- `PointcutId(TypePattern or Id, ...)`

  Picks out all join points that are picked out by the user-defined pointcut designator named by PointcutId.

- `! Pointcut`

  Picks out all join points that are not picked out by the pointcut.

- `Pointcut0 && Pointcut1`

  Picks out all join points that are picked out by both of the pointcuts.

- `Pointcut0 || Pointcut1`

  Picks out all join points that are picked out by either of the pointcuts.

3.4.2.3 Advice

Advice is a method-like mechanism used to declare that certain code should execute at each of the join points in a pointcut. AspectJ supports three kinds of advice: *before*, *after*, and *around* advice. Additionally there are two special cases for the after advice, *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return from the execution of a join point.

Before and after advices are additive with respect to the normal computation at the join point. Before advice runs when a join point is reached and before the computation proceeds, i.e. it runs when computation reaches the method call and before the actual method starts running. After advice runs after the computation under the join point finishes, i.e. after the method body has run, and just before control is returned to the caller. Around advice runs when the join point is reached. It preempts the normal computation at the join point and it has explicit control over whether it is allowed to continue running.

Advice declarations define advice (before, after or around) by associating a code body with a pointcut.

For example:

```
after(): move(){
 System.out.println("A figure element has moved");
}
```

defines that the string "A figure element has moved" is written in the standard output whenever a figure element finishes handling a call to the move() method.

Each time a join point is reached, all advices are examined to see whether any apply at that join point. The ones that apply are selected and ordered according to their specificity and executed in the following order:

1 around advices first, in order of specificity. Proceed calls the next most specific piece of around advice.

2 all before advice, most-specific first.

3 the computation identified by the join point proceeds.

4 after returning and after throwing advices.

5 after advice, least-specific first.

6 the return value from step 3 is returned to proceed with step 1 and the innermost piece of around advice continues running.

3.4.2.4 Aspect

An *aspect* encapsulates the implementation of a concern that cuts across the functionality of the system. The structure of a crosscutting concern written as an aspect is explicit and easy to reason about. Aspects are defined by aspect declarations, being able to include pointcut declarations, advice declarations as well as any kind of declaration allowed in class declarations.

```
aspect FigureCallsTrace{

 pointcut figurecalls():
  !instanceof(FigureElement) && calls(void FigureElement.*(*));
 before(): figurecalls() {
  System.out.println("Before the method is executed");
 }

 after(): figurecalls() {
  System.out.println("After the method was executed");
 }
}
```

The code shown before illustrates an aspect, which traces all the calls to the methods defined in the class FigureElement that take one argument as input and have no return value. It prints a message before and after those methods are called.

3.4.3 Aspect Extension

An aspect, abstract or concrete, may extend a class and may implement a set of interfaces. Extending a class does not provide the ability to instantiate the aspect with a new expression; the aspect may still only define a null constructor.

Aspects may extend other abstract aspects, in which case not only the fields and methods are inherited but so are pointcuts and advices.

It is important to notice that an aspect can define concrete or abstract pointcuts with or without concrete or abstract advice. The abstract ones can be specialized in the specializations of the aspect (subclasses). Extending aspects can define concrete pointcuts, add advice to inherited pointcuts and override normal inherited methods advice.

3.4.4 Aspect instantiation

Aspect instances are automatically created to cut across programs.

Because advice only runs in the context of an aspect instance, aspect instantiation indirectly controls when advice runs.

- *Singleton aspects*

  By default, or by explicitly using the modifier *issingleton*, an aspect has exactly one instance that cuts across the entire program. That instance is available at any time during program execution with the static method *aspectOf()* defined on the aspect.

  ```
  aspect Id { ... }
  aspect Id issingleton { ... }
  ```

- *Per-object aspects*

  ```
  aspect Id perthis(Pointcut) { ... }
  ```

  ```
  aspect Id pertarget(Pointcut) { ... }
  ```

  If an aspect named A is defined *perthis(Pointcut)*, then one instance of the aspect type A is created for every object that is the executing object (i.e., "this") at any of the join points picked out by Pointcut. The advice defined in A may then run at any join point where the currently executing object has been associated with an instance of A.

  Similarly, if an aspect A is defined *pertarget(Pointcut)*, then one instance of the aspect type A is created for every object that is the target object of the join points picked out by Pointcut. The advice defined in A may then run at any join point where the target object has been associated with an instance of A.

  In both cases, the static method call *A.aspectOf(Object)* can be used to get the aspect instance (of type A) registered with the object.

- Per-control-flow aspects

  ```
  aspect Id percflow(Pointcut) { ... }
  ```

  ```
  aspect Id percflowbelow(Pointcut) { ... }
  ```

  If an aspect A is defined *percflow(Pointcut)* or *percflowbelow(Pointcut)*, then one instance of the aspect type A is created for each flow of control of the join points picked out by Pointcut, either as the flow of control is entered, or below the flow of control, respectively. The advice defined in A may run at any join point in or under that control flow. During each such flow of control, the static method *A.aspectOf()* will return an instance of type A.

3.4.5 Aspect Domination

To control precedence between aspects that do not exist in an extends relationship, *AspectJ* provides the feature *dominates*. An aspect can declare that the advices defined in it should dominate the advices defined in some other aspect. The declaration:

```
aspect Id dominates TypePattern { ... }
```

states that the advice declared in the aspect where this declaration is defined has more precedence and will run before the advice in the aspects from TypePattern.

Chapter 4

Case Study

The case study we will use in this dissertation is an example of an electronic store application that provides support for personalization issues.

In the following sections we describe the characteristics of this kind of applications. We also justify the choice of this kind of application as a representative case to illustrate the problems and needs in the externalization, configuration and application of business rules.

4.1 The e-store: core entities

The example application models an electronic store that basically sells products to customers. The customer can chose the products he/she wants to buy and add them to the shopping cart, indicating the number of units for each product. When the customer wants to confirm the purchase an order is generated and the check-out process starts. Normally, the check-out process consists of steps through which the order generated with the items selected by the customer must be processed. These steps include information about the paying mechanism, shipping address, delivery options and wrapping details. Each customer of the e-store has an account containing the buying history of that customer. In Figure 4.1 we present an object model representing the key features of the e-store example.

In this example application we will consider that product prices can be personalized. Thus, different business rules that specify how the prices should be personalized have to be provided.

4.2 Why is this case representative?

Nowadays, as it is recognized in current issues of ACM communications [Cra] more and more emphasis is put on web-based applications. The main

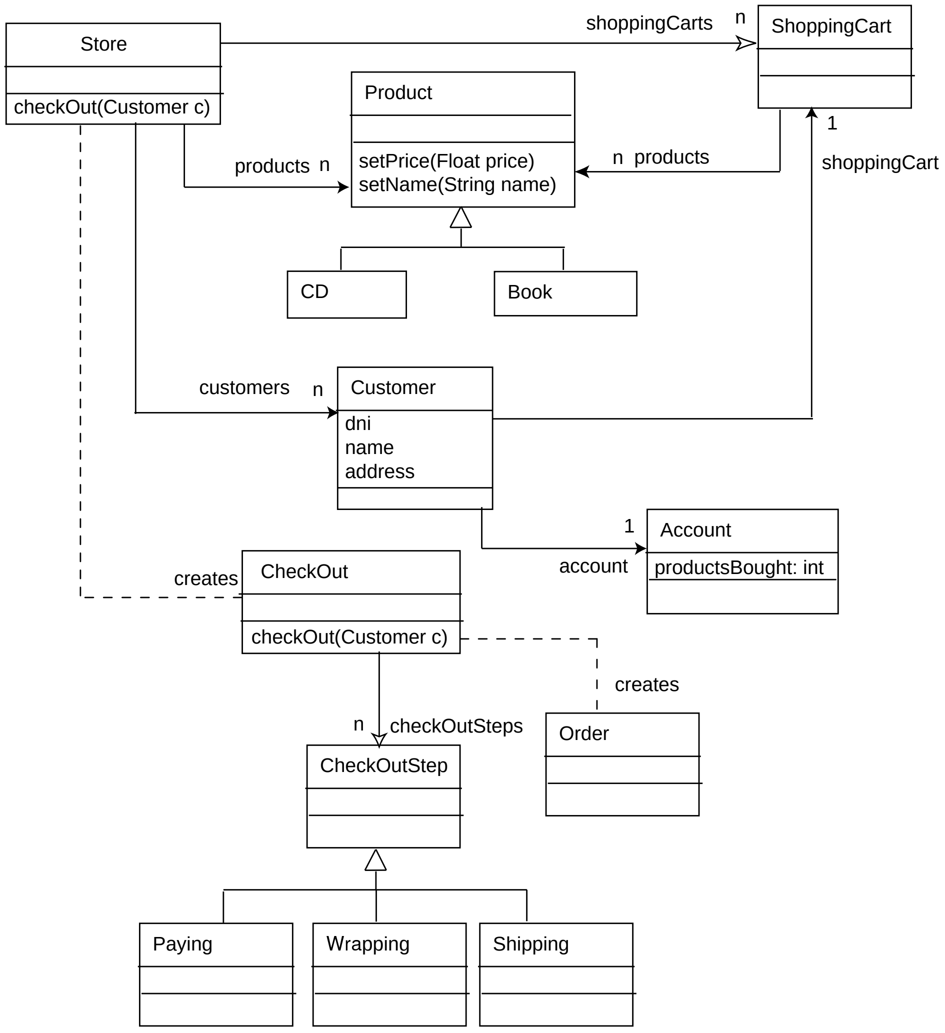


Figure 4.1: Conceptual model for an e-store application

problem today regarding web systems is the impossibility to satisfy all the heterogeneous needs of the many different users. Examples of this problem are web courses that present the same static learning material to students with different backgrounds about the subject, web e-stores that offer the same featured items to customers that have different preferences and needs, web museums that provides the same information to different visitors with very different goals and interests. What is missing in most of current systems is the ability to adapt themselves to the particular needs, interests, goals, preferences, wishes of specific users or groups of users.

Generally, business rules capture the personalization issues representing the information about how the adaptation should take place. It is important to remark that due to the evolvable nature of e-commerce applications and business needs, these rules evolve rapidly to reflect personalization requirements. This motivates the need to have them decoupled from the rest of the application.

Thus, we believe that this case is a good example of an application that needs to rapidly evolve to reflect changes in the business requirements.

Moreover, several approaches in the state of the art on business rules [IBMb] [RDR$^+$] [CRFS01] also consider e-commerce applications as representative cases to illustrate the problems and needs when dealing with business rules.

Thus, we consider this example as a representative situation to illustrate the problems concerning business rules. In this dissertation we will use this application as the case study to base the examples on.

Chapter 5

Analysis of Business Rules

5.1 Different dimensions for the classification of Business Rules

Business Rules can be classified according different criteria.

In the following sections different issues to take into account in the classification of rules are presented.

5.1.1 Regarding the specification of the Business Rule

Conceptually speaking we can mainly classify the business rules in two different groups: *constraints* and *"if then"* business rules. The first ones state certain conditions about some properties or relationships among business objects that should be satisfied whereas the second ones specify certain actions to be performed when certain conditions are satisfied.

An example of a constraint can be *"the age of a customer must be greater than 18"*. A constraint is an assertion that must always be satisfied in the system. They can generally be expressed using some variant of logic like first order or predicate logic.

The "if then" business rules specify statements of the form *"IF condition THEN action"*. The antecedent generally resembles a constraint business rule that represents the condition of the rule, i.e., the circumstances under which the action specified as the consequent of the rule must be performed. An example of "if then" business rule is: *"if the customer buys the same product more than 10 times, then apply 10% discount"*.

Basically we can identify different kinds of conditions. Also the actions to perform when the rule is applicable, i.e., when the condition is satisfied, can be classified in different cases. We propose the following classification for actions and conditions of business rules:

5.1.1.1 Conditions

We can mention two types of conditions:

- *conditions that refer to structural situations:*

  These conditions are defined in the context of some business objects, referring to a property that should be satisfied. For instance, a condition can check if *"the age of the customers is greater than 18"*. This condition refers to a structural situation because it checks that the attribute age, part of the structure of a customer object, contains some value.

  A special case for this kind of conditions occurs when they consider checks on the types of objects, for instance, a condition that checks if *"the customer is a gold member"*, where a gold member is an instance of a class GoldMember.

- *conditions that refer to contextual situations:*

  These conditions refer to situations that involve checks on dynamic information. Generally this dynamic information defines contextual paths of execution. As an example we can think of a condition that checks if *"the customer has performed some actions"* before certain other operations can be allowed. For instance, the shop can require the customer to log in the system before he/she can perform a purchase. Suppose the information that the customer has logged in the shop is not kept as part of the structure of the customer, as an attribute. Thus, to check this condition we have to explore the dynamic path of execution of the customer.

5.1.1.2 Actions

We can classify the actions for the *"if then"* business rules as follows:

- *additional functionality:*

  This additional functionality can either replace some existing behavior or can be added as extra functionality after or before its occurrence. For instance, when dealing with price personalization, we can imagine a business rule that specifies as the action a new way to calculate the price that "overrides" the original calculation.

  But in other cases we just need to specify some extra behavior to be performed before or after certain core application processes occur. For instance, the rule *"if a customer has bought more than 10 products, then classify him/her as a frequent customer"*. The classification of the customer must be done after the customer buys the products. Thus, this business rule specifies as its action some behavior that must be executed after the confirmation of the purchase take place.

- *structural additions:*

  Business rules can specify as their actions changes in the original structure of the system, through the definition of new attributes and methods that extend certain types defined in the original application.

  For instance, the shop can consider a business rule that specifies: *"if the shop has sold more than 1000 products, then the customers can start accumulating points in the shop"*. This means that the customers can start gaining points for each purchase they perform to get, using the accumulated points, some extra benefits in the shop. To achieve this new functionality, the class Customer should define a new behavior to accumulate points. Thus, the method accumulatePoints(int points) should be added in the Customer class.

- *invocation of existing behavior:*

  The action of a business rule can consist of the invocation of a behavior already defined in the core application. In this case, the rules need to interact with the business objects that define the needed behavior.

  For example, suppose the products define a behavior to increase their price in some percentage. Thus, consider the business rule: *"if a product has been bought more than 100 times, then increase the price of the product in 10%"*. In this business rule, the action implies the invocation of the existing method increasePrice(int percentage) defined in the Product class.

- *activation and deactivation of business rules:*

  Business rules can also take decisions about other business rules. For instance, a business rule can select or deselect other business rules as part of its action, i.e, it can activate or deactivate other business rules to be considered or not when their application time arrives.

  An example of this kind of rules is: *"if the customer spent more than 100 euros in a purchase, then he/she gets a discount of 10% in the next purchase"*. When the condition of this rule is satisfied, a new business rule stating *"if the same customer buys a product, then apply 10% discount"*, is activated to be considered when the prices are personalized.

The presented classification of business rules into constraints and "if then" rules corresponds to the natural way to think about business rules.

It is important to notice that even if both categories are conceptually different, they motivate the same issues to be taken into account for their application and management (see 5.2). It is not the focus of this dissertation to specially focus on constraints. Some research has been done [Str00] [Ver01] on how constraints can be mapped to the implementation and how their automatic validation can be ensured. In the rest of the dissertation we will concentrate on "if then" business rules.

5.1.2 Regarding scope

Some rules are specified in terms of features that make reference to specific objects: *"the CD called Let's talk about love by Celine Dion has a special discount"* is stating something about a specific product. These rules affect specific business objects.

Rules can also be specified in terms of one or many common properties of a group of business objects. For instance, *"all the clients that pay with visa card have a 10% discount"* specifies a property that involves a set of customers, the ones that have visa card.

Finally, there are rules that are "environmental", i.e., they do not depend on characteristics of specific business objects like products or clients but they depend on general situations or conditions that are global to the system. For instance *"the anniversary day of the shop, all the products have a special discount of 5%"* specifies a rule that needs to check for the current date that is part of the environmental information.

Notice that some rules can mix different scopes of information needed. For instance, *"if today is San Valentine's day, then all the CDs of love have a 10% discount"* is a rule that uses the environmental information about the date and also defines a property that affects a set of objects,i.e., the CDs with theme love.

Notice that, if we are using traditional object-oriented approaches for the management of business rules, the previous classification of the rules according to their scope motivates the need for defining the ownership of the rules. As it was motivated in the introduction of this dissertation, when dealing with rules at different levels of scope we need to decide where to place the rules to avoid redundancy. A rule that specifies some business decision involving products and customers at the same time, should be placed in the Product class, in the Customer class? Which are the links between the rules and the core business objects involved in their specification? As it is presented in the following chapter, the approach we present in this dissertation completely factors out the business rules from the rest of the application, which makes it unnecessary to define who owns the business rules. In our approach rules do not need to be placed in specific business objects because they are defined separately from the core application. They are connected with the rest of the application code in the places where they have to be applied.

5.1.3 Regarding life time

The following situation motivates the idea of having business rules that have other business rules as actions. Suppose this example business rule:

"if a customer buys a CD in the category of classical music, then he/she

gets a discount of 15% on the next classical music CD".

In this rule, the condition defines that *"the customer buys a classical music cd"*, and the consequent can be seen as another business rule that says that *"if the same customer buys another classical music CD, he/she gets the discount of 15%"*. We call the business rule that is specified as the consequent *dependent business rule*, because it is applied only if the outer rule was applied and its condition satisfied. Thus, its application *depends* on the result of the application of its enclosing rule.

Therefore, we can consider that business rules have *life time*. Some rules are always active whereas other rules become active through the execution of other business rules. Moreover, the dependent business rule can remain active after its occurrence or it can become inactive under certain conditions as well. All these situations themselves are controlled by the specification of other business rules.

We can think of many situations to determine the life time of the business rules that are activated by other business rules. For instance, if the dependent business rule is activated, we could consider the business rule that activated it inactive until the dependent business rule is applied once, or both can remain active at the same time, or any other situation as well.

The fact that rules can define other rules as their actions or consequents implies different cases for the application time of the final action specified in the business rules. On one hand, there are rules whose actions must take place at exactly the same time the rule is applied. For instance, the rule:

"if a client buys more than 10 books at the same time, he/she gets a 20% discount"

must be applied when the price is calculated and the action, i.e., the discount, should be applied at that moment as well. On the other hand, there are other rules whose effect or action must take place later, when another situation occurs. In the example rule presented before where a discount is specified for the second time a CD of classical music is bought by the same customer, the discount should be applied later on, when a new situation occurs, that is, when the customer buys another CD of classical music.

5.2 Requirements and Features needed for Business Rules

5.2.1 Business Rules as first class entities

Rules need to be explicitly represented in the software to facilitate their management. Moreover, it is desirable to have business rules decoupled from the rest of the application. Decoupling and treating them as first class citizens would allow us to keep the application software more adaptable.

Ideally it should be possible to plug, remove, change and refine the rules without affecting the core application, both statically and dynamically.

5.2.2 Dynamic join point model

As it was mentioned in the introduction, we want to model business rules as decoupled as possible from the core business objects. Nevertheless, after having decomposed the core application on one side and the business rule logic on the other side, we want to compose or weave the two parts together in order to obtain an operational software application that behaves in the desired way. We need a dynamic join point model where we can capture very well-defined points in the execution of a program to specify the exact moment a business rule should be applied. In this way, we would be able to insert the business rules in specific dynamic places in the core application.

The power of a dynamic join point model is needed for the specification of the following features:

5.2.2.1 Identification of the application time

The business rules must be applied when certain situations occur. Generally, these situations correspond to well-defined points in the core business logic.

For instance, we would like to capture the point in the execution of the program when the price of a product is being calculated to apply at that time certain business rules that define different policies to personalize the price. Or we might need to identify the moment when a customer navigates the recommended items of the shop to trigger the application of a rule that personalizes the recommended products offered to that specific customer.

5.2.2.2 Context sensitive application

The business rules might be applied within a certain context. This means that when the moment for the application of a business rule arrives, we will take it into account and apply it only if we are in the situation described by the context.

As an example let's consider the following business rule:

"If the customer is buying more than 10 products, then apply 10% of discount".

Suppose it is desired to apply the discount only for the products that are being considered in the checkout process and not for any arbitrary price calculation. Thus, the *context* for the application of this business rule is defined by the check out process. If the price of the same products is calculated outside the context of the checkout process, for instance when the customer is browsing the products of the store, looking the catalog of the shop to decide what to buy, the discount should not be performed and thus, the rules not applied.

5.2.3 Information needed for the application

5.2.3.1 Environmental information vs. specific business objects

When applying a business rule we might need to use different kinds of information. A business rule might need to use environmental information, always available in the system but it might also need to interact with different business objects either to check the condition or to perform the action. Thus, it is important to have the necessary mechanisms to identify the objects that contain the information needed and make them available for the rule at the moment it has to be applied.

For instance, in the rule that states *"if today is the birthday of the customer, then apply 5% of discount"*, both the customer and product business objects and also the environmental information about the current date are needed for its application.

5.2.3.2 Non-invasive extension of unanticipated information

When dealing with business rules, two kinds of information can be identified: the one that defines the core basis and that is intended to be kept by business objects defined in the main application and the information that is only needed by the business rule logic.

As it is recognized in [CRFS01], business rules might need for their application extra information that was not previously identified when the core application was developed. Thus, it is important to keep track of that information, explicitly represent it and make it accessible for the business rules when they are applied. This information should be clearly separated from business objects and should grow in a transparent way. It is important to notice that most of this information is generated by core business operations, e.g., buying a product, adding a product to the shopping cart. Thus, a mechanism to intercept these business processes and capture, maintain and evolve the required information is needed, possibly through the addition of new code, extending existing business objects or identifying and introducing new abstractions. Dynamic crosscutting would be very useful in this sense to express more accurately these points of interception. Static crosscutting is needed as well to extend existent types with the new structure required to keep the unanticipated information.

Suppose the shop would like to offer its customers some price reductions depending on their buying history. In this case, business rules would need to specify some decisions based on the amount of products previously bought by the customer. Suppose the amount of products bought by a customer corresponds to information not anticipated from the beginning and needed only for the business rules. Thus, it should be kept apart from the core application. This historical information represents the information needed by the business rule logic and should be added non-invasively into the application.

5.2.4 Dealing with interferences between rules

First of all we need to identify and decide which kind of conflicts are important to be considered in the context of this dissertation. Basically we will focus on the conflicts that appear at the business level, when different business decisions to be applied at a certain moment can conflict or interfere with each other.

For instance, suppose that during certain periods of time, it might be the case that the company decides to consider specific sales for certain products and to apply as well certain global policies in the calculus of the price.

At this point, when combining several rules, conflict problems can occur. As soon as there is more than one applicable rule that describes what to do under certain circumstances, some decision to solve the conflicts must be taken to decide effectively which action to take.

Let's consider the following rules:

> AmountBR: *"if a client buys more than 10 books, then he/she gets a 20% discount."*
>
> ChristmasBR: *"if today is Christmas, all the products in the shop have a 10% discount"*

What will happen if a client buys more than 10 books on the Christmas day? Should he have a 10% and then 20% discount? Or will only one of them be applied? This example represents a conflicting situation where a decision should be taken to solve the interference. Thus an appropriate design structure for handling conflicts must be part of the global design of business rules. Moreover, it should be possible to change the policy that decides how to solve the conflict in a non-invasive way. Maybe today business considerations imply that it is better to consider the application of the AmountBR in case both rules are applicable, but maybe tomorrow business decisions change and the application of the other rule or even of both is more appropriate.

There are different desired issues to take into account regarding the interference or conflicts between rules:

5.2.4.1 Composition issues decoupled from business rules

We want to be able to decouple the details of how to combine the business rules from the specification of the business rule itself.

As we saw in section 5.1.1, a business rule can specify only a constraint to be satisfied or a certain action to perform when a certain condition is satisfied. All the other considerations about how to compose business rules should be considered separately to facilitate their management.

For instance, suppose that we have several business rules each customizing the price in a certain way. Depending on the needs we would like to combine them by means of concatenating the discounts, or just choosing the biggest discount or applying other criteria. The way to combine them should be specified independently from the business rules that are being combined. However, the combination of rules should also be specified as part of the business logic.

5.2.4.2 Dynamic selection of combination strategies

Ideally it should be possible to combine the rules adopting different strategies and selecting between them depending on dynamic situations.

As an example suppose that different ways to combine rules that personalize the price of a product are provided and that the selection is made at run time according to some dynamic information. For instance, if the customer that is currently buying a product is a frequent customer of the shop, then a strategy that chooses the biggest discount among the applicable ones specified by different rules is preferable; otherwise, if the customer is not a frequent one, a strategy where only the lowest discount would be applied is chosen.

Or for instance the shop can decide to apply a certain combination of policies depending on how much money the customer has spent in the shop.

To conclude with, notice that regarding the combination of business rules, two different levels can be defined. On one hand the basic mechanism consists of specifying which actions to take when two business rules are applied and both conditions are satisfied. On the other hand, and more sophisticated approach is the one where the mechanism is chosen depending on dynamic conditions. Notice as well that these conditions used for the decision of which combination strategy to select, do not necessarily have relation with the conditions of the business rules themselves.

5.3 Relationships between business rules

It could be the case that the application of one business rule can have impact on the validity of other business rules. For instance, *"if the customer made purchases for more than X money, then the customer is classified as a privileged one"*, meaning that he/she gets more benefits. The fact that now the customer is considered a privileged one might be used later, in the application of other business rules.

Other situations where the relations between the rules should be analyzed is the triggering of rules. Suppose that the application of one business rule can trigger the application of other business rules. For instance, suppose the rule that states *"if the price is smaller or equal to 90, then apply a 20% discount"* and another one saying that *"if it is Christmas, then apply a 10% discount"*. Suppose that only the condition of the second rule is satisfied for a given product and that the price of the product is 100. Then the second rule is applied obtaining a new price of 90; but immediately after this application, the condition of the first rule is satisfied as well. The question is, what to do in this context? The conflict resolution mechanism must specify the logic of mixing the business rules considering the impact of the application of some rules on the applicability of other rules. The same way the application of a business rule can make valid another business rules, the opposite can occur: after applying one rule, others can become not applicable anymore. All these issues have to be analyzed when dealing with rules that can influence each other.

After having analyzed and described all the necessary ingredients required for the management of business rules as well as the different forms of conditions and actions that can be considered, we now focus on studying which mechanisms are the most appropriate to deal with the identified issues.

In the following chapter we present generic solutions to deal with most of the identified problems and requirements identified and analyzed in this chapter.

Chapter 6

AOP for Business Rules

Aspect-Oriented Programming provides a mechanism to explicitly capture crosscutting structure, thus being able to express crosscutting concerns in a modular way. For this reason we decided to explore its suitability for the decoupling and management of business rules. For practical considerations we have chosen *AspectJ* as a concrete example of an aspect-oriented programming language to base the experiments on.

As a first attempt it seems natural to model the core application with objects and the business logic with aspects. However, we found that it is only the connectivity between the core application and the business logic that requires the weaving power of *AspectJ* whereas the conditions and most actions of the business rules can be expressed using objects.

Moreover it is desirable to keep the business logic as reusable as possible. For instance, we would like to develop a set of business rules that personalize the price of products in an e-store and use them in many different e-store applications. Because of that and due to the complexity of business logic we think it is better to use all the power of object-oriented paradigm to model business logic. Thus, in our approach both the core application and the business rules are expressed as objects and we use aspects developed in *AspectJ* for connecting the rules with the core application.

In the following sections we will describe some design decisions to consider for each of the issues involved in the management of business rules (see 5) and a detailed description of the necessary ingredients for the decoupling, configuration and application of business rules. A generic solution is provided for each of the considerations and illustrated with examples from the case study application.

These generic solutions represent an important contribution of this thesis. They can be used as methodological guidelines for the decoupling of business rules and their weaving in a software application.

6.1 Design decisions

In this section we will explain the general design decisions taken for the solutions presented in this chapter.

As we mentioned before, both the core application and the business rules will be expressed in objects. We will follow the idea of the *rule object pattern* [Arsa] to model the business rules. Aspects defined in *AspectJ* will be used for the connectivity and configuration layer. Several aspects must be defined for the different considerations involved as part of the connectivity layer:

First of all, we need to define aspects to make explicit the moment when a rule or set of rules has to be applied. We named this moment *business rules application time* (see 6.2). Three cases can be distinguished: in the simple case, the application time identifies a point in the execution of the core application where the application of the rules needs to be triggered; other more sophisticated cases occur when we need to restrict the application of a rule or set of rules to specific contexts. These contexts can be defined directly (situation where the application time occurs within the scope of certain method) or indirectly (when the application time occurs within a specific control flow).

Secondly, we also need to determine the *kind of information needed* by the rules for their application and define the needed aspects to make it accessible for the rules to be applied (see 6.3). Four cases can be encountered: the rules can need information that is available and reachable at the moment when the rules are applied; or it can occur that the rules need specific business objects that are not reachable at the moment of their application; the rules can also use environmental information always available in the system; finally, non-anticipated information, i.e., information that was not foreseen nor represented in the core application can be needed as well.

After defining the two previous issues, we need to put all the pieces together in another aspect that triggers the *application of a rule or set of related rules* when the corresponding application time arrives (see 6.4). Different ways to define and instantiate this aspect are presented depending on whether we need to apply only one rule or a set of rules at the same time, whether we need to provide support to combine rules and to solve the conflicts that can arise between them and also whether the way to solve the combination of rules must depend on conditions based on dynamic information.

Thus, any rule that needs to be applied will imply the definition of several aspects for the three main issues just described before, i.e., identification of application time, provision of information needed for the application and triggering of the rules at a certain point in time. All these aspects combined together form the *connectivity layer* in between the *core application* and the *business rules* themselves.

In Figure 6.1 we present an illustrative diagram of the different layers identified in the decoupling of business rules and their connectivity with the core application.

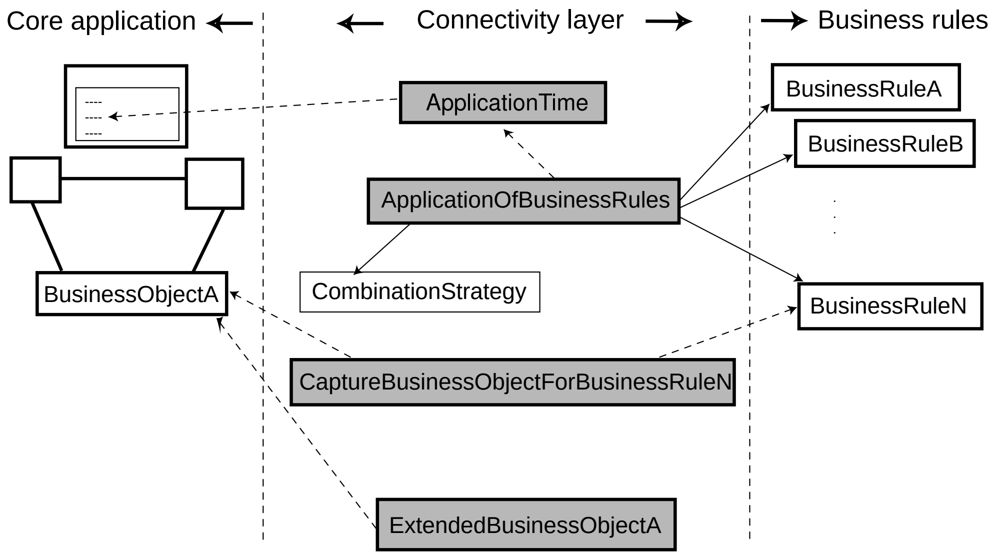


Figure 6.1: Different layers for the decoupling of rules and their connectivity

In the rest of the chapter we will present generic solutions for each of these issues considering the different cases that can be encountered for each of them. At the end of this chapter we also describe some special cases of business rules. In particular we analyze rules that perform checks on types of objects 6.5.1 and rules that take decisions about other rules 6.5.2.

6.2 Identification of the application time

Considering the moment when a business rule should be applied we can distinguish two cases:

6.2.1 Basic application time

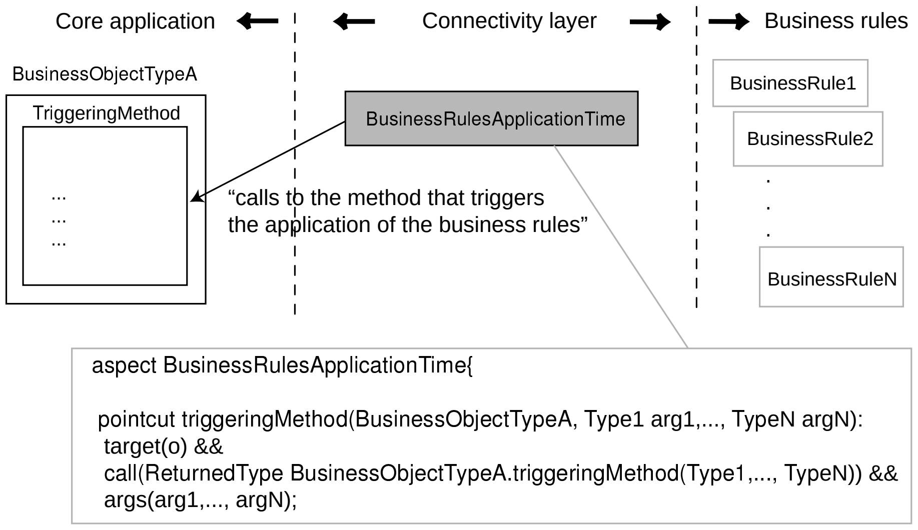
Each business rule must be applied when some situations occur. Generally, these situations correspond to well-defined points in the core application. Typically business rules need to be applied when some behavior is invoked in the core application or when some operations involving the structure of some business objects are performed, by accessing or setting structural fields.

We need to be able to precisely identify and explicit these points in the execution of the program to trigger the application of the rules.

Solution

We can precisely identify the points in the core business application when the business rules have to be applied by means of the definition of a pointcut in an aspect. As we saw in 3.4.2.2, a pointcut picks out join points, as well as data from the execution context of those join points. Thus, we can apply the business rules when any of the situations that can be captured in pointcuts occurs, i.e., at any of the nodes in the runtime object call graph described in 3.4.2.2.

The aspect that contains the definition of the pointcut that identifies the application time corresponding to a method invocation is shown in Figure 6.2. Notice that the grey boxes correspond to aspects whereas the white ones correspond to classes.



```
aspect BusinessRulesApplicationTime{

 pointcut triggeringMethod(BusinessObjectTypeA, Type1 arg1,..., TypeN argN):
  target(o) &&
  call(ReturnedType BusinessObjectTypeA.triggeringMethod(Type1,..., TypeN)) &&
  args(arg1,..., argN);
```

Figure 6.2: Application time of business rules

The business objects that form part of the execution context of the captured join points in the core application might be exposed in the pointcut, as its formal parameters. For instance, suppose the core business process that identifies the application time corresponds to a method invocation. In this case, the pointcut can expose the business objects that correspond to the target object and arguments of the method (as shown in Figure 6.2). What should be actually exposed depends on which of those business objects are needed for the application of the business rules.

Example

For instance, consider that the shop specifies business rules for the personalization of product prices. Thus, these business rules should be applied *when the price of the product is being calculated.*

We need to define an aspect that declares a pointcut for picking up the moment when the method for the calculation of the price is being called.

```
aspect WhenToPersonalizePrice{

 pointcut priceCalculation(Product p):
  target(p) && call(Float Product.getPrice());
}
```

6.2.2 Contextual application time

In the previous section we have identified the dynamic points in the core business application when the rules have to be applied. However, sometimes only the identification of those points might be not enough to accurately express the desired moment for the application.

For some business rules it might be needed to restrict their application to certain contexts, i.e., certain situations that depend on dynamic information. Two cases can be identified for the situations described by the context:

6.2.2.1 Direct contextual application time

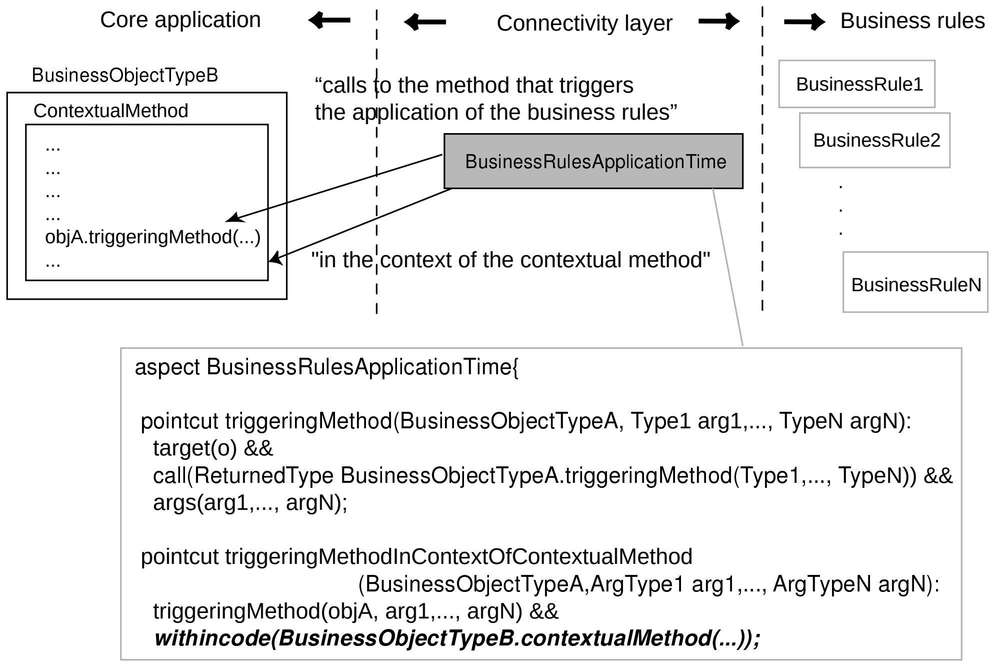
This case identifies the situation where the join points picked up by the pointcut that describes the application time, occur within the scope of certain method (the contextual method). This means that the invocation of the method involved in the application time is performed within the code of the contextual method.

Solution

We can specify the situations that describe the context for the application by means of extending the aspect for the application time with another pointcut. This new pointcut restricts the join points captured by the application time pointcut to those occurrences within the desired context. Thus, this pointcut should perform the logical *and* operation between the application time pointcut and the joint point that describes the contextual situation.

Using the feature *withincode* provided by *AspectJ* we can restrict the occurrence of the pointcut for the application time controlling that it occurs within the code of another method, the contextual one. Figure 6.3 shows a scheme of this solution.

Example


Core application
Connectivity layer
Business rules
BusinessObjectTypeB
ContextualMethod
...
...
...
...
objA.triggeringMethod(...)
...
“calls to the method that triggers the application of the business rules”
BusinessRulesApplicationTime
"in the context of the contextual method"
BusinessRule1
BusinessRule2
BusinessRuleN

```
aspect BusinessRulesApplicationTime{

 pointcut triggeringMethod(BusinessObjectTypeA, Type1 arg1,..., TypeN argN):
   target(o) &&
   call(ReturnedType BusinessObjectTypeA.triggeringMethod(Type1,..., TypeN)) &&
   args(arg1,..., argN);

 pointcut triggeringMethodInContextOfContextualMethod
                              (BusinessObjectTypeA,ArgType1 arg1,..., ArgTypeN argN):
   triggeringMethod(objA, arg1,..., argN) &&
   withincode(BusinessObjectTypeB.contextualMethod(...));
```

Figure 6.3: Contextual application time: context expressed using *withincode*

For instance, consider that we want to restrict the application of the price personalization rules and apply them only when the customers are performing the checkout process, confirming the purchase of the products already added to the shopping cart. Thus, the application time is identified by the invocation of the price calculation from within the body of the checkout method:

```
aspect WhenToPersonalizePrice{

 pointcut priceCalculation(Product p):
  target(p) && call(Float Product.getPrice());

 pointcut priceCalculationInCheckOut(Product p):
  withincode(public Float CheckOut.checkOut(Customer))
   && priceCalculation(p);
}
```

This aspect, by means of the pointcut priceCalculationInCheckOut, defines “when” and “in which context” the business rules that personalize the price must be applied.

6.2.2.2 Indirect contextual application time

This case identifies the situation where the join points for the application time, occur after the invocation of a certain method (the contextual method) and within its control flow. It identifies the situation where the join points are originated indirectly from the contextual method.

This case is a general case for the first one, because if one method is invoked within the body of another method, it is invoked within its control flow as well. Nevertheless, to be complete a solution for this case will also be provided.

Solution

For this case we also need to define an extra pointcut that restricts the join points captured by the application time pointcut to those occurrences within the desired context. But in this case we need to control that the pointcut for the application time occurs within the control flow of the execution of the contextual method. *AspectJ* provides the primitive *cflow* that can be used to exactly achieve this (Figure 6.4):

Example

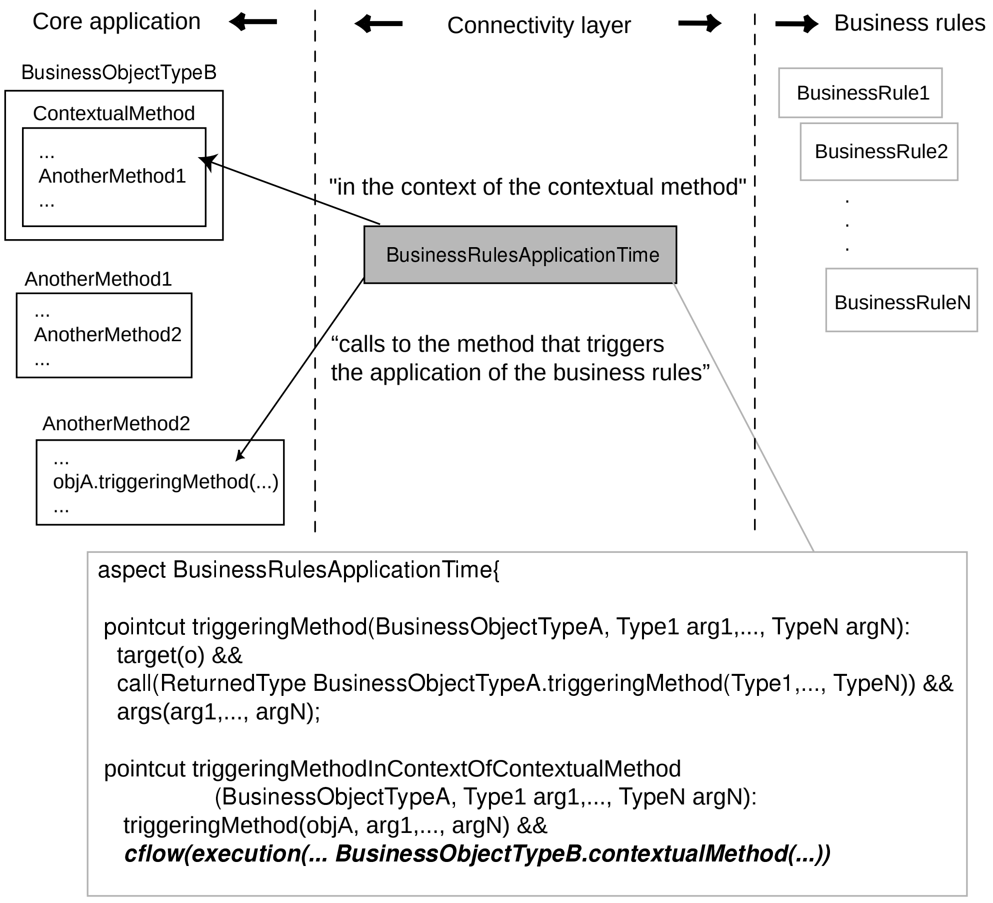
Consider the previous example as in 6.2.2.1. We can also restrict the application of the price personalization rules to the context of the checkout process using the *cflow* primitive:

```
aspect WhenToPersonalizePrice{

 pointcut priceCalculation(Product p):
  target(p) && call(Float Product.getPrice());

 pointcut priceCalculationInCheckOut(Product p):
  cflow(execution(Float CheckOut.checkOut(Customer)))
   && priceCalculation(p);
}
```

Notice that this construct using the primitive *cflow* will also capture the desired situation when the checkout method calls the priceCalculation indirectly. Thus, it is more powerful than the construct using *withincode*.

6.3 Information needed for the application

To determine whether a rule must be applied or not as well as to execute its action, collaborations with certain specific business objects or with any other information available in the system might be needed. In the following



```
aspect BusinessRulesApplicationTime{

 pointcut triggeringMethod(BusinessObjectTypeA, Type1 arg1,..., TypeN argN):
   target(o) &&
   call(ReturnedType BusinessObjectTypeA.triggeringMethod(Type1,..., TypeN)) &&
   args(arg1,..., argN);

 pointcut triggeringMethodInContextOfContextualMethod
                (BusinessObjectTypeA, Type1 arg1,..., TypeN argN):
    triggeringMethod(objA, arg1,..., argN) &&
    cflow(execution(... BusinessObjectTypeB.contextualMethod(...))
```

Figure 6.4: Contextual application time: context expressed using *cflow*

section we identify and classify the possible cases that can occur regarding the information needed for the application of the business rules.

This difference in the information required by the business rules will have to be taken into account at the moment the business rules are applied.

6.3.1 Environmental information

As it was identified in 5, there are business rules that are defined at an environmental scope. These business rules need for their application some system information, i.e., information that is always available in the system.

Solution

As the information needed for these business rules is always reachable in the system, no extra mechanisms are needed to obtain it. The rules can directly access and use this information for their application.

Example

Suppose we consider the rule that states:

"if it is Christmas, a 5% discount is applied over the price of any product.

The rule needs to obtain the current date to check its condition. Thus, as the current date is part of the system information always available, the rule can use that information in its specification.

```
public class ChristmasBR{

 private boolean isChristmas(){
  ...
  //returns true if the current date
  //corresponds to Christmas date
 }

 boolean conditionForCustomizingThePrice(){
  return isChristmas();
 }

 Float personalizePrice(Float price){
  return (new Float(price.doubleValue() -
          price.doubleValue() * 0.05));
 }

 Float apply(Float price){
  if (conditionForCustomizingThePrice())
   return personalizePrice(price);
  else return price;
```

```
   }
  }
```

Notice that these business rules do not keep any state because they do not need to keep any specific information needed for their application. As only one instance of them is needed they can be implemented as singletons [GHJV95].

6.3.2 Business objects available at the application time

Some business rules might need for their application specific business objects. We need to capture those objects to be provided to the rules for their application.

Suppose the business rules need for their application some business objects that are reachable at the moment the rules are applied.

Solution

As we described in 6.2, the pointcut that identifies the application time for the business rules can expose the objects that are part of that execution context. These exposed objects might be needed for the business rules to be applied. Thus, we need to supply them to the rules for their application.

Notice that if we are dealing with a business rule that replaces the original execution of some existing behavior with a new functionality, the original return value (the value that would have been returned without considering the current rule) might be needed for its application. In this case, this value can be captured during the application of the business rule (using the *AspectJ* primitive *proceed*) and passed to the rule at the moment of the application.

To conclude, during the application of a business rule, either the business objects exposed by the application time pointcut as well as the original returned value (for the case of business rules that replace core behavior) can be passed to the business rules to be used in their application.

Example

Consider the following business rule:

"if the product has been offered in the shop for sale for more than one year then a discount of 10% is applied to its price".

This rule must be applied when the price of a product is being calculated, i.e., when the method getPrice() is invoked. As the target of this method is the product that is being bought, the product is available at that application time and thus exposed by the corresponding pointcut to be passed to the rule during its application.

```
public class ProductReleaseDateBR{

 boolean conditionForCustomizingThePrice(Product product){
  //returns true if the date of the product release
  //is more than one year ago;
 }

 Float personalizePrice(Float price){
  return (new Float(price.doubleValue()
          - price.doubleValue() * 0.1));
 }

 Float apply(Product product, Float price){
  if (conditionForCustomizingThePrice(product))
   return personalizePrice(price);
  else return price;
 }
}
```

6.3.3 Business objects not available at application time

Some business rules might need for their application specific business objects. Suppose the business rules need business objects that are not available at their application time, i.e., they need business objects not possible to be captured and exposed during the application time. We need somehow to capture those objects when they are still reachable and pass them to the rules to be used during the application.

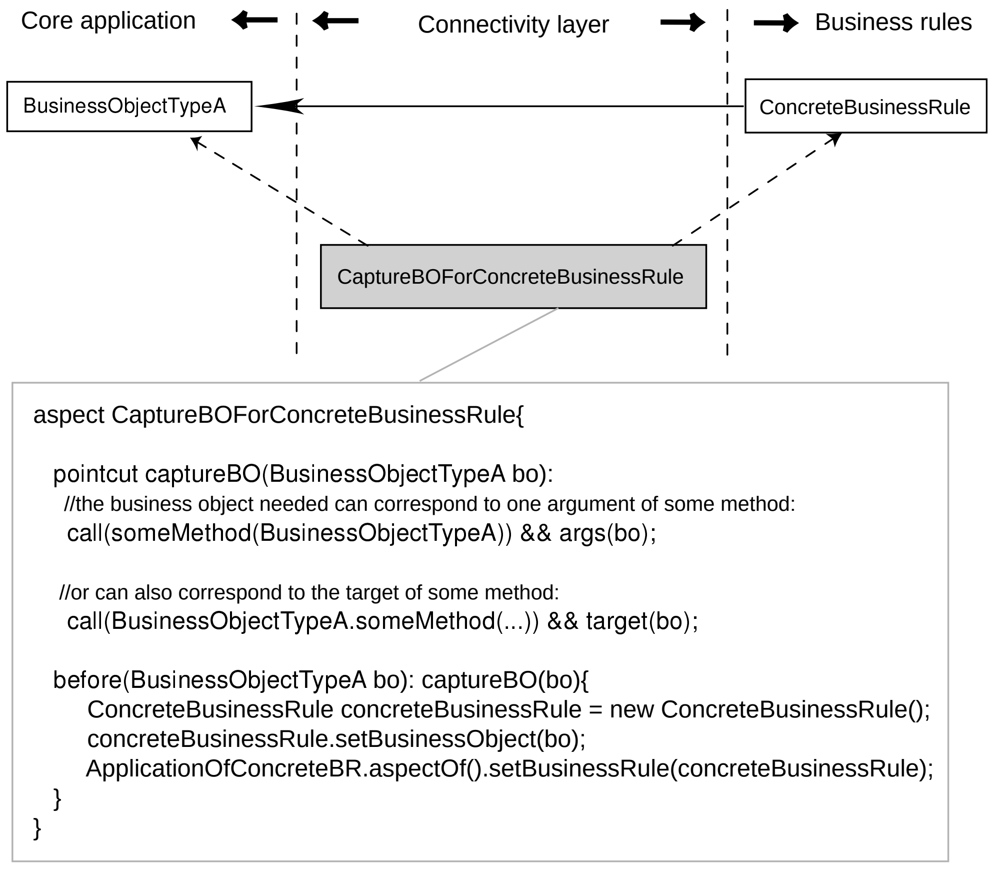
Solution

We need to define aspects to capture those needed business objects at the moment they are still available. This "moment" corresponds to a well-defined point in the execution of the program and thus it can be captured with a pointcut. As part of a piece of after advice on that pointcut, a new instance of the business rule that needs that information is created, the captured business objects are passed to this new instance and the linking with the correspondent instance of the application aspect is performed.

Notice that, contrary to the rules that only use environmental information or objects reachable during the application time, these rules have a *state* because they have to keep the needed objects as instance variables. Thus they cannot be implemented as singletons; several instances of this kind of rules are needed. Figure 6.5 shows an schematic diagram of the solution.

Example

Suppose we consider the AmountBR business rule:



```
aspect CaptureBOForConcreteBusinessRule{

   pointcut captureBO(BusinessObjectTypeA bo):
     //the business object needed can correspond to one argument of some method:
      call(someMethod(BusinessObjectTypeA)) && args(bo);

     //or can also correspond to the target of some method:
      call(BusinessObjectTypeA.someMethod(...)) && target(bo);

   before(BusinessObjectTypeA bo): captureBO(bo){
        ConcreteBusinessRule concreteBusinessRule = new ConcreteBusinessRule();
        concreteBusinessRule.setBusinessObject(bo);
        ApplicationOfConcreteBR.aspectOf().setBusinessRule(concreteBusinessRule);
   }
}
```

Figure 6.5: Capture Business Objects needed for the application of the business rules

"if a customer has bought more than 10 items, he gets a 20% discount on the current purchase".

This rule is stating a condition about the buying history of the customer that is currently performing the purchase. Thus, the customer is needed to be able to apply the rule. But the customer cannot be captured and exposed at the application time because it is not reachable at that pointcut (i.e., at the moment the method getPrice() is invoked). For this reason, we need to define a new pointcut that picks up a join point in the execution when the customer is still available in that context and provide it to the business rule. The business rule needs to keep this business object as part of its state.

A new aspect definition is needed to define this new pointcut:

```
aspect CaptureBOForAmountBR{

 pointcut checkOut(Customer c):
  call(public Float CheckOut.checkOut(Customer)) && args(c);

 before(Customer c): checkOut(c){
  AmountBR amountBR = new AmountBR();
  amountBR.setCustomer(c);
  ApplicationOfBRForPrice.aspectOf().setBusinessRule(amountBR);
 }
}

public class AmountBR{

 Customer customer;

 public void setCustomer(Customer customer){
  this.customer = customer;
 }

 boolean conditionForCustomizingThePrice(){
  return (customer.getAccount().getBuyedProducts() > 10);
 }

 Float personalizePrice(Float price){
  return (new Float(price.doubleValue()
         - price.doubleValue() * 0.2));
 }

 Float apply(Float price){
  if (conditionForCustomizingThePrice())
   return personalizePrice(price);
  else return price;
  }
 }
```

In this code, ApplicationOfBRForPrice refers to the aspect that defines the application of the rule (see below 6.4). For this kind of business rules the aspect for their application cannot be implemented as a singleton, as it is the case for the rules without state (see 6.4.1). A one to one relationship between the instances of the business rule and the instances of the application aspect is needed. Observe that in the advice defined in CaptureBOForAmountBR aspect, ApplicationOfBRForPrice.aspectOf() picks the right aspect instance to be associated with the right business rule instance.

6.3.4 Unanticipated information

There are business rules that need to consider for their specification information that is not represented in the core application. Usually this extra information is not explicitly modelled in the original application due to the fact that it is considered to reflect changes in the business decisions that were not anticipated from the beginning, when the original application was developed. It is not possible to predict all the potential changes in the business considerations that might occur and thus, some support to express these extra considerations is needed. Moreover, a non-invasive way to achieve this extension is desirable.

Solution

The solution consists of the definition of aspects that extend the corresponding business object types with the extra information needed by the business rules. This can be achieved using *Introduction*, *AspectJ*'s static crosscutting mechanism. The extra information can be added as instance variables or methods. To deal with this new information, extra business rules might be defined to determine when and how to manage and update it (see Figure 6.6).

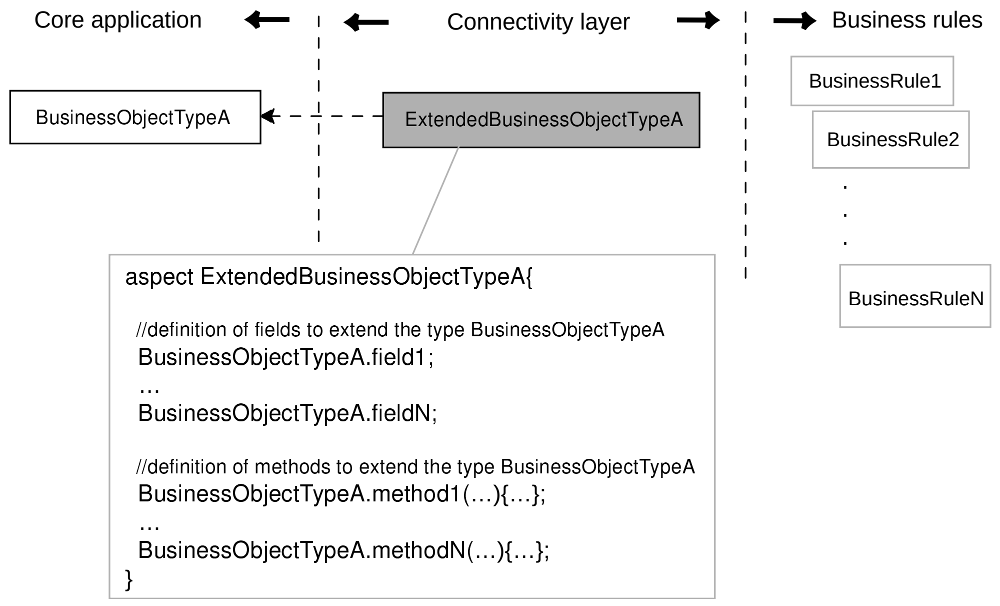
Example

Suppose the shop wants to classify the customers depending on the buying history of the customers. Thus, a new business rule is considered stating that:

"if a customer bought more than 10 items in the shop, then consider him/her as a frequent *customer of the shop"*

This business rule somehow is classifying the customers in roles (frequent or not frequent role) depending on dynamic conditions, i.e., depending on the amount of products bought by the customer.

To achieve this classification it is necessary to extend the Customer type with the addition of a flag that specifies the frequent customer role. This is



```
aspect ExtendedBusinessObjectTypeA{

 //definition of fields to extend the type BusinessObjectTypeA
 BusinessObjectTypeA.field1;
 ...
 BusinessObjectTypeA.fieldN;

 //definition of methods to extend the type BusinessObjectTypeA
 BusinessObjectTypeA.method1(...){...};
 ...
 BusinessObjectTypeA.methodN(...){...};
}
```

Figure 6.6: Unanticipated Information

achievable by means of *Introduction* in AspectJ.

```
aspect ExtendedAccount{

 private boolean Account.frequent = false;

 public boolean Account.isFrequent(){
  return frequent;
 }

 public void Account.becomeFrequent(){
  this.frequent = true;
 }
}
```

In this example, the update of this added field is regulated by the previously mentioned business rule whose implementation is shown as follows:

```
public class FrequentCustomerBR{

 boolean condition(Customer customer){
  return (customer.getAccount().getBoughtProducts() > 10);
 }

 void action(Customer customer){
  customer.getAccount().becomeFrequent();
 }

 void apply(Customer customer) {
  if (condition(customer))
   action(customer);
 }
}
```

Each time a customer performs a checkout to confirm a purchase, the application of this business rule is triggered. Thus, the application time for this rule is defined as follows:

```
aspect FrequentCustomerBRApplicationTime {

 pointcut checkOut(Customer c):
  args(c) && call(Float CheckOut.checkOut(Customer));
}
```

Notice that the customer, object needed for the application of the rule, is available and exposed by the pointcut `checkOut` to be used by the rule. Refer to 6.4.1 to see the definition of the application of this type of rules.

6.4 Application of business rules

The application of a business rule depends on the kind of consequent the business rule defines (see 5.1.1.2). Depending on whether the business rule defines an additional functionality that replaces an existing one, or that is added after or before an existing behavior, or defines an structural change, the application aspect will be different.

Moreover, the kind of business rule according to the scope (environmental information, specific business objects...) also influences the way the application of the rule needs to be defined: the fact that the rule needs to keep additional business objects as part of its state will influence the way the application aspect will be instantiated.

6.4.1 Application of only one business rule

As a first case, suppose we need to apply only one business rule at a given application time.

Solution

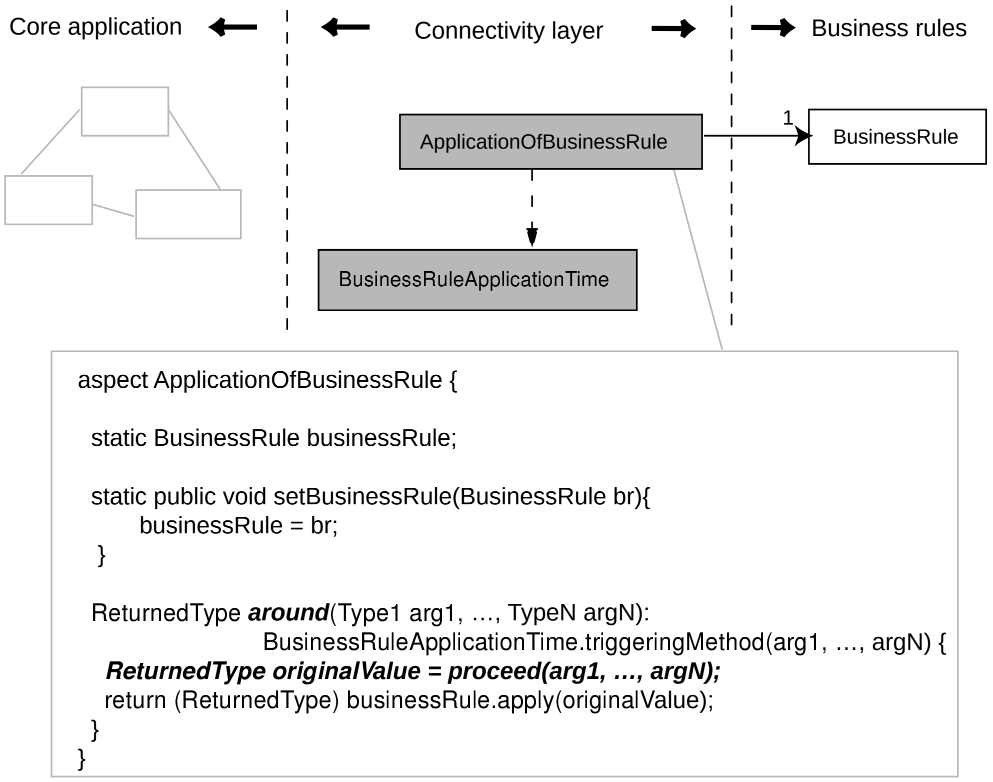
The suggested solution consists of the definition of an aspect for the specification of the business rule application. This aspect declares an advice on the pointcut that specifies the application time. In that piece of advice the application of the business rule is performed. The type of advice, i.e., if it is a before, after or around advice, depends on the kind of business rule is being applied (additional behavior, replacing one, etc.). Moreover, the instantiation of the application aspect will differ depending on the scope of the business rule.

Let's consider all these variations.

Regarding the kind of action specified in the business rule:

- the business rule replaces an existing behavior with the new functionality defined as the action of the rule:

  For this kind of business rule, as we need to preempt the normal execution and replace it with the new functionality, the advice defined in the application aspect must be an *around* advice. The value that the replaced method originally returns is obtained using the primitive *proceed* and passed to the rule to be eventually used in its application (Figure 6.7).



```
aspect ApplicationOfBusinessRule {

  static BusinessRule businessRule;

  static public void setBusinessRule(BusinessRule br){
        businessRule = br;
   }

  ReturnedType around(Type1 arg1, ..., TypeN argN):
                    BusinessRuleApplicationTime.triggeringMethod(arg1, ..., argN) {
    ReturnedType originalValue = proceed(arg1, ..., argN);
    return (ReturnedType) businessRule.apply(originalValue);
  }
}
```

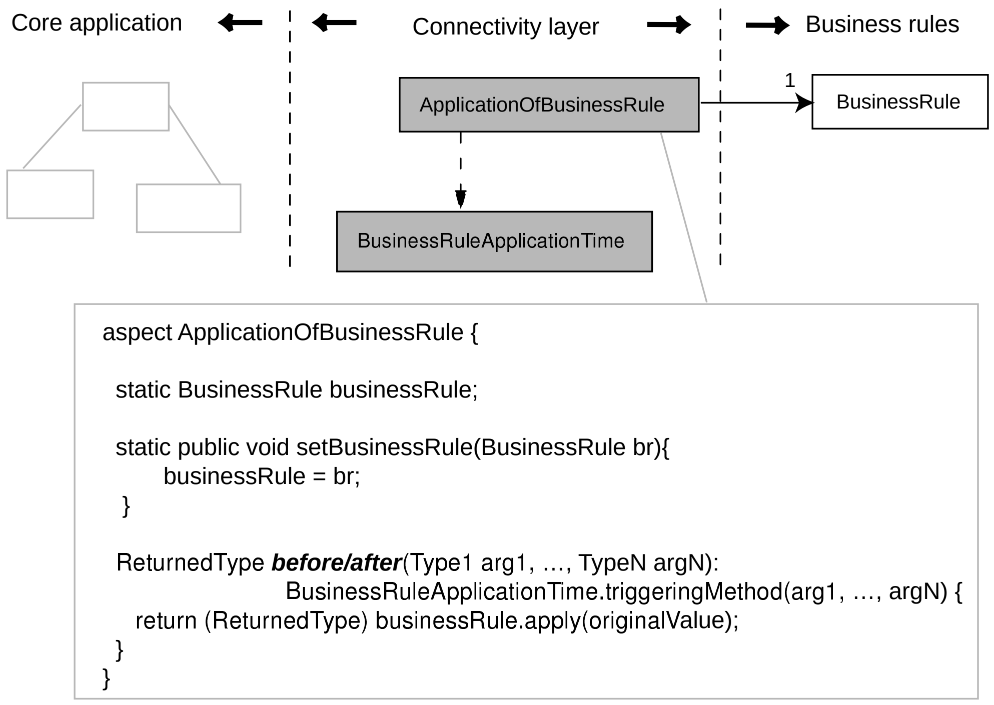
Figure 6.7: Application of one business rule that replaces an existing behavior

Example

Generally, a rule that personalizes the price of products defines, as its action, a new behavior that should replace the normal way the price is calculated. For instance, the ChristmasBR business rule is applied as part of an *around* advice (see 6.4.1) where first the original product price is obtained and then the rule is applied using that value.

- the business rule defines as its action a new functionality to be added before or after some existing behavior.

  For this kind of business rules the application of the rules needs to be specified as *before* or *after* advices (Figure 6.8).



```
aspect ApplicationOfBusinessRule {

  static BusinessRule businessRule;

  static public void setBusinessRule(BusinessRule br){
        businessRule = br;
   }

  ReturnedType before/after(Type1 arg1, …, TypeN argN):
                     BusinessRuleApplicationTime.triggeringMethod(arg1, …, argN) {
    return (ReturnedType) businessRule.apply(originalValue);
  }
}
```

Figure 6.8: Application of one business rule that adds new behavior

Example

Consider the rule presented before for the classification of the customers into frequent and not frequent roles:

"if a customer bought more than 10 items in the shop then consider him/her as a frequent *customer of the shop"*

This business rule is defining some functionality to be performed after certain business operations take place. In this case, after the customer buys some products, we need to check if he/she satisfies the condition to be considered as a frequent one. We need to apply the rule after the customer confirms the purchase. Thus, the application of the business rule is defined as part of an *after* advice.

```
aspect ApplicationOfFrequentBR{

 static FrequentCustomerBR frequentCustomerBR;

 static public void setBusinessRule(FrequentCustomerBR br){
  frequentCustomerBR = br;
 }

 after(Customer c):
        FrequentCustomerBRApplicationTime.checkOut(c){
  frequentCustomerBR.apply(c);
 }
}
```

Regarding the information needed by the rule for its application:

Two cases can be distinguished: the application of a rule that only needs information available at the moment of the application and the application of business rules that have a state (composed by certain business objects captured previously to the application).

As the aspect that defines the application of a business rule needs to know the rule that it has to apply, we might need different instances of that aspect if different instances of the rule can exist:

- A business rule that only uses environmental information or business objects reachable at the moment of its application does not need to keep any state. It just defines the corresponding behavior for its condition and action. Thus, as only one instance of this rule is needed, only one instance of the aspect that defines its application is needed as well.

  This implies that the application aspect can be defined as a singleton (*AspectJ* by default considers the definition of an aspect as a singleton. Figures 6.7 and 6.8 illustrate the default definition of the application aspects as singletons).

  Example

  The aspect that defines the application of the ChristmasBR business rule would look like:

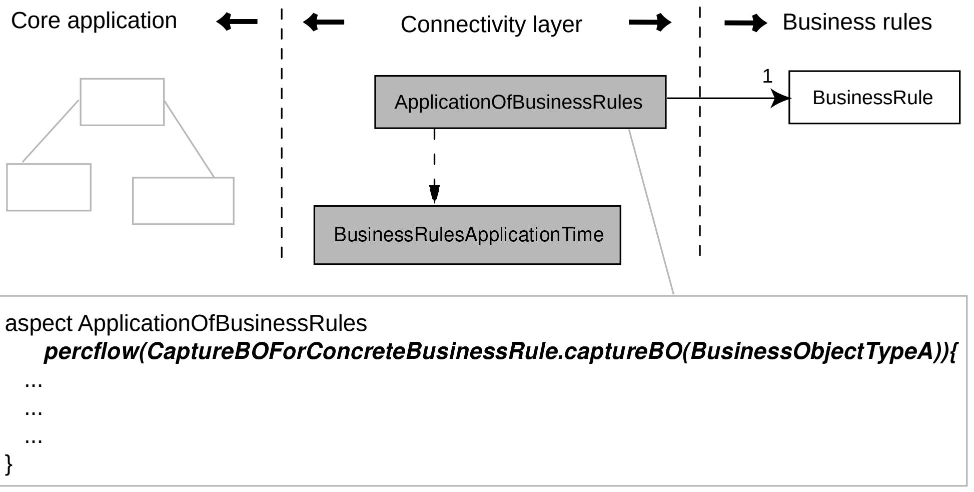
  ```
  aspect ApplicationOfChristmasBRForPrice{

   static ChristmasBR businessRule;

   static public void setBusinessRule(ChristmasBR br){
    businessRule = br;
   }
  ```

```
 Float around(Product p): WhenToPersonalizePrice.
     priceCalculationInCheckOut(p){

 Float price = proceed(p);
  return businessRule.apply(price);
 }
}
```

- A business rule that uses extra business objects not reachable at the moment of the application needs to keep those business objects as part of its state (see 6.3.3). We cannot implement this kind of rules as singletons and thus, a different instance of the aspect that defines its application is needed for each instance of the business rule we need to apply.

  As we explained before, a pointcut is needed to be able to capture the business objects that define the state of the rules. Thus, each time a new control flow of that pointcut occurs, a new instance of the rule is created and a new instance of the application aspect is needed as well.

  The feature *percflow(pointcut designator)* of *AspectJ* is used for the creation of one application aspect instance per control flow of the corresponding pointcut.



```
aspect ApplicationOfBusinessRules
   percflow(CaptureBOForConcreteBusinessRule.captureBO(BusinessObjectTypeA)){
  ...
  ...
  ...
}
```

Figure 6.9: Application of one business rule with state

Example

As we mentioned before, the AmountBR business rule (see 6.3.3) needs to keep track of the customer that is currently buying. The aspect for the application of this business rule is defined as follows: As it is shown in the example, we need a different instance of the application

```
aspect ApplicationOfAmountBRForPrice
 percflow(CaptureBOForAmountBR.checkOut(Customer)){

 AmountBR businessRule;

 public void setBusinessRule(AmountBR br){
  businessRule = br;
 }

 Float around(Product p):
  WhenToPersonalizePrice.priceCalculationInCheckOut(p){

 Float price = proceed(p);
  return businessRule.apply(price);
 }
}
```

for each new control flow of the pointcut checkOut(Customer) defined in CaptureBOForAmountBR (see 6.3.3), aspect for the capturing of the state of the rule being applied. This is achieved with the feature *percflow()* of *AspectJ*. The link between the instance of the application and the correspondent instance of the rule is defined by the aspect CaptureBOForAmountBR.

6.4.2 Application of a set of rules: Combination of rules

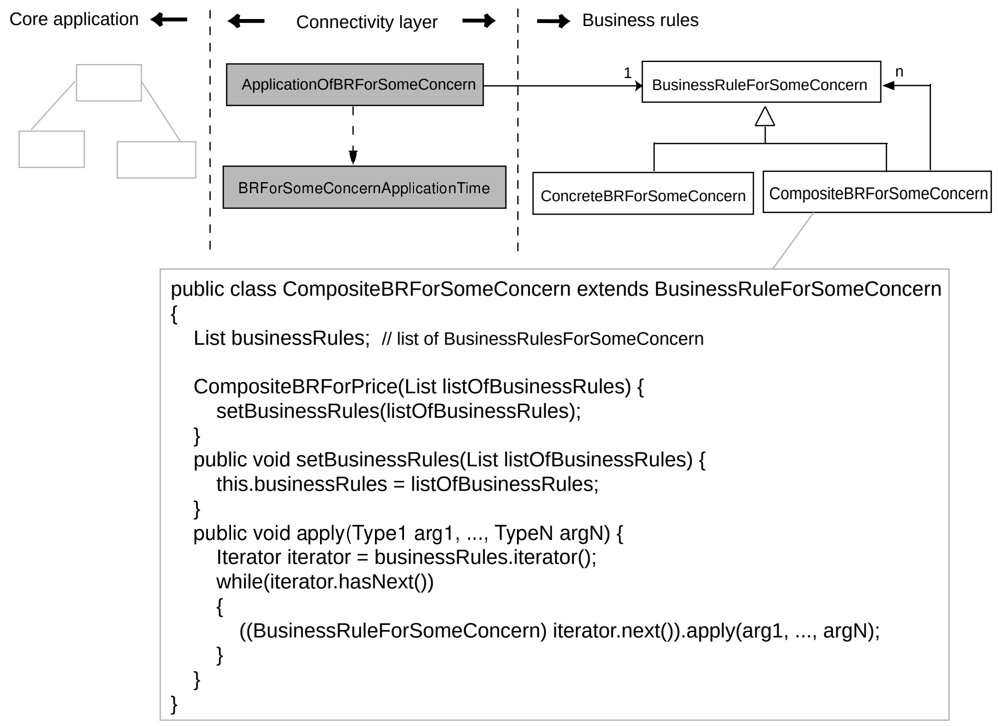
Suppose we have to apply at a certain point a set of rules that specify decisions for some common concern. As it was described in 5.2.4, when several rules are applied at the same time interference between them can occur. Thus, at this point we need to take into account some considerations about how to combine the actions declared by the rules.

Suppose we need to apply at a given moment a set of related rules that define some business considerations regarding the same concern. As many of those rules can be applicable under certain conditions, and their actions can conflict between each other, we need to explicitly provide a way to solve the conflicts that can arise during the combination.

To begin with the simplest case, let's consider that only one way to combine the rules is provided.

Solution

We can use the *composite* pattern [GHJV95] to model compound business rules. The compound business rule would know a set of business rules it is composed of. The way to combine the rules would be defined in the method apply of the compound business rule (see Figure 6.10).


Core application
Connectivity layer
Business rules
ApplicationOfBRForSomeConcern
BRForSomeConcernApplicationTime
1
n
BusinessRuleForSomeConcern
ConcreteBRForSomeConcern
CompositeBRForSomeConcern

```
public class CompositeBRForSomeConcern extends BusinessRuleForSomeConcern
{
    List businessRules;  // list of BusinessRulesForSomeConcern

    CompositeBRForPrice(List listOfBusinessRules) {
        setBusinessRules(listOfBusinessRules);
    }
    public void setBusinessRules(List listOfBusinessRules) {
        this.businessRules = listOfBusinessRules;
    }
    public void apply(Type1 arg1, ..., TypeN argN) {
        Iterator iterator = businessRules.iterator();
        while(iterator.hasNext())
        {
            ((BusinessRuleForSomeConcern) iterator.next()).apply(arg1, ..., argN);
        }
    }
}
```

Figure 6.10: Application of set of business rules: Compound business rule

As we showed previously, either the way the application aspect and the rules themselves are instantiated differs depending on the kind of information the rules manipulate (see 6.4.1).

If we combine rules of only environmental scope, then the application aspect would be defined as a singleton aspect. However, if we combine environmental rules with rules that have state, for which several instances are needed, the application aspect should follow the form of the application for rules with state (several instances for each control flow).

Example

Consider we have to apply both the ChristmasBR and AmountBR rules to personalize the price of products. Thus, both rules need to be applied at the moment when the price is calculated in the context of the checkout process.

We are combining one environmental business rule with one rule that needs specific information. Thus, the way to instantiate the application aspect is driven by the rule with state. The code for the compound business rule would look like:

```
public class CompositeBRForPrice extends BRForPrice{

 List businessRules;
 CompositeBRForPrice(List listOfBusinessRules){
  setBusinessRules(listOfBusinessRules);
 }

 public void setBusinessRules(List listOfBusinessRules){
  this.businessRules = listOfBusinessRules;
 }
 ...
 public Float apply(Float price){
  Iterator iterator = businessRules.iterator();
  while(iterator.hasNext()){
   price = ((BRForPrice) iterator.next()).apply(price);
  }
  return price;
 }
}
```

As it is shown in the previous code fragment, the combination of the rules is solved in the method apply. In this example each rule is applied one after the other, concatenating the discounts defined as the actions of both rules.

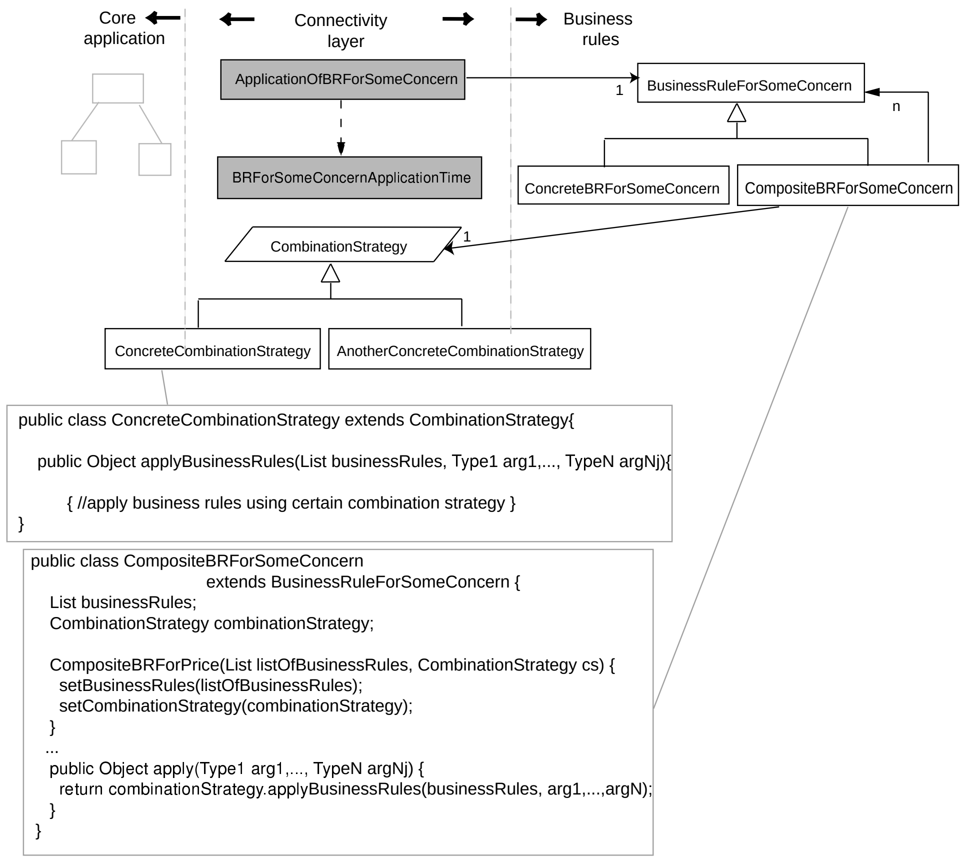
Notice that there might be a problem if we need to combine several business rules that have state. The problematic situation occurs when the business objects that form the state of the rules are picked up at different pointcuts (execution points) for each rule. In this case, as one instance of the application aspect is considered per each control flow of one pointcut, which of the pointcuts to chose to instantiate it?

6.4.3 Several ways to combine the rules

As in the previous case, suppose we need to apply at a certain moment a set of related business rules that define some business considerations regarding a certain concern. Suppose now that we need to consider several ways to solve the combination of those rules actions. As it was mentioned in 5.2, we want to decouple as much as possible the issues related with the combination of rules from the specification of the rules itself.

Solution

The solution consists of abstracting out the way to combine business rules from the compound business rule into a new class hierarchy of combination strategies. In the previous approach, conflicts were solved in the apply method of the compound rule, coupling the definition of the rule itself with the combination concern. Applying the *strategy* design pattern [GHJV95] we can decouple the way the combination is done in a different hierarchy that models combination strategies. Each combination strategy defines a different way to combine business rules (Figure 6.11).


Core application
Connectivity layer
Business rules
ApplicationOfBRForSomeConcern
BRForSomeConcernApplicationTime
CombinationStrategy
1
BusinessRuleForSomeConcern
1
n
ConcreteBRForSomeConcern
CompositeBRForSomeConcern
ConcreteCombinationStrategy
AnotherConcreteCombinationStrategy

```
public class ConcreteCombinationStrategy extends CombinationStrategy{

   public Object applyBusinessRules(List businessRules, Type1 arg1,..., TypeN argNj){

        { //apply business rules using certain combination strategy }
}
```

```
public class CompositeBRForSomeConcern
                                  extends BusinessRuleForSomeConcern {
   List businessRules;
   CombinationStrategy combinationStrategy;

   CompositeBRForPrice(List listOfBusinessRules, CombinationStrategy cs) {
     setBusinessRules(listOfBusinessRules);
     setCombinationStrategy(combinationStrategy);
   }
  ...
   public Object apply(Type1 arg1,..., TypeN argNj) {
     return combinationStrategy.applyBusinessRules(businessRules, arg1,...,argN);
   }
 }
```

Figure 6.11: Application of a set of business rules: Decoupling the combination strategy

Example

Let's consider the combination of the same rules as before, the ChristmasBR and AmountBR rules. Using the *composite* pattern [GHJV95] we can define a compound business rule to group a set of related business rules.

```
public class CompositeBRForPrice extends BRForPrice{

 List businessRules;
 CombinationStrategy combinationStrategy;

 CompositeBRForPrice(List listOfBusinessRules,
                  CombinationStrategy combinationStrategy){
   setBusinessRules(listOfBusinessRules);
   setCombinationStrategy(combinationStrategy);
 }

 public void setCombinationStrategy(CombinationStrategy cs){
  this.combinationStrategy = cs;
 }

 public void setBusinessRules(List listOfBusinessRules){
  this.businessRules = listOfBusinessRules;
 }
 ...
 public Float apply(Float price){
  return combinationStrategy.
               applyBusinessRules(businessRules, price);
 }
}
```

Notice that the method apply of the compound business rule changed from the previous solution. Instead of implementing the combination directly in the same method, it collaborates with a combination strategy object delegating the application of the rules to it.

The different ways to solve the combination of rules is defined as a new hierarchy of combination strategies using the *strategy* pattern [GHJV95]. Each different strategy for the combination is modelled as a subclass of the abstract class CombinationStrategy.

```
abstract class CombinationStrategy{
 abstract public Float applyBusinessRules
                      (List businessRules, Float price);
}
```

For instance, the class ConcatenatingStrategy represents a combination strategy that solves the combination of rules applying each rule after the other, concatenating the discounts.

```
public class ConcatenatingStrategy extends CombinationStrategy{

 public Float applyBusinessRules(List brs, Float price){

  Iterator iterator = brs.iterator();
   while(iterator.hasNext()){
    price = ((BRForPrice) iterator.next()).apply(price);
   }
   return price;
 }
}
```

Another way to combine the rules is shown in the class ExclusionStrategy, where only the first rule that is satisfied is applied, ensuring mutual exclusion between the rules.

```
public class ExclusionStrategy extends CombinationStrategy{

 public Float applyBusinessRules(List brs, Float price){

  Iterator iterator = brs.iterator();
  BRForPrice rule = new AmountBR();
  boolean noRuleHold = true;
  while(iterator.hasNext() && noRuleHold){
   rule = (BRForPrice) iterator.next();
   noRuleHold = !(rule.condition());
  }
  if (!noRuleHold)
   price = rule.action(price);
  return price;
 }
}
```

In the same way these two strategies were contemplated in the example, other ways to solve the combination of a set of rules can be added as subclasses of the CombinationStrategy class.

6.4.4 Combination of business rules based on dynamic conditions

Suppose we need to apply a set of business rules that specify actions regarding some common concern and that several ways to solve the combination of the rules are defined. Moreover, consider that the choice of which strategy to use to solve the combination is made based on conditions that depend on dynamic information. This means that to solve the combination of the business rules we deal with other rules of the kind:

"if some dynamic condition is satisfied then consider a particular way to combine the business rules for this application".

Solution

We need to define new aspects that encapsulate the definition of the conditions used to dynamically decide which combination strategy to use. These conditions are defined in a new aspect as *pointcuts* of the kind *if(condition)*.

These conditions need to be defined as pointcuts because they are used for the instantiation of the application aspects. The application aspects will be instantiated, using the feature *percflow(pointcut designator)*, based on the occurrence of those conditions represented as pointcuts.

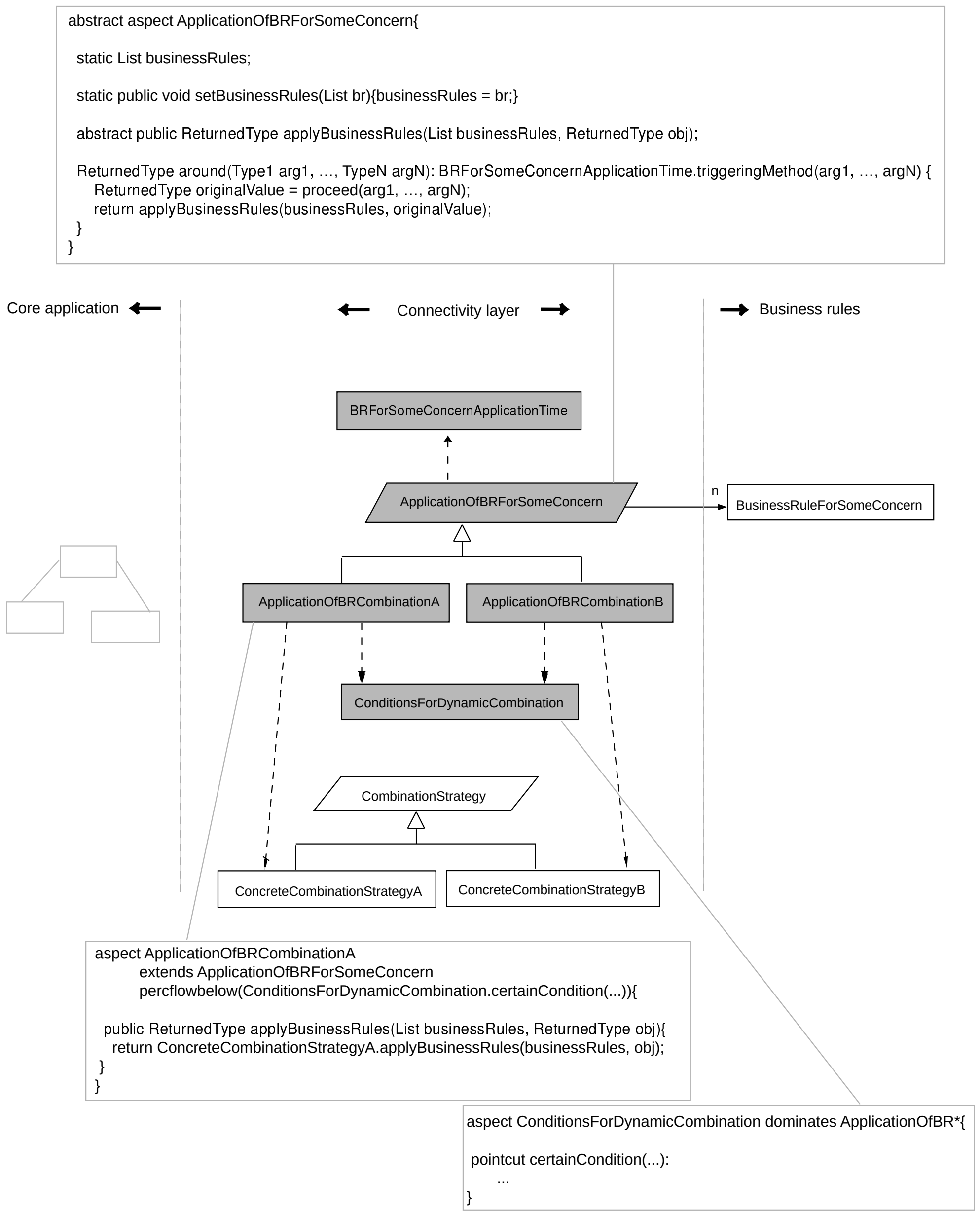
In this solution we do not use the *composite* pattern for modelling compound business rules because the list of business rules is needed to be controlled in the aspect side, as the combination is solved now in that side also. The aspect that specifies the application of the business rules at the specific application time directly knows the set of business rules to apply and its concrete subaspects, in collaboration with the combination strategy hierarchy, apply the rules using the desired strategy. Figure 6.12 shows a scheme of the solution.

Example

Suppose we want to apply both rules the ChristmasBR and the AmountBR when the price is calculated in the checkout process. As both rules specify a different discount to be applied on the price of the products, we need to define how to combine them.

Suppose that the combination used to solve the interference between those two rules depends on some condition that is checked at run-time. For instance, suppose that depending on whether the customer that is currently buying is frequent or not, we choose one combination strategy or the other to decide how to apply and combine the rules. If the customer is frequent we want to apply the rules one after the other concatenating the discounts of each (i.e., using the combination strategy defined in the ConcatenatingStrategy; see the previous example 6.4.3). And if the customer is not a frequent one, we want to apply only the first applicable business rule using the strategy ExclusionStrategy (see previous example 6.4.3).

Thus, we need to define a hierarchy of aspects for the specification of the application. The aspect ApplicationOfBRForPrice defines an around advice on the pointcut that defines the application time for rules that personalize the price. In that advice the application of the rules is triggered. The way the business rules are applied, i.e., the way to combine them is defined in the subaspects of ApplicationOfBRForPrice. Each of the concrete subaspects in-


Core application
Connectivity layer
Business rules
BRForSomeConcernApplicationTime
ApplicationOfBRForSomeConcern
n
BusinessRuleForSomeConcern
ApplicationOfBRCombinationA
ApplicationOfBRCombinationB
ConditionsForDynamicCombination
CombinationStrategy
ConcreteCombinationStrategyA
ConcreteCombinationStrategyB

```
abstract aspect ApplicationOfBRForSomeConcern{

  static List businessRules;

  static public void setBusinessRules(List br){businessRules = br;}

  abstract public ReturnedType applyBusinessRules(List businessRules, ReturnedType obj);

  ReturnedType around(Type1 arg1, …, TypeN argN): BRForSomeConcernApplicationTime.triggeringMethod(arg1, …, argN) {
      ReturnedType originalValue = proceed(arg1, …, argN);
      return applyBusinessRules(businessRules, originalValue);
  }
}
```

```
aspect ApplicationOfBRCombinationA
          extends ApplicationOfBRForSomeConcern
          percflowbelow(ConditionsForDynamicCombination.certainCondition(...)){

  public ReturnedType applyBusinessRules(List businessRules, ReturnedType obj){
    return ConcreteCombinationStrategyA.applyBusinessRules(businessRules, obj);
 }
}
```

```
aspect ConditionsForDynamicCombination dominates ApplicationOfBR*{

 pointcut certainCondition(...):
       ...
}
```

Figure 6.12: Application of set of business rules: combination strategy choice based on dynamic conditions

```
    abstract aspect ApplicationOfBRForPrice{

     static List businessRules;

     static public void setBusinessRules(List br){
      businessRules = br;
     }

     abstract public Float applyBusinessRules
                          (List businessRules, Float price);

     Float around(Product p): WhenToPersonalizePrice.
                            priceCalculusInCheckOut(p){

      Float price = proceed(p);
      return applyBusinessRules(businessRules, price);
     }
    }
```

teracts with the proper combination strategy to define the way the rules are applied and combined. The subaspect ExclusionApplicationOfBR interacts with the ExclusionStrategy combination strategy to apply the rules ensuring mutual exclusion whereas the subaspect ConcatenatingApplicationOfBR collaborates with the combination strategy ConcatenatingStrategy to apply all the discounts specified in the rules that are being applied, one after the other.

```
aspect ExclusionApplicationOfBR
        extends ApplicationOfBRForPrice
                percflowbelow(ConditionsForDynamicCombination.
                              frequentCustomer(Customer)){

 public Float applyBusinessRules(List rules, Float price){
        return ExclusionStrategy.
                applyBusinessRules(rules, price);
 }
}

aspect ConcatenatingApplicationOfBR
        extends ApplicationOfBRForPrice
                percflowbelow(ConditionsForDynamicCombination.
                      noFrequentCustomer(Customer)){

 public Float applyBusinessRules(List rules, Float price){
        return ConcatenatingStrategy.
                applyBusinessRules(rules, price);
 }
}
```

Notice that both aspects ExclusionApplicationOfBR and ConcatenatingApplicationOfBR are instantiated using *AspectJ*'s feature *percflowbelow(Pointcut)* creating a new instance of the aspect for each entrance to the control flow below the join points defined by *Pointcut*. In this example, we want to apply the rules using mutual exclusion when the customer is a frequent one. Thus, a new instance of the ExclusionApplicationOfBR aspect is needed if the customer is a frequent one. That is specified defining the instantiation of the aspect with the expression percflowbelow(ConditionsForDynamicCombination.frequentCustomer(Customer)).

The pointcut frequentCustomer defined in the aspect ConditionsForDynamicCombination specifies the condition that tests if the customer that is checking out is frequent.

Analogously, we want to apply the rules concatenating all the discounts specified in the rules when the customer is not a frequent one. Thus, a new instance of the ConcatenatingApplicationOfBR aspect is needed if the customer is not frequent. That is specified defining the instantiation of the aspect with the expression percflowbelow(ConditionsForDynamicCombination.noFrequentCustomer(Customer)).

The pointcut noFrequentCustomer defined in the aspect ConditionsForDynamicCombination specifies the condition that tests if the customer that is checking out is not frequent.

```
aspect ConditionsForDynamicCombination
                        dominates *ApplicationOfBR{

 pointcut frequentCustomer(Customer c):
   call(Float CheckOut.checkOut(Customer)) && args(c)
     && if(c.getAccount().isFrequent());

 pointcut noFrequentCustomer(Customer c):
   call(Float CheckOut.checkOut(Customer)) && args(c)
     && if(!c.getAccount().isFrequent());
}
```

Notice that for a given customer, either one of the two pointcuts occurs because either a customer is frequent or not frequent. This implies that for a given customer during the checkout process either one of the concrete aspects ExclusionApplicationOfBR or ConcatenatingApplicationOfBR is instantiated. As a consequence, the way to apply the rules for price personalization is customized depending on the characteristics of the customer.

This mechanism allows us to decide dynamically for each customer which combination strategy to choose to combine the rules for the personalization of the price.

It is important to mark that it is at this point, i.e., when dealing with combination of business rules, where the approach that considers rules as aspects is limited. Even if we could make use of some features provided by *AspectJ*, like the *dominates* feature to establish priorities between aspects and thus between rules, the combination can be achieved only until a certain extent. As an example, we could say that "ChristmasBR dominates AmountBR", meaning that the application of the rule ChristmasBR should have precedence over the one of AmountBR. But the support for combinations we could achieve considering rules as aspects was limited to the definition of the rule aspects (compile time), not being possible to decouple the combination from the rule itself, and of course, not being possible to change the way the combinations are solved at run time.

Moreover, if we consider rules as aspects, we can think in writing specific aspects to control the combination of rules. For instance, we can express an aspect that ensures the mutual exclusion between two rule aspects. But some problems appear when trying to relate aspect instances. Little support for relating aspect instances is provided by *AspectJ*. Moreover with this approach it is not possible to dynamically change policies for the combination of rules depending on dynamic conditions. For instance, if we want to consider different orderings, priorities or combinations according to some dynamic information (e.g., depending on the customer being a frequent one), or we want to apply certain policies when some dynamic conditions are satisfied (for instance, when the customer spent more than certain amount of money), the approach of having rules as aspects is not flexible enough to achieve it.

6.5 Special cases of business rules

6.5.1 Conditions as type checks

Up until now we have considered that basically an "if then" business rule can be suitably represented using the *rule object pattern* [Arsa], by means of the definition of a condition and an action. However, this solution is not always the most suitable for any situation.

For instance, there are business rules that define conditions that deal with the possible types of an object. This conditions usually involve checking the type of certain objects.

Some design options can be considered when dealing with rules that check the type of an object:

- **Definition of an interface to query the type of an object:**

One possible solution consists of the definition of an interface that defines methods to query the type of an object and make the classes of the queried objects implement it.

In this solution we can still model the rules using the rule object pattern. The condition of the rules would use the query methods defined in the interface to query the type of the objects.

This solution has the advantage that either the condition and action will still remain encapsulated in the rule itself, completely decoupled from the rest of the application.

- **Definition of a "hook" in the types checked:**

  Another alternative is to define a new method in the types that are being checked as a hook to which all the actions of business rules that check those types can be mapped to.

  This solution, though it implies only a small invasive change, it affects the idea of separation of concerns, as actions of independent business rules would be coupled together.

- **Delegating business rules:**

  As it is also identified in [RFC], *the rule object pattern* is not always suitable for expressing all kinds of business rules, in particular when dealing with rules whose conditions imply checking the type of objects, and that specify actions to perform according the result of the type check. These rules are of the form:

  "if certain object is of type A then perform action1"

  "if certain object is of type B then perform action2"

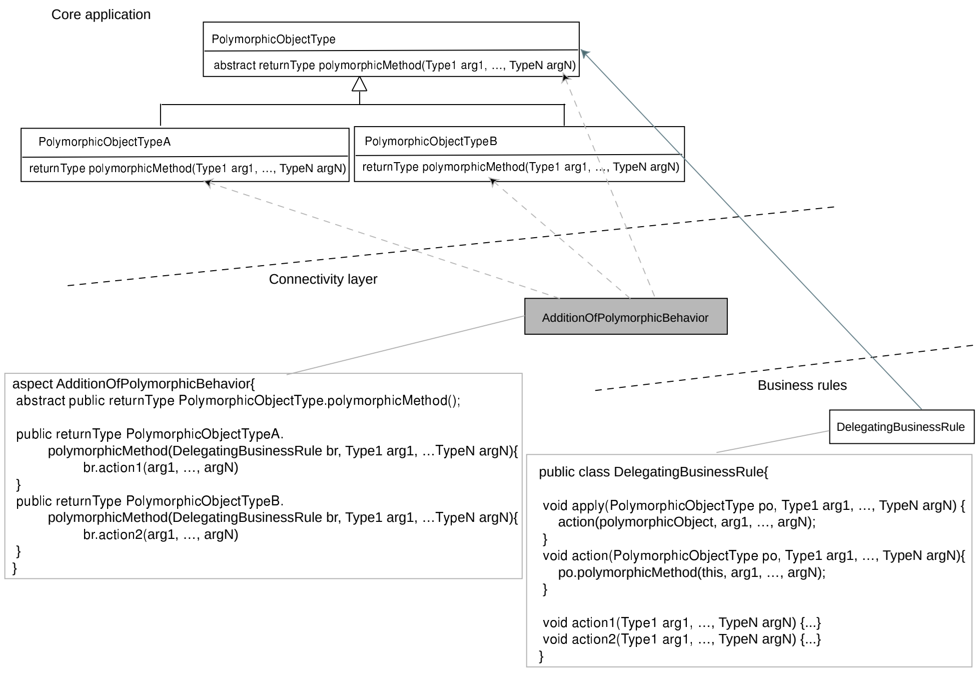
  For this kind of rules, the *rule object pattern* might not the best option because we will end up writing conditions that directly ask the type of objects, and this is not appropriate from the point of view of the object-oriented paradigm.

  We think that a solution that uses the power of the polymorphism instead of explicitly asking the types would be better from an object-oriented design point of view.

  The suggested solution uses the power of the static crosscutting provided by *AspectJ*. The idea consists in the non-invasive insertion of a polymorphic method in each of the classes that form the hierarchy of types checked. This method works as the trigger point for the application of the rules. Each of the types involved needs to redefine that polymorphic method with the triggering of the corresponding action to be performed for that type. This is done in collaboration with the rule itself, which is the one that defines all the possible actions to be performed for all the cases.

  For instance, in the type A the inserted trigger point would invoke on the business rule the execution of the action1; in the type B the inserted trigger point would invoke the execution of the action2, and so on.

In this way, the rules would only need to delegate the decision of which action the trigger to the polymorphic objects. The condition, i.e., the query on the type of the objects, is checked implicitly by means of polymorphism (see Figure 6.13).



Core application
PolymorphicObjectType
abstract returnType polymorphicMethod(Type1 arg1, ..., TypeN argN)
PolymorphicObjectTypeA
returnType polymorphicMethod(Type1 arg1, ..., TypeN argN)
PolymorphicObjectTypeB
returnType polymorphicMethod(Type1 arg1, ..., TypeN argN)
Connectivity layer
AdditionOfPolymorphicBehavior
Business rules
DelegatingBusinessRule

```
aspect AdditionOfPolymorphicBehavior{
 abstract public returnType PolymorphicObjectType.polymorphicMethod();

 public returnType PolymorphicObjectTypeA.
        polymorphicMethod(DelegatingBusinessRule br, Type1 arg1, ...TypeN argN){
              br.action1(arg1, ..., argN)
 }
 public returnType PolymorphicObjectTypeB.
        polymorphicMethod(DelegatingBusinessRule br, Type1 arg1, ...TypeN argN){
              br.action2(arg1, ..., argN)
 }
}
```

```
public class DelegatingBusinessRule{

 void apply(PolymorphicObjectType po, Type1 arg1, ..., TypeN argN) {
    action(polymorphicObject, arg1, ..., argN);
 }
 void action(PolymorphicObjectType po, Type1 arg1, ..., TypeN argN){
    po.polymorphicMethod(this, arg1, ..., argN);
 }

 void action1(Type1 arg1, ..., TypeN argN) {...}
 void action2(Type1 arg1, ..., TypeN argN) {...}
}
```

Figure 6.13: Business rules that check the type of objects

Two cases can be identified for this context:

Anticipated types

There are business rules that basically check the type of an object where the types correspond to classes or interfaces anticipated from the beginning in the core application.

Solution

The suggested solution implies the definition of an aspect that extends, using *Introduction*, the business objects types with the polymorphic method for the triggering of the action of the rules. In this case the business rules would only need to delegate on the objects the decision of which effective action to perform. The rules do not need to define the condition as an explicit method. The condition is implicitly checked through polymorphism.

Example

Suppose we have the following rule:

"if the customer is a manager member then show the welcome message "Welcome Manager member", and if he/she is a sales one, show the message "Welcome Sales member"".

Clearly, this business rule is checking the type of the objects of the type Customer and performing a different behavior depending on the kind of customer that logs in to the shop.

The solution consists in extending the Customer hierarchy with the addition of the polymorphic method triggerWelcomeMethod. This method is redefined in each of the subclasses in the Customer hierarchy to trigger the invocation of the corresponding action on the rule to obtain the welcome message for that specific type of customer.

```
aspect AdditionOfWelcomeMessageBehavior{

 abstract public String Customer.
                      welcomeMessage(CustomerBR br);

 public String Manager.welcomeMessage(CustomerBR br){
  return br.welcomeManager();
 }

 public String Sales.welcomeMessage(CustomerBR br){
  return br.welcomeSales();
 }
}
```

The aspect AdditionOfWelcomeMessageBehavior uses *Introduction* to add the polymorphic method in each of the types of the customer hierarchy. A different implementation for the method welcomeMessage() is provided for both subclasses Manager and Sales to trigger the printing of the correct welcome message for each case. In this way, the class Manager triggers the execution of the action welcomeManager defined in the CustomerBR and analogously Sales invokes on the rule the method welcomeSales.

```
    public class CustomerBR{

     void action(Customer customer) {
      customer.welcomeMessage(this);
     }

     public void welcomeSales() {
      System.out.println("Welcome Sales member");
     }

     public void welcomeManager() {
      System.out.println("Welcome Manager member");
     }

     void apply(Customer customer) {
      action(customer);
     }
    }
```

The business rule CustomerBR delegates on the customer object (received as parameter in the apply method) the decision of which message to show by means of invoking the polymorphic method welcomeMessage().

```
aspect ApplicationOfCustomerBR{

 static CustomerBR businessRule;

 static public void setBusinessRule(CustomerBR br){
  businessRule = br;
 }

 after(Customer c):
  WhenToPersonalizeWelcomeMessage.customerLoggedIn(c){
   businessRule.apply(c);
 }
}
```

The aspect ApplicationOfCustomerBR defines the application of the CustomerBR as an after advice. This advice is defined on the pointcut customerLoggedIn that captures the moment when the customer logs in the shop (see definition of aspect WhenToPersonalizeWelcomeMessage below) and the welcome message should be shown.

```
aspect WhenToPersonalizeWelcomeMessage {

 pointcut customerLoggedIn(Customer c):
  target(c) && call(boolean Customer.login(String, String));
}
```

Unanticipated types

This case refers to the business rules that as the condition for their applicability check the type of an object and where the types were not anticipated from the beginning. This means that the types that are checked are needed only from the point of view of the business rules logic. They can be seen as roles imposed by the business considerations.

Solution

The solution consists in the definition of delegating business rules (as in the previous case) and also in the non-invasive addition of the unanticipated types. This is achieved by means of the static crosscutting provided by *Introduction.*

Example

Suppose that in the previous example, the types Manager and Sales were not anticipated from the beginning. Thus, they need to be added to the core application using *Introduction* as subclasses of the Customer class.

```
aspect AdditionOfWelcomeMessageBehavior{

 declare parents: Manager extends Customer;

 declare parents: Sales extends Customer;
 ...//same code as previous solution
}
```

6.5.2 Business rules that select other business rules

There are business rules that take decisions about other business rules. They can for instance chose which rules to consider to be applied at a certain application time.

6.5.2.1 Decision time before the application time

Suppose the decision of which rules to chose to be applied at a certain point is taken before the application time arrives.

Solution

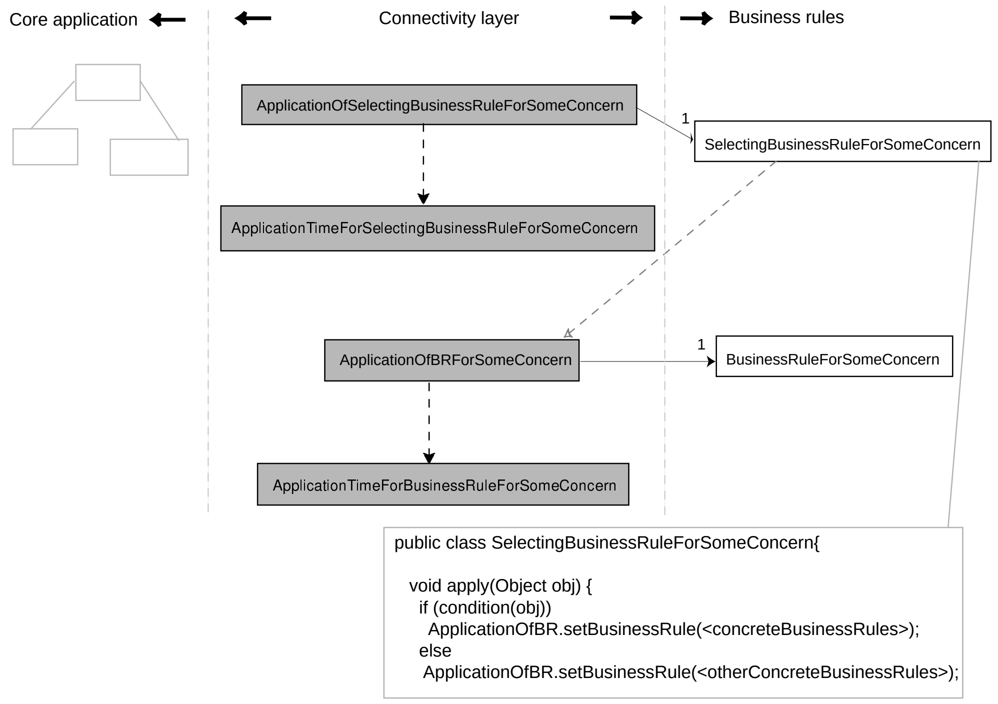
We need to define an aspect that specifies, through the definition of a pointcut, the moment when the selection must take place. The business rule that performs the selection is applied at that time. The selection of the desired set of rules is defined as the action of the rule.

Example

Suppose the following business rule:

"if the customer is a frequent one, then consider the ChristmasBR rule to be applied in the personalization of the price. Otherwise, consider the AnniversaryShopBR rule".

```
public class SelectionBR{
```


Core application
Connectivity layer
Business rules
ApplicationOfSelectingBusinessRuleForSomeConcern
1
SelectingBusinessRuleForSomeConcern
ApplicationTimeForSelectingBusinessRuleForSomeConcern
ApplicationOfBRForSomeConcern
1
BusinessRuleForSomeConcern
ApplicationTimeForBusinessRuleForSomeConcern

```
public class SelectingBusinessRuleForSomeConcern{

   void apply(Object obj) {
     if (condition(obj))
       ApplicationOfBR.setBusinessRule(<concreteBusinessRules>);
     else
      ApplicationOfBR.setBusinessRule(<otherConcreteBusinessRules>);
```

Figure 6.14: Business rules that select other business rules: selection time before the application time.

```
 public boolean condition(Customer customer){
  return customer.getAccount().isFrequent();
 }

 public void  selectRuleForFrequent(){
  ApplicationOfBRForPrice.setBusinessRule(new ChristmasBR());
 }

 public void  selectRuleForNotFrequent(){
  ApplicationOfBRForPrice.setBusinessRule(new AnniversaryShopBR());
 }

 public void apply(Customer customer) {
  if (condition(customer))
     selectRuleForFrequent();
  else selectRuleForNotFrequent();
 }
}
```

Notice that we need to define two methods for the action, because the rule has the form *if then else*.

```
aspect WhenToSelectPriceRule {

 pointcut customerLoggedIn(Customer c):
  target(c) && call(boolean Customer.login(String, String));
}
```

The aspect WhenToSelectPriceRule defines, trough the definition of the pointcut customerLoggedIn, the moment when the customer logs in the shop. This is the moment when the selection of the rules for the price personalization has to take place.

```
aspect ApplicationOfSelectionBR{

 static SelectionBR businessRule;

 static public void setBusinessRule(SelectionBR br){
  businessRule = br;
 }

 after(Customer c): WhenToSelectPriceRule.customerLoggedIn(c){
  businessRule.apply(c);
 }
}
```

The aspect ApplicationOfSelectionBR triggers the application of the rule SelectionBR when the customer logs in. The aspect ApplicationOfBRForPrice defines the triggering of the application of the rules for price personalization.

```
aspect ApplicationOfBRForPrice{

 static BRForPrice businessRule;

 static public void setBusinessRule(BRForPrice br){
  businessRule = br;
 }

 Float around(Product p):
  WhenToPersonalizePrice.priceCalculationInCheckOut(p){
   Float price = proceed(p);
   return businessRule.apply(price);
 }
}
```

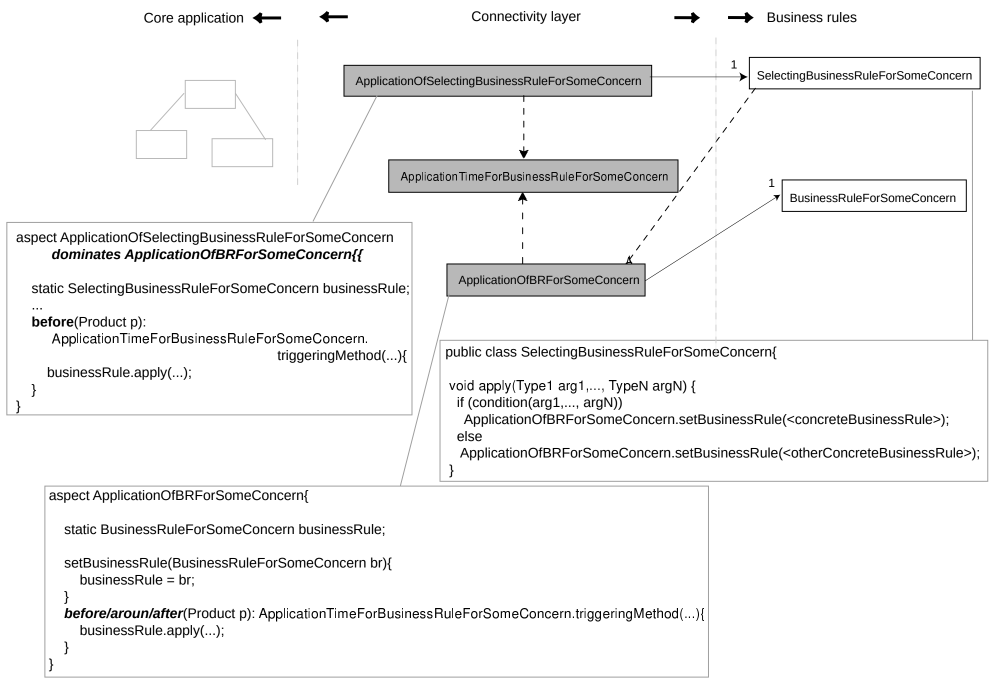
Notice that the business rule SelectionBR that performs the selection of the rules to consider for the price personalization needs to interact with the aspect ApplicationOfBRForPrice to provide the selected business rules.

6.5.2.2 Decision and application at the same time

Suppose the decision of which rules to chose to be applied at a certain point is taken at the same moment when the application must take place.

Solution

This business rule defines the selection of the desired rules to be applied as its action. It has to be applied at the same time of the application of the chosen rules. Thus, the application aspect for the selection of rules must be defined in terms of the same application time pointcut that captures the moment of the application for the chosen rules. Thus, it must define a before advice on the same pointcut that captures the moment for the application of the selected rules (Figure 6.15).



Core application
Connectivity layer
Business rules
ApplicationOfSelectingBusinessRuleForSomeConcern
1
SelectingBusinessRuleForSomeConcern
ApplicationTimeForBusinessRuleForSomeConcern
1
BusinessRuleForSomeConcern
ApplicationOfBRForSomeConcern

```
aspect ApplicationOfSelectingBusinessRuleForSomeConcern
        dominates ApplicationOfBRForSomeConcern{{

    static SelectingBusinessRuleForSomeConcern businessRule;
    ...
    before(Product p):
        ApplicationTimeForBusinessRuleForSomeConcern.
                                        triggeringMethod(...){
        businessRule.apply(...);
    }
}
```

```
public class SelectingBusinessRuleForSomeConcern{

 void apply(Type1 arg1,..., TypeN argN) {
   if (condition(arg1,..., argN))
     ApplicationOfBRForSomeConcern.setBusinessRule(<concreteBusinessRule>);
   else
    ApplicationOfBRForSomeConcern.setBusinessRule(<otherConcreteBusinessRule>);
 }
```

```
aspect ApplicationOfBRForSomeConcern{

    static BusinessRuleForSomeConcern businessRule;

    setBusinessRule(BusinessRuleForSomeConcern br){
        businessRule = br;
    }
    before/aroun/after(Product p): ApplicationTimeForBusinessRuleForSomeConcern.triggeringMethod(...){
        businessRule.apply(...);
    }
}
```

Figure 6.15: Business rules that select other business rules: selection and application at the same time.

Example

Consider the following business rule:

"if a product was already bought more than 100 times, then consider the ChristmasBR rule for the price personalization. Otherwise, consider the AnniversaryShopBR rule".

The only difference with the previous solution is the definition of the aspect ApplicationOfSelectionBR for the application of the rule that performs the selection.

```
aspect ApplicationOfSelectionBR{

 static SelectionBR businessRule;

 static public void setBusinessRule(SelectionBR br){
  businessRule = br;
 }

 before(Product p):
        WhenToPersonalizePrice.priceCalculusInCheckOut(p){
  businessRule.apply(p);
 }
}
```

In this case, as either the rule for the selection and the personalization of the price are applied at the same moment, only one aspect for the definition of the application moment is needed (the aspect WhenToPersonalizePrice).

As both rules, the selecting rule and the price personalization rule, are applied at the same time, when the price is calculated, both aspects ApplicationOfSelectionBR and ApplicationOfBRForPrice trigger the application of the respective rules as an advice on the same pointcut priceCalculusInCheckOut.

Even if the pointcut for the application of both rules, the selection rule and the price personalization one, is the same, the selection is applied making use of a before advice to occur before the personalization itself. In case the selected business rules were applied within a *before* advice, we would need to make use of the feature *dominates* to ensure that the selection of the rules has priority and thus occurs before the application of the rules itself.

Chapter 7

General conclusion

7.1 Conclusion

Business rules tend to change and evolve rapidly to cope with changes in the volatile business world. Thus, it is desirable to keep them as decoupled as possible from the rest of the application that implements the core business processes. To achieve this, separation of concerns is fundamental. Nevertheless, it is usually hard and not straightforward to separate the concerns involved in the management of business rules. Usually these concerns are not localized but scattered through different parts of the system and even tangled with software that addresses other concerns. Thus, business rules concerns are generally hard to modularize fitting the dominant decomposition mechanism that current software engineering techniques provide and we believe that this is the reason for their complexity.

As part of this dissertation we have been extensively studying and analyzing the different approaches in the literature that address the problem of decoupling business rules. A lot of research has focused and agreed on the need of keeping business rules as separate as possible from the core application [IBMb] [CRFS01] [RDR$^+$] [Vae01]. However, not much support has been found for the detailed linking of the rules; not many approaches have given special consideration to the issues involved when we need to connect or weave the rules into the rest of the application to obtain the desired software system. Many requirements have to be taken into account for a mechanism that allows having the business rules decoupled and their subsequent connection with the rest of the application. Our approach focuses on this connection part, i.e., the linking of the rules with the core application. We have analyzed the necessary ingredients for connecting, configuring and applying decoupled business rules and come up with a detailed list of the requirements needed. This list constitutes a first contribution of this dissertation.

To deal with these requirements, extra support for separation of concerns is needed. That was the motivation to look at current Aspect-Oriented Software Development techniques, because they have been successfully applied to encapsulate and explicitly represent concerns that inherently crosscut the natural modularity of the rest of the implementation.

Contrary to many of the approaches in the literature (which are usually based on rule engines) our approach focuses on the use of an advanced Aspect-Oriented Programming technology called *AspectJ* for achieving the weaving of the rules into the application and deal with different configuration issues taken into account also as part of the connectivity layer. Based on the capabilities of *AspectJ* we came up with a set of generic solutions for the identified requirements. These generic solutions constitute a second contribution of this thesis.

On one hand the features provided by *AspectJ* are useful and perfectly suit the kind of problems we need to solve. For instance, the dynamic join point model provided by *AspectJ* allows us to capture well-defined points in the execution of a program that can be used to specify the time when business rules need to be applied. Another interesting feature is the support for static crosscutting provided by means of *Introduction*. *Introduction* enables the non-invasive extension of the application code with information that was not anticipated from the beginning and that is needed by the business rules logic.

On the other hand, *AspectJ* suffers from some limitations as well. One of the main *AspectJ*'s limitations found during the experiments was the lack of support for the communication and relation of aspect instances. As mentioned before, our approach focuses on separation of concerns, not only decoupling concerns from the core application and the business rules but also stressing the decoupling even further, among the different issues involved in the management and configuration of the rules. This implied the definition of several different aspects as part of the connectivity layer, each of them in charge of specific behavior. Several instances of these aspects were needed in some cases. All these aspect instances need to interact and communicate and at that point is where the power of *AspectJ* falls short. When aspect instances need to share state it is always possible to merge them in only one aspect; but in some situations the aspect instances need to interact in a more intelligent way, and they need to be kept separate because their behaviors are inherently different. These are the situations where relations between instances are usually solved ad-hoc in *AspectJ*. These limitations motivate the need of improvements to the *AspectJ* language.

To summarize, Aspect-Oriented Programming techniques are suitable for expressing the kind of concerns we need to address when dealing with business rules. We have experimented with *AspectJ* as an Aspect-Oriented Programming language that supports dynamic and static crosscutting. Generally speaking, the results achieved are very positive. We conclude that *AspectJ* is suitable for addressing most of the identified requirements for the decoupling, configuration and application of business rules.

7.2 Future Work

The work done in this dissertation can be continued in several directions:

- Exploration of other Aspect-Oriented Programming approaches to determine until which extent the problems solved in this thesis can be addressed with other AOP techniques as well.

  In the process of deciding which AOP techniques could be useful to solve the kind of problems we needed to address, we have looked and analyzed other AOP techniques as well. Based on this analysis we have concluded that *AspectJ* has a lot of useful features that can be used for the configuration and connection of the business rules with the core application and thus we have adopted it to base the generic solutions on.

  The approach of Composition Filters is similar to *AspectJ* in the way the join point model is defined. Although we believe many of the results achieved with *AspectJ* could be obtained using *Composition Filters*, a deep analysis of the suitability of *Composition Filters* could be performed.

  It is important to note that the solutions presented so far depend on which business objects and relations between them are considered in the core application. If we would need to express business rules independently of the concrete business objects and relations contemplated in the core application we would need to make abstraction of the concrete class graph. *Adaptive Programming* [LOO01] would be suitable to achieve the independency from the concrete class hierarchy, allowing the business rules to be expressed in a more independent way avoiding referring to concrete business object hierarchy.

  Another technique to further explore is *HyperJ* [TOHS99]. Due to the way *HyperJ* enables crosscutting implementation, it could be suitably used to express business rules that imply mappings of concepts.

  Thus, further research can be done to analyze and determine with more detail the suitability of the other AOP approaches for the typical issues of business rules.

- Address the requirements for business rules in earlier stages in the software lifecycle.

  Some research could be done to incorporate the externalization of business rules within the analysis and design process. To achieve that, some notational support should be provided at the design level for expressing the places in the model where the rules must be applied, which information is required and how the rules are combined.

- Automatic support for the generation of the aspects that form the connectivity layer in between the business rules and the core application can be provided.

  Depending on the representation of the rules it could be possible to perform analysis to automatically derive the join points where we need to apply the business rules and determine which information the rules need for their application.

  However, it would be difficult and not straightforward to automatically determine which generic solutions to apply for a given business rule. First of all, to achieve the automatization, some way of describing the points where the rules are applied should be provided at design time to be able to map these design elements to code. Some research has been done to automatically derive the points in the code where constraints should be checked and automatically generate and insert the code needed to trigger the checking [Str00] [Ver01]. But complexity increases when dealing with more complex business rules; several kinds of information can be used, conflicts can occur between rules, the application can depend on dynamic conditions, different strategies for the application can be considered, etc. However, some automatic support can be provided to help the software engineer in identifying the needed ingredients for business rules.

Bibliography

[AB] Alan Abrahams and Jean M. Bacon. Event-centric policy specification for e-commerce applications.

[AFW] L.F. Andrade, J.L. Fiadeiro, and M. Wermelinger. Enforcing business policies through automated reconfiguration.

[Arsa] Ali Arsanjani. Rule object 2001: A pattern language for adaptive and scalable business rule construction.

[Arsb] Ali Arsanjani. Using grammar-oriented object design to seamlessly map business models to component-based software architectures.

[Asp] AspectJ. http://www.aspectj.org.

[bus00] Defining business rules ∼ what are they really? the Business Rules Group, July 2000.

[Cen01] Information Technology Support Center. Ui tax and benefits system modernization: Business rules, 2001.

[Cra] Diane Crawford. *Communications of the ACM*, 44(10).

[CRFS01] Juan Manuel Cappi, Gustavo H. Rossi, Andrés Fortier, and Daniel Schwabe. Seamless personalization of e-commerce applications. In Springer-Verlag, editor, *2nd International Workshop on Conceptual Modeling Approaches for e-Business*, December 2001.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[GBN] Jeff Gray, Ted Bapty, and Sandeep Neema. Aspectifying constraints in model-integrated computing.

[GBNT01] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GMB82] S. J. Greenspan, J. Mylopoulos, and A. Borgida. Capturing more world knowledge in the requirements specification. In *Proceedings of the 6th International Conference on Software Engineering*, pages 225–235. IEEE Computer Society Press, September 1982. This paper was cited as the best/most influential paper by the program committee for ICSE 16 in 1992.

[GRF$^+$01] Sofie Goderis, Gustavo H. Rossi, Andrés Fortier, Juan M. Cappi, and Daniel Schwabe. Combining meta-level and logic-based constructs in web personalization. In *Active Media Technology*, volume 2252 of *Lecture Notes in Computer Science*, pages 47–64, 12 2001.

[HL95] Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, February24 1995.

[IBMa] IBM. http://www.research.ibm.com/aem/brb.html.

[IBMb] IBM. http://www.research.ibm.com/rules.

[ILO] ILOG. Ilog rules, white paper.

[KHH$^+$01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[KHH$^+$01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, 2001.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KRS00] Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Modeling customizable web applications. In *Kyoto International Conference on Digital Libraries*, page 387, 2000.

[LOO01] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.

[Lou] Isabelle Rouvellou Lou. Combining different business rules technologies: A rationalization.

[OMG01] OMG. Object constraint language specification, omg-unified modeling language, v1.4 6-1, 2001.

[OT01] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, October 2001.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Per] Alan Perkins. Business rules = meta-data. Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00).

[RDR$^+$] Isabelle Rouvellou, Lou Degenaro, Kevin Rasmus, Dave Ehnebuske, and Barbara McKee. Extending business objects with business rules.

[RFC] Gustavo H. Rossi, Andrés Fortier, and Juan M. Cappi. Mapping personalization policies into software structures. In *2nd International Conference on Adaptive Hypermedia and Adaptive Web Based Systems 2002, Workshop on Recommendation and Personalization in eCommerce.*

[SAA$^+$99] Guus Schreiber, Hans Akkermans, Anjo Anjewierden, Robert de Hoog, Nigel Shadbolt, Walter Van de Velde, and Bob Wielinga. *Knowledge Engineering and Management: The Common KADS Methodology.* MIT Press, 1999.

[Str00] Raignhild Van Der Straeten. Using and enforcing constraints in object-oriented application development, 2000.

[Sys] Michel Tilman System. Position paper for the tools and environments for business rules workshop of the ecoop '98 conference.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Con-*

ference on Software Engineering, pages 107–119. IEEE Computer Society Press / ACM Press, 1999.

[Vae01] R.G.H Vaessen. Business logic extraction, June 2001.

[Ver01] Bart Verheecke. From declarative constraints in conceptual models to explicit constraint classes in implementation models, 2001.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, 2001.

[YBJ] Joseph Yoder, Federico Balaguer, and Ralph Johnson. Adaptive object-models for implementing business rules. Position Paper for Third Workshop on Best-practices for Business Rules Design and Implementation, OOPSLA 2001.

[YJ] Joseph Yoder and Ralph Johnson. The adaptive object-model architectural style. Working IEEE/IFIP Conference on Software Architecture (WICSA) 2002.