Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica
System and Software Engineering Lab

# Connecting High-Level Business Rules with Object-Oriented Applications: An approach using Aspect-Oriented Programming and Model-Driven Engineering

**María Agustina Cibrán**
June 2007

*Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen*

**Promotors:** Prof. Dr. Viviane Jonckers, Dr. Maja D'Hondt

Print: Grafikon, Oostkamp

VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. ++32 (0)2 289 26 50
Fax ++32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5487 428 7
NUR 980
Legal deposit D/2007/11.161/018

# Abstract

This dissertation addresses the problem of connecting high-level, executable business rules with existing object-oriented applications.

State-of-the-art research on developing object-oriented software applications with rule-based knowledge advocates making rules explicit and separate from the object-oriented core functionality. Although many approaches target this goal and are a considerable improvement on the solution that embeds rules as conditional statements in object-oriented applications, these approaches still suffer from three major inherent problems.

First, even when business rules are successfully decoupled, the rule connection code is still tangled with and scattered in the implementation of the core application functionality. Therefore, when existing business rules need to be integrated differently, or when new business rules need to be connected at unanticipated events, the source code of the core application must be adapted manually at different places. Consequently, it becomes difficult to localize, add, change or remove rule connections.

Secondly, regardless of the approach taken to decouple the business rules, executable rules are ultimately low-level. This makes rules not understandable to domain experts who are not adept at programming. A third, and closely related problem is a tight coupling between the business rules and the existing implementation of the core application. This causes rules to be fragile and not reusable, and prohibits the non-invasive realization of rules in terms of unanticipated implementation elements. As a consequence, business rules cannot be deployed by the domain experts without the intervention of a developer.

This dissertation presents a comprehensive solution to these problems, enabling existing applications to integrate business rules at the domain level. The first problem is addressed by encapsulating the rule connection in a separate module, decoupled from both the core application functionality and the rules. This decoupling is not straightforward because rule connections *crosscut* the core application functionality. *Aspect-Oriented Programming* (AOP) provides new modularization mechanisms, i.e. aspects, for the encapsulation of crosscutting code while ensuring *dependency inversion* between the core application and the aspects. These properties make AOP suitable for encapsulating crosscutting rule connections. This dissertation identifies commonalities and variabilities in the implementation of rule connection aspects and proposes abstracting these recurrent issues as elements of *aspect patterns*.

The second and third problems are addressed by building a layer of abstraction, a *domain model*, which allows for the expression of business rules in terms of domain concepts. This domain layer is able to represent domain concepts explicitly. A dedicated high-level business rule language is provided which enables the expression of high-level rules in terms of the domain concepts. Consequently, the coupling between the existing implementation of the core application and the rules is loosened. Moreover, the domain model is evolvable, which allows for the realization of unanticipated domain concepts and business rules that appear as a result of domain evolution.

This dissertation observes that, although the proposed aspects are a suitable solution to the problem of decoupling crosscutting rule connections, they exclude domain experts, as these aspects reside at the implementation level. Moreover, rule connection aspects need to take into account several recurrent issues, which renders the task of implementing these aspects difficult for developers. This dissertation supports the expression of rule connections at the domain level. A second dedicated high-level language is provided for this purpose.

The solution presented in this dissertation incorporates ideas from *Model-Driven Engineering (MDE)* in order to achieve the automatic generation of executable implementations for the high-level rules and rule connections. High-level rules and rule connections are automatically transformed to rule objects and rule connection aspects respectively.

The approach presented in this dissertation is evaluated in the domain of *Service-Oriented Architecture (SOA)*. Service-oriented applications are very volatile: new services appear, services become unavailable, non-functional properties of services vary (even at run-time), and applications need to cope with all these changes. Moreover, clients also change their requirements with respect to the selection and integration of services. This dissertation shows how high-level business rules can automate the customization of service-oriented applications. A Web services management layer, the WSML, is used as case study. Two scenarios are presented: an *evolution scenario*, which shows that it is possible to add new rules to the existing management framework, and a *refactoring scenario*, which shows that existing rules in the core WSML implementation can be refactored and externalized as high-level business rules.

# Samenvatting

Deze verhandeling behandelt de verbinding tussen hoogniveau, uitvoerbare business rules en bestaande objectgeoriënteerde applicaties.

Het huidige onderzoek rond de ontwikkeling van objectgeoriënteerde softwareapplicaties met regelgebaseerde kennis pleit ervoor om regels expliciet te maken, en te scheiden van de objectgeoriënteerde kernfunctionaliteit. Hoewel vele benaderingen dit doel beogen en een aanzienlijke verbetering vormen ten opzichte van de aanpak om regels als conditionele uitdrukking in te bouwen in objectgeoriënteerde applicaties, vertonen zij nog steeds drie belangrijke tekortkomingen.

Ten eerste, zelfs wanneer de business rules succesvol ontkoppeld worden, blijft de regelverbindingscode vermengd met, en verspreid over de implementatie van de kernfunctionaliteit van de applicatie. Wanneer bestaande business rules anders dienen te worden geïntegreerd, of wanneer nieuwe regels moeten worden verbonden aan niet-geanticipeerde gebeurtenissen, dient de broncode van de kernfunctionaliteit manueel op verscheidene plaatsen aangepast. Het wordt bijgevolg moeilijk om regelverbindingen te lokaliseren, toe te voegen, te wijzigen of te verwijderen.

Ten tweede blijven de uitvoerbare regels uitgedrukt op een laag niveau, onafhankelijk van de aanpak die gebruikt wordt om de business rules te ontkoppelen. Hierdoor zijn de regels niet begrijpbaar voor de domeindeskundigen die geen technische programmeerkennis hebben. Een derde, nauw verwant probleem is de sterke koppeling tussen de business rules en de bestaande implementatie van de kernapplicatie. Hierdoor worden de regels breekbaar en niet herbruikbaar, wat een niet-invasieve realisatie van regels die refereren naar niet-geanticipeerde implementatie-elementen verhindert. Bijgevolg kunnen business rules niet worden ingezet door domeindeskundigen zonder de tussenkomst van een ontwikkelaar.

Deze verhandeling introduceert een brede oplossing voor deze problemen die toelaat om business rules te integreren in bestaande applicaties op het domeinniveau. Het eerste probleem wordt aangepakt door de regelverbindingen in te kapselen in een afzonderlijke module, losgekoppeld van zowel de hoofdfunctionaliteit van de applicatie als van de regels. Deze ontkoppeling ligt niet voor de hand aangezien de regelverbindingen de hoofdfunctionaliteit van de applicatie als het ware doorsnijden (Eng. *crosscutting*). Aspectgeoriënteerd programmeren (AOP) biedt nieuwe modularisatietechnieken, zgn. aspecten, voor het inkapselen van *crosscutting* code terwijl ook de afhankelijkheid tussen de kernapplicatie en de aspecten wordt omgekeerd. Deze eigenschappen maken AOP geschikt voor het inkapselen van *crosscutting* regelverbindingen. Deze verhandeling identificeert gemeenschappelijke en veranderlijke factoren in de implementatie van regelverbindingsaspecten, en stelt voor om de terugkerende elementen te abstraheren als elementen van aspectpatronen.

Het tweede en derde probleem worden aangepakt door het opbouwen van een domeinmodel, d.i. een abstractielaag die toelaat om de business rules uit te drukken met behulp van domeinconcepten. Deze domeinlaag kan domeinconcepten expliciet voorstellen. Er wordt een specifieke, hoogniveau business rule-taal aangeboden die de uitdrukking van hoogniveau regels in termen van domeinconcepten mogelijk maakt. Daardoor wordt een minder sterke koppeling tussen de bestaande implementatie van de hoofdfunctionaliteit en de regels bekomen. Bovendien kan het domeinmodel verder evolueren, hetgeen de realisatie van niet-geanticipeerde domeinconcepten en business rules, die ontstaan als gevolg van domeinevolutie, mogelijk maakt.

In deze verhandelingen stellen we vast dat, hoewel de voorgestelde aspecten een gepaste

oplossing bieden voor het probleem van het ontkoppelen van *crosscutting* regelverbindingen, ze de domeindeskundigen uitsluiten, aangezien ze op implementatieniveau uitgedrukt worden. Bovendien dienen deze aspecten met verscheidende terugkerende elementen rekening te houden, wat het implementeren van deze aspecten bemoeilijkt voor ontwikkelaars. Deze verhandeling ondersteunt het uitdrukken van regelverbindingen op het domeinniveau met een tweede specifieke hoogniveau taal.

De oplossing die in deze verhandeling wordt voorgesteld integreert ideeën uit het domein van de modelgedreven softwareontwikkeling (MDE) om de automatische productie van uitvoerbare implementaties van hoogniveau regels en regelverbindingen te bewerkstelligen. Hoogniveau regels en regelverbindingen worden automatisch omgevormd tot respectievelijk regelobjecten en regelverbindingsaspecten.

De in deze verhandeling voorgestelde benadering wordt geëvalueerd binnen het domein van de dienstgeoriënteerde architecturen (SOA). Dienstgeoriënteerde applicaties zijn zeer veranderlijk: nieuwe diensten kunnen worden aangeboden, diensten kunnen onbeschikbaar raken, en de niet-functionele eigenschappen van diensten kunnen veranderen (ook tijdens de uitvoering); applicaties dienen al deze veranderingen aan te kunnen. Daarenboven veranderen cliënten hun vereisten met betrekking tot de selectie en integratie van diensten. Deze verhandeling toont hoe hoogniveau business rules het op maat aanpassen van dienstgeoriënteerde applicaties kunnen automatiseren. Een dienstgeoriënteerd framework voor *Web services* (genaamd WSML) wordt als case study gebruikt. Twee gevallen worden behandeld: een evolutiescenario toont dat het mogelijk is om nieuwe regels toe te voegen aan een bestaand management framework, en een refactoringscenario toont dat bestaande regels in de WSML-implementatie afgescheiden kunnen worden als hoogniveau business rules.

# Acknowledgements

It seems unbelievable to be writing the acknowledgments because it means I am finally finishing my PhD thesis!!!

First of all, I would like to thank my promoter Viviane Jonckers, who gave me the opportunity to pursue this degree by accepting me as a PhD student. I am grateful for her precise and constructive advice, and for all the time and effort she spent reading and re-reading many parts of my text. Her honest approach to guidance helped me stay focused and productive.

I would like to show my appreciation for my co-promoter and friend Maja D'Hondt. I found in Maja not only a great mentor and researcher, but also — most importantly — a wonderful human being. Her proactive and positive attitude inspired me during these long years of research and writing. Her presence was fundamental in keeping my motivation up and her advice was always very useful. Thank you also, Maja, for always making time to help me with the text, especially for having read and corrected chapter 6 more than 5 times!

I would like to express my gratitude to the members of my jury: Ana Moreira, Siobhán Clarke, Geert-Jan Houben, Wolfgang De Meuter and Wim Vanderperren.

My colleagues at SSEL have created such a friendly atmosphere in the office and have been very generous and helpful to me: Bruno De Fraine, Wim Vanderperren, Davy Suvée, Miro Casanova, Dennis Wagelaar, Bart Verheecke, Ragnhild Van Der Straeten, Mathieu Braem, Niels Joncheere and Dirk Deridder. During the first two years of my PhD I collaborated with Bart on the WSML. This project certainly taught me a lot, and for that I am grateful. I would also like to thank Wim and Davy for their great help on JAsCo. I particularly want to thank Bruno for helping me with latex and svn issues. Also, thank you to Bruno and Bart for helping me with the Dutch version of the abstract. Thanks to Dennis and Ragnhild for our discussions on MDE. In addition, thanks a lot to Wim, Dirk and Bart for proofreading my text and for giving me invaluable feedback. Thanks also to Fiona Coulter for reading my introduction and advising me on language.

Being so far away from home for so many years was not always easy. I would like to thank all my friends in Brussels, who helped me feel more at home and accompanied me during these years. Also thanks to them for keeping me entertained during endless milonga nights.

A very special thanks goes to my parents, Agustín Alejandro Cibrán and María Cristina Giménez, who always supported me with my projects and studies and were always present despite the physical distance. Thanks for all the love and care you gave me during all the years of my life, in any way (even over the phone!). I am extremely thankful to them! I also appreciated my brother's advice which was always very useful, so thank you to Federico Gustavo Cibrán!

Finally, I would like to thank Felix Zimmermann for putting up with me during these years. Felix always believed in me and was more convinced than me that I would successfully get to the end. Even and especially, during the last and stressful months of hard work, he still bore with me! So many thanks for that and for being so special to me!

# Contents

# List of Figures

# List of Tables

# List of Code Fragments

# List of High-Level Specifications

# Chapter 1

# Introduction

## 1.1 Problem Statement

In a competitive global business environment, software engineers and domain experts have an increasingly difficult role to play. The software applications they create must accommodate to complex technical concerns and changing business needs. A particular challenge is dealing with domain knowledge that is inherently volatile in real-world domains and businesses. Identifying this domain knowledge explicitly and managing it effectively is crucial. Unfortunately, however, these are complex tasks, as domain knowledge is usually not localized — rather, it is tangled and scattered in the implementation of software applications. The approach developed in this dissertation can help software engineers and domain experts accommodate changes in volatile domain knowledge — thus putting order into the new complexity of their tasks.

Domain knowledge refers to the concepts, and relations between concepts, which are inherent to a domain. It also refers to the constraints on those concepts and relations, and rules that state how to infer or 'calculate' new concepts and relations [SAA$^+$00]. In this dissertation we focus on this last part of the domain knowledge also known as *rule-based knowledge*. Rule-based knowledge can appear in different forms. Some applications have knowledge-intensive subtasks, such as (semi-)automatic scheduling, intelligent help desks and advanced support for configuring products and services. Other applications contain rule-based knowledge embodying business policies or *business rules*. The Business Rules Group defines a business rule as a *statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control the behaviour of the business* [BRG01]. In the e-commerce domain for instance, business rules typically guide customer preferences, discount strategies, return and refund policies, recommendations and so on. In the domain of healthcare, more sophisticated business rules are present as complex legislation rules regulating the payment of medical costs by patients, or in the financial business as for example international agreements on bank transfers. In this thesis we concentrate on this latter kind of rule-based knowledge. The first category of rule-based knowledge is considered in [D'H04] but is outside the scope of this thesis.

Besides the many rules that can be found in real-world domains, we observe that technical domains are also business rules driven. In this dissertation we are particularly interested in the domain of *Service-Oriented Architecture (SOA)* [PG03]. SOA advocates building applications by selecting and integrating third-party *Web services*. This is a very volatile application domain, as new services appear, services become unavailable, non-functional properties of services vary (even at run-time), and as a result client applications need to cope with all these changes. Moreover, client applications themselves change their requirements with respect to which service functionality needs to be selected and integrated. Therefore, we observe that service-oriented applications — and more specifically service selection, integration and management — are driven by service criteria based on dynamic non-functional service properties. These criteria are also examples of business rules that need to be considered and managed accordingly in order to achieve high flexibility in this kind of applications.

The applications considered in this dissertation typically provide a substantial *core application functionality* tackling the many technical concerns and supporting the users in their tasks. This core application functionality is usually developed and maintained using traditional software engineering techniques, such as object-oriented software development. We observe that when using current software-engineering methodologies and techniques, business rules are often implicit in the code, as *if* ⟨*condition*⟩ *then* ⟨*action*⟩ statements which are tangled and scattered with the core application functionality. This complicates the task of the application engineer, who has to have in mind the two aspects of the system, the core functionality and the business rules, and has to manually integrate them accordingly. Moreover, the presence of non-localized rules violates the principle of separation of concerns [Dij76a; HL95; Par72], which states that the core functionality has to be separated from other concerns of the system or aspects, in this case the rules. This lack of separation of concerns has a negative impact on the understandability, maintainability, reusability and evolvability of the whole application's code.

State-of-the-art research on developing object-oriented software applications with rule-based knowledge advocates making rules explicit and separate from the object-oriented core functionality [vH01; Ros03; Dat00]. Moreover, this decoupling is pursued throughout the whole software development process and as such, development of business rules also progresses from the discovery phase, to analysis, design and finally implementation, at all times keeping the rules separate from the core application functionality. These approaches claim that separating business rules is crucial in order to trace them to business policies and decisions, externalize them for a business audience, and evolve them, especially because they do not necessarily change at the same pace as the core application functionality [Ars01]. Decoupling the two aspects of the system, the core functionality and the business rules, helps reduce the dependencies between them which in turn improves overall understandability. Furthermore, the two aspects of the systems can be developed, maintained and evolved independently.

Many approaches exist that are targeted towards these goals and use radically different technical solutions. First of all, business rules can be represented separately in the object-oriented programming language itself. Specialized object-oriented design patterns — referred to as *Rule Object Patterns* [Ars01] — are proposed for representing rule-based knowledge explicitly and decoupled in object-oriented applications. Other approaches focus on externalizing explicit rules using XML, such as *Business Rule Beans* [RDR$^+$00]. Finally, there are approaches based on rule-based systems which provide dedicated language constructs for representing rules, and manage the flow of rules automatically. Among them, dozens of both commercial and academic so-called *hybrid systems* can be mentioned, which integrate a fully-fledged rule-based language with a state-of-the-art object-oriented programming language [ILO; YAS; FH03; JBob; Halb]. *Hybrid* in this context refers to the combination of the rule-based and object-oriented programming paradigms, as defined in [D'H04]. Independently of the implementation mechanism, business rules are executed at certain events, *i.e.* points in the execution of the core application functionality.

It is generally the case that once the initial application is developed, unanticipated business rules often appear as a result of changes in the business requirements. In this thesis, particular interest lies in the *non-invasive* adaptation of existing applications in order to cope with the appearance of unanticipated business rules.

We observe that, although existing approaches that support the separation of business rules from the core application considerably improve the more traditional object-oriented software development, they still suffer from several inherent problems:

**1. Rule connection code is crosscutting:** The first problem is in relation to the *connection* of the business rules with the existing core applications. By *rule connection code* or *rule connection* we refer to the code not only in charge of triggering the application of the rules at certain events, but also gathering the necessary information for their application and incorporating their results in the rest of the core application functionality. We observe that, even when the decoupling of the business rules is successfully achieved, the connection code is still tangled with and scattered in the core application functionality. This situation occurs independently of the concrete approach used for representing the rules. Therefore, when either the existing business rules change the way they need to be integrated in the existing application, or new business rules are added which need to be connected at unanticipated events of the core application, the source code of the core application must be adapted manually at different places. Consequently, it becomes difficult to localize, add, change or remove rule connections.

**2. Executable rules are low-level:** We observe that regardless of the approach taken to decouple business rules, executable rules are ultimately low-level: they are either implemented in a rule-based language — which is invariably a programming language — or expressed using design patterns or other technical solutions such as, for example, XML. Some rule-based systems provide support for the expression of rules at the domain level. However, high-level rules in these systems are not fully executable, as their deployment still requires the manipulation of low-level rule representations. Moreover, as these systems create overheads, they exclude certain applications. More lightweight approaches, on the other hand, fail to express executable business rules at the domain level. Therefore, executable business rules are not understandable to domain experts who are not adept at programming and do not necessarily have technical skills. As a consequence, the deployment of rules require the intervention of developers, who are in turn not experts in the domain. This creates a communicational gap between the two which reduces understandability and makes the software application error-prone.

**3. A tight coupling exists between executable rules and the implementation of the core application:** Executable rules are expressed in terms of concrete implementation elements from the existing core application. This creates a tight coupling between the business rules and the existing core application's implementation, causing the following three problems:

i Rules are *fragile*, as they become incorrect or invalid when the core implementation elements they refer to change.

ii Rules are not reusable among other applications of the same domain.

iii Unanticipated rules require manual extension of the existing code: new (and unanticipated) rules can appear that need to talk about concepts which are not present in the existing implementation of the core application. Thus, to be able to implement these new rules, code implementing the new concepts needs to be added manually to the existing implementation, which might result in scattering and tangling. Moreover, once again, the domain expert is excluded as these extensions require a good understanding of the existing implementation and therefore the need for having programming skills.

**4. Executable rule connections are low-level and complex:** Finally, connecting executable (and low-level) rules requires manually writing code which is tightly coupled with the existing implementation. Therefore, analogously to the problems identified for low-level rules, low-level rule connections exclude the domain expert, are fragile, cannot be reused and — when unanticipated — cannot be incorporated non-invasively in the existing application. Also, writing rule connection code is a complex task for the application developer, as this code needs to tackle many inherent and interrelated connection concerns (e.g. interrupt the application at the rule application time, make the required information available to the rule, trigger the rule, retrieve and incorporate the rule results, proceed with the core application, etc.).

## 1.2 Research Goals and Approach

The goal of this dissertation is to provide a solution for the problems identified in the previous section. We envisage a solution that enables existing applications to integrate business rules in a high-level way. This dissertation considers both business rules that are anticipated in some way in the implementation of the core functionality and business rules that are unanticipated in the existing implementation and that appear as a result of changes in the business requirements. The aim is to minimize the coupling between the two parts of the application, i.e. core functionality and business rules, contributing to their independent development, evolution and variability.

The problem of crosscutting rule connections can be addressed by encapsulating the code implementing the rule connections in separate modules, decoupled from both the core application functionality and the rules. This decoupling is not straightforward because rule connections are tangled and scattered in the core applications, and thus support for better separation of concerns is needed. The phenomenon of tangled and scattered code is known as *crosscutting* code in the area of *Aspect-Oriented Programming* (AOP) [KLM$^+$97]. AOP is an innovative approach that identifies the need for having new modularization mechanisms that enable the encapsulation of crosscutting code. Although AOP is usually employed for encapsulating implementation-level issues like logging and synchronization, the idea of domain knowledge as an aspect is introduced in [DMW99] and [DC02]. One of the advantages of AOP is that it introduces *dependency inversion* between the core application and the aspects: the core application does not invoke the aspects explicitly but instead the aspects actively 'observe' and react to certain events that occur during the core application's execution. This makes AOP suitable for encapsulating rule connections, as the aim is to avoid having to manually change the core application's code in many places with calls to business rules. Therefore, the first step is to investigate the suitability of AOP for encapsulating crosscutting business rule connections.

The second, third and fourth problems are addressed by building a layer of abstraction — a *domain model* — on top of the existing implementation, the low-level rules and the aspects encapsulating the rule connections.

In particular, the second problem is addressed by providing a high-level dedicated business rule language as part of the domain model. This high-level language provides abstractions for expressing business rules in domain terms and therefore it is easier to use and adopt by domain experts than a fully-fledged programming language.

In order to solve the third problem, abstractions are provided in the domain model for representing domain concepts explicitly. This allows to abstract over concrete implementation entities of the existing application. High-level rules are then expressed in terms of these domain concepts. Consequently, the coupling between the existing implementation and the rules is loosened (addressing problem 3-i), allowing rules to become reusable among applications that share the same domain abstractions (addressing problem 3-ii). Furthermore, this domain model is intended to be evolvable, i.e. to be able to incorporate unanticipated domain concepts that appear as a result of domain evolution (addressing problem 3-iii). These extensions are intended to occur at the domain level — without having to write new code manually — which enables the domain experts to also actively participate in the definition of these new domain concepts.

During the course of this research, it was found that aspects are a good solution to the first problem, but do not involve the domain expert as they reside completely at the implementation level. This implies the need for having programming skills — and more particularly AOP skills — to connect the rules with the core application, which again excludes the domain expert. Moreover, connection aspects need to take into account several recurrent issues, which complicates the task of the application engineer in charge of writing these aspects from scratch every time a new rule connection is needed. Thus, analogously to the rules, the abstraction of the rule connection aspects as higher-level entities, completely specified at the domain level, is pursued. This addresses the fourth problem.

We pursue our high-level rules and rule connections to be executable so that they can be directly integrated with the existing application. Thus, it is necessary to obtain executable implementations for the high-level rules and connections. However, in order for the domain experts to remain oblivious to the low-level implementations and to overcome the communicational gap that can exist between them and the developers, these implementations need to be generated automatically. *Model-Driven Engineering (MDE)* aims at building applications by defining models describing certain views of the software system at different levels of abstraction and specifying how those models map [MCF03]. This mapping is specified by *model transformations*. In MDE, model transformations specify how models are refined, evolved into a new version, or used to generate executable code. Following the MDE philosophy, we envision the definition of transformations in charge of generating code for the high-level rules and connections. This generated code must be encapsulated and well-modularized. Therefore, in the case of the crosscutting rule connections, aspects are generated (solution for problem 1), avoiding tangled and scattered code. This is important since MDE is considered for only a 'slice' of the software development, i.e. the business rules, and not the entire application. The rest of the application is developed and maintained separately and therefore changes induced by the high-level approach must be encapsulated.

The approach presented in this dissertation is advocated for certain kinds of existing OO applications. In the case of an application which has a stable core functionality (i.e. it does not change very frequently) that needs to be adapted or customized at different points according to certain circumstances, this approach is beneficial. The adaptations and customizations can be represented as business rules; business rule connections can encapsulate the link between the two parts of the application, core application and rules. Both parts can then change separately. This set up preserves the investment of building the core OO application as rules and their connections change.

A second kind of applications which can also benefit from this approach is characterized by a rich core functionality, in which a stable part and a set of pluggable functionalities can be identified. At a given point in time, the stable part can be combined with one or more pluggable parts, resulting in a complete instantiation of the application. However, the guidelines for selecting and plugging the extra functionalities are not fixed but can vary depending on the requirements. The approach presented in this dissertation can be used to represent these configuration guidelines explicitly as business rules.

Note that in the case of an application that has as main functionality a problem solving task (e.g. scheduling or diagnosis), a fully-fledged, rule-based system which relies on a reasoning engine is preferred [SAA$^{+}$00]. The core OO part in this kind of applications is typically an interface to the rule-based system, being in charge of gathering all the necessary data required for the reasoning process and triggering the activation of the rules at a few localized points. The use of our approach in this kind of applications does not present fundamental benefits as it does not contribute to the main challenge of building the rule set. In addition, in real-time systems (e.g. airplane control or life-support systems), failures cannot be tolerated and therefore the use of rules to adapt, customize, guide or configure functionality requires the utmost care. This is however a general problem for all rule-based systems and therefore not specific to this approach.

This dissertation envisions an approach that combines the advantages of AOP and MDE in order to realize the integration of high-level business rules into existing object-oriented applications. This leads us to the following hypothesis:

> *Aspect-oriented programming enables encapsulating and decoupling the rule connection code in between the business rules and the core object-oriented functionality. Model-driven engineering enables the expression of high-level business rules and connections that are also executable.*

The remainder of this chapter is going to present our approach which is divided in three parts. First, the challenges that appear when decoupling business rule connections from object-oriented applications are introduced, AOP is briefly presented, and its suitability for decoupling business rule connections in the form of aspects patterns is discussed (section

1.3). Second, the domain model is presented, consisting of domain concepts as well as high-level business rules and connections and the idea of using model transformations to link the two models, i.e. domain and implementation models, in a transparent and automatic way is introduced (section 1.4). Third, in this dissertation we analyze and show how the proposed high-level business rules approach — intended for any domain — can be used in particular for improving the flexibility of SOA. Our approach is evaluated in a non-trivial and technical domain as well as concrete results are shown using a complex web services management layer as a case study (section 1.5). Finally, the chapters of this dissertation are summarized and the contributions are listed at the end of this chapter.

## 1.3 Integrating Business Rules with Object-Oriented Applications using Aspect-Oriented Programming

This dissertation proposes using AOP for the encapsulation of the crosscutting rule connection code. We identify a number of issues that need to be taken into account in the connection of the business rules with object-oriented functionality in order to improve flexibility and configurability. These issues are independent of the concrete technology or approach used to implement the business rules, whether it is object-oriented patterns, externalized as XML, or implemented in a rule-based language.

In this dissertation, the focus is on the kind of business rules found in the applications of our industrial partners, as well as the ones presented in books on business rules [vH01; Ros03]. Examples of this kind of rules are price personalization discount rules typically present in the e-commerce domain and drug interference rules in the medical domain. In order to implement this kind of business rules, a lightweight approach can be taken that does not rely on the full power of a rule-based system. Therefore, in this dissertation a simple approach for implementing the business rules is chosen, namely the Rule-Object Pattern [Ars01], as the main challenges are imposed by the rule connections and not the rules themselves. Moreover, an approach based on a fully-fledged rule-based system might be overkill for existing applications with relatively simple business rules. In our approach the real challenges are posed by the connection of the rules — even relatively simple ones — with existing core applications.

In order to encapsulate the rule connections, a good first step is to analyze and experiment with existing general-purpose aspect-oriented approaches. Their ability to address the identified connection requirements is investigated. As a result of this research, it is observed that AOP features are suitable to accomplish the rule connection requirements. An overview of the use of AOP for encapsulating the rule connection is shown in Figure 1.1. The arrow group (1) depicts the interception of the core application execution at the places where the rule objects need to be executed, depicted as arrow group (2). The results of this first experiment are also presented in [Cib02; CDJ03; CDS$^+$03; CSD$^+$04; CDS$^+$05].

### 1.3.1 Aspect-Oriented Programming

A software application involves many heterogeneous concerns. Concerns are properties or areas of interest in a system. Typically concerns can range from high-level notions like security and quality-of-service to low-level notions such as caching, buffering, synchronization and transaction management [EFB01]. In order to deal with all these heterogeneous concerns in a software application, *separation of concerns* (SoC) is fundamental. SoC refers

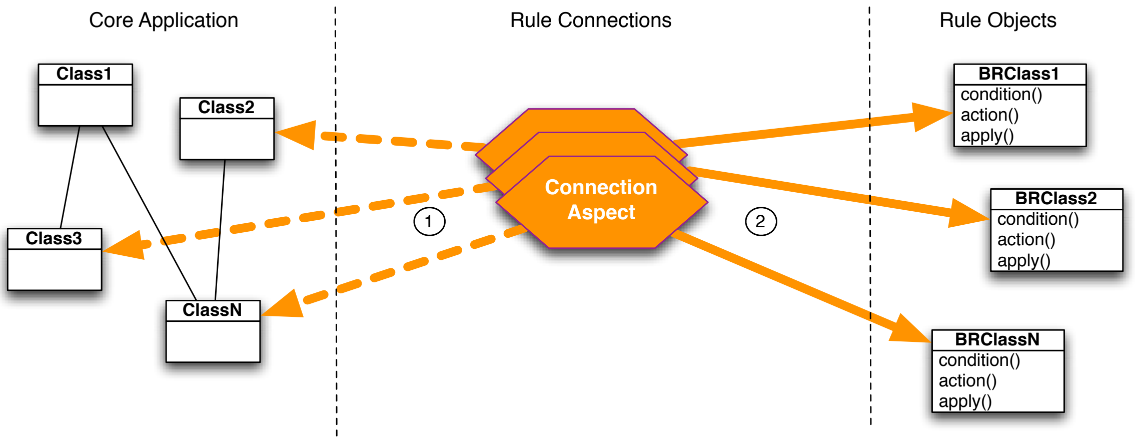


*Figure 1.1:* Overview of our approach for integrating business rules with AOP

to the ability to identify, encapsulate, and manipulate only those parts of the software that are relevant to a particular concept, goal or purpose [Dij76b; OT01]. SoC is a crucial property for realizing comprehensible and maintainable software. It aims at being able to think about the design and implementation of a system in natural units of concerns rather than in units imposed by specific languages or tools. 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 of thinking to the limitations imposed by the languages and tools [KHH$^+$01].

Once software systems reach a certain complexity, the modularization constructs provided by current languages and environments fall short in separating all these heterogeneous — and even sometimes interrelated — complex concerns. *Aspect-Oriented Programming* argues that some concerns of a system cannot be cleanly modularized using current software engineering techniques. This is because these 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 [KLM$^+$97]. This problem is identified in [TOHS99] as the "*tyranny of the dominant decomposition*": the program can be modularized in only one way at a time, and as a consequence the many concerns that do not align with that modularization end up *scattered* across many modules and even *tangled* with code that addresses other concerns. Examples of typical decompositions are objects in the object-oriented paradigm, modules in the imperative paradigm, and rules in the rule-based paradigm. In this thesis the concentration is on the object-oriented paradigm as a decomposition mechanism.

The tangled and scattered concerns of a system are referred to as *'system-wide' concerns* as they do not nicely fit into the chosen modularization of the system. Therefore they *crosscut* its main decomposition. Because crosscutting concerns are not encapsulated, it is difficult to add, edit or remove them. Typical examples of well-known crosscutting concerns in object-oriented applications are debugging concerns such as logging and contract verification, security concerns such as confidentiality, access control and transactions as well as verification of design or architectural constraints, systemic properties and features.

*Aspect-Oriented Programming* aims at achieving a better separation of crosscutting concerns in object-oriented software applications. To this end, AOP introduces a separate module — called *aspect* — which is able to encapsulate the implementation of a crosscutting concern. Moreover, AOP allows the description of the relationships that exist between the different concerns of a system and the mechanisms to *weave* or *compose* them together into a coherent program. Originally, separation of concerns was oriented towards implementation concerns. More recently, AOP (aspects at the implementation level) is not the only area of discussion but additionally, debate surrounds Aspect-Oriented Software Development (AOSD), as the community recognizes the need for separation of concerns throughout the whole software development cycle. This is because crosscutting concerns may arise at any stage of the software development life-cycle, including requirements specification, analysis, design, implementation, debugging, etc.

At the implementation level, AOP introduces a new module called *aspect* that is able to modularize crosscutting concerns. Typically an aspect consists of *pointcut* and *advice* definitions. A pointcut identifies a set of points in the program's execution where an aspect can be applied. Each of these points is called a *joinpoint*. Thus, a pointcut specification is a concise description of a set of joinpoints where the aspect should be applied. An advice specifies a concrete behavior to be executed at a certain pointcut, typically before, after or around the original behavior identified by the joinpoints. The additional logic defined in a before advice or an after advice has to be executed before or after the original behavior respectively. An around advice replaces the original behavior but is still able to invoke it if necessary. The advice language typically consists of the host language augmented with a limited number of special keywords that offer aspectual reflection and control over the execution of the original joinpoint. In order to apply the advices at the joinpoints specified in the aspect's declared pointcuts, the aspect needs to be weaved with the base application. Traditionally, weaving takes place at compile time, which means that the advices are inserted into the target application at the source or byte-code level, however more flexible approaches allow weaving to occur at runtime.

Although AOP is a rather recent paradigm, numerous aspect-oriented approaches have already been proposed for which advanced tool support has been developed. These include AspectJ [KHH$^+$01], Adaptive Programming [LOO01], Composition Filters [BA01], JBoss/AOP [FR03], Spring/AOP [J$^+$], AspectWerkz [BV] — which has recently merged forces with AspectJ — and JAsCo [SVJ03]. Some of these approaches are currently reaching maturity and are being used also in industrial projects, e.g. AspectJ, JBoss/AOP and Spring/AOP. Another representative approach, radically different to the previous ones, is Hyper/J [OT01].

### 1.3.2 Requirements

This dissertation identifies and analyses typical situations that occur when connecting rules to the core application and observes that traditional approaches fail to solve them without inducing invasive changes to the existing code. In order to achieve highly-flexible and configurable business rules, a suitable approach must be able not only to encapsulate tangled and scattered code, but also to accomplish a set of requirements that we identify [Cib02; CDJ03; CDS$^+$03; CSD$^+$04; CDS$^+$05], namely the ability to:

1. connect business rules to core application events which depend on run-time properties

2. expose information available at dynamic events and pass it to the business rules applicable at those events

3. provide means for capturing extra information also needed for the application of the business rules but that is not available at the dynamic events that denote the rule connection time

4. enable the introduction of unanticipated information required by the business rules

5. configure and reuse existing business rules at different dynamic events; analogously, configure and reuse existing dynamic events with different rules

6. combine, prioritize and exclude business rules when they interfere with one another

7. control the instantiation and initialization of business rule connections

8. and preferably accomplish all the above dynamically — without interrupting the application's execution — and non-invasively — without changing the core application's code.

### 1.3.3 AOP for Decoupling Business Rule Connections

Several experiments are carried out as part of this dissertation with different AOP approaches which let us analyze their suitability with respect to the identified requirements. We show in [Cib02; CDJ03] and [CDS$^+$03; CDS$^+$05] how *AspectJ* and *JAsCo*, respectively, deal with the integration issues. The first experiment considers AspectJ [KHH$^+$01], one of the most mature and well-known AOP approaches. The main advantage of AspectJ is its expressiveness with respect to describing and manipulating events of the core application. This feature allows addressing the first three requirements successfully. However, AspectJ only allows the static pluggability of aspects and thus it does not accomplish the dynamic configurability requirement. A second experiment uses JAsCo [SVJ03], a dynamic aspect-oriented approach which aims at integrating AOP ideas into Component-Based Software Development (CBSD). JAsCo can be considered as an AspectJ-like approach, as they both have a similar join-point model and structure aspects in a similar manner. However JAsCo aspects are more reusable as all deployment details are encapsulated in a separate module called *connector*. This feature enables a more fine-grained control over the instantiation and initialization of the business rules. Moreover, explicit combination strategies can be defined in connectors which enables the specification of more advanced and fine-grained business rule combinations. Because JAsCo allows the dynamic pluggability of aspects, rules and rule connections can be instantiated at run-time to fit the application at hand.

Other state-of-the-art AOP approaches — namely JBoss/AOP, JAC, HyperJ and AspectWerkz — were investigated as well and analyzed against the requirements [CSD$^+$04].

These experiments let us observe that, even though there exist radical differences between the current state-of-the-art AOP approaches — and therefore the identified requirements were partially accomplished by them — AOP features are suitable to encapsulate the business rule connections.

### 1.3.4 Distilling Aspect Patterns

The previous experiments have also shown that the aspects that encapsulate the rule connections are built up of the same elements that vary with certain situations [Cib02; CDJ03; CDS$^+$03; CDS$^+$05; CD06a]: rule application time, rule activation time, rule data manipulation (passing and retrieving information to and from the rule, respectively) and rule triggering. As such, we propose abstracting these recurrent issues in *aspect patterns* for implementing different kinds of business rule connections. These patterns capture commonalities and variabilities in the implementation of the connection aspects. Figure 1.2 depicts the recurrent connection elements that are part of a connection aspect.

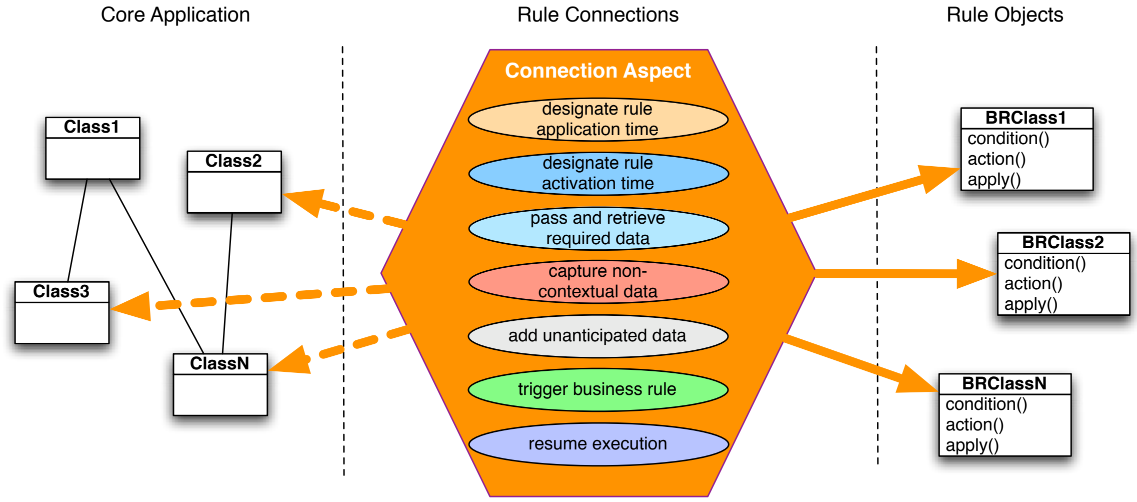


*Figure 1.2:* Recurrent elements in a rule connection aspect

## 1.4 Expressing Executable Business Rules at the Domain Level using Model-Driven Engineering

Understandability is one of the fundamental goals of software engineering. The definition of understandability depends on the intended audience: management, domain expert, developer, or user. In this dissertation the intended audience are domain experts and developers. The domain expert considered in this dissertation is an individual who is both experienced and knowledgeable about a particular problem domain or area of interest as well as he or she is knowledgeable about domain analysis and design techniques. Typically this domain expert has some understanding of software systems without requiring a deep technical background or programming skills. This definition is in accordance to what Neighbors defines as *domain analyst* in [Nei84]. In that work a domain analyst is the person responsible for *"conducting domain analysis, understanding the domain of application, and even performing some system analysis as well as communicating with the players in each of these areas"*.

With this audience in mind, understandability is defined as the property that results from hiding technical complexity. This definition is strongly related to the principle of abstraction, which can in turn be defined as the process of moving to a higher level, extracting essential properties while omitting inessential details. In [RGI75], Ross et al recognize the close relation between these two software engineering properties, motivating how increasing abstraction helps improving understandability. The authors state that *"abstractions employed to achieve the goal of understandability mean that each level of abstraction, while presenting more and more detailed views of the system, must do so in terms which are understandable to the intended audience"*.

In order to involve the domain expert in the process of defining rules and their connection and to simplify this process for the developer, we propose a *high-level domain model* which consists of three parts: domain entities, business rules about domain entities, and connections of business rules to the core application in terms of domain entities. A mapping between the domain entities, representing domain concepts, and the existing implementation must be provided. Model transformations are used to translate high-level rules and connections to executable implementations using the mapping for the domain entities. An overview of the domain model is depicted in Figure 1.3 and explained in the coming subsections. The proposed domain model is also presented in [CD05; CDJ05; CDJ06a; CD06c; CD06a].

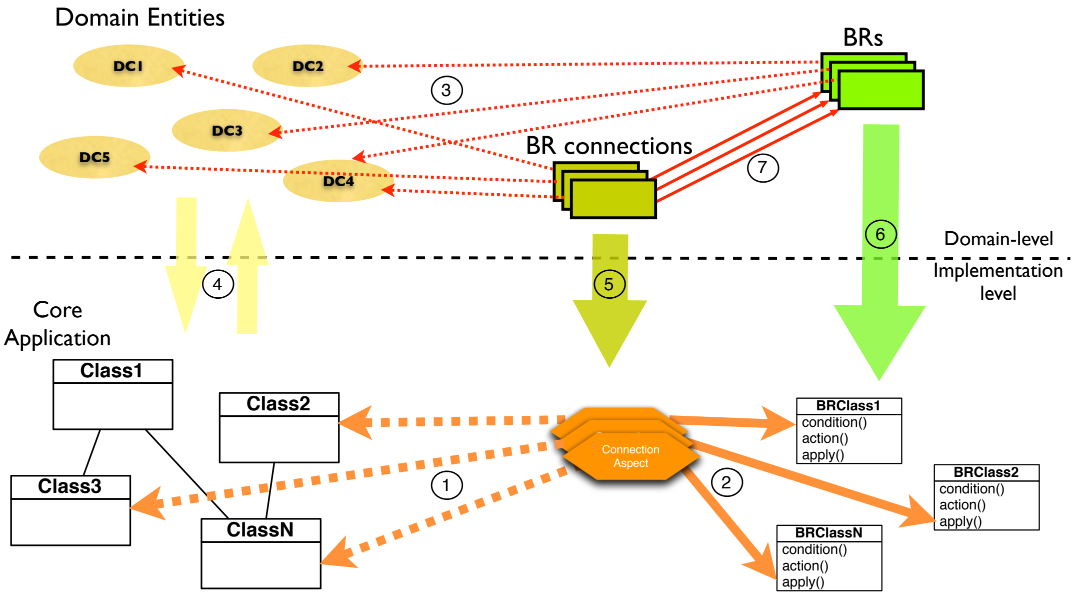


*Figure 1.3:* Overview of our approach for high-level business rules

The domain model approach presented in this dissertation improves understandability because it increases the level of abstraction for expressing business rules and rule connections and, most importantly, only abstracts those features that are of interest to the target audience. On the one hand, the domain expert is not aware of the technical complexity: he or she is able to express business rules in terms of the domain, without having to be aware of how those rules and their connections as well as the core application are actually implemented. Moreover, as the proposed dedicated languages are simpler than fully-fledged programming languages — they offer simpler features than the ones found in programming languages —, so that the domain expert is able to adopt them more easily. On the other hand, the task of the developer is simplified, as he or she can now concentrate on the actual problem of connecting business rules, without having to be an expert in the underlying technologies, i.e. OOP and AOP.

### 1.4.1 Model-Driven Engineering

*Model-driven engineering* is an approach to software engineering that aims to raise the level of abstraction and to develop and evolve complex software systems by means of manipulating models. Therefore, models are its primary assets. A model describes a certain view of the software system at a certain level of abstraction. For example in a bank application, different models can be defined to represent customer management, transaction management and account management. Models can be specified at different levels of abstraction and sometimes also in different languages. The ultimate goal of MDE is to have a software development environment at our disposal with off-the-shelf models and mapping functions that *transform* one model into another. Mellor et al. [MCF03] state that *"Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing".* Adopting this idea, this dissertation proposes a *model* that can help the domain expert understand and express the business rules and connections; *the real thing* is then having 'executable' implementations for the business rules and connections that can be directly integrated in existing — and even running — applications. The manipulation of models is achieved by means of model transformation, which is considered to be the heart and soul of model-driven engineering [SK03]. A model transformation can encode a refinement step, an evolution step, and even a code generation step. Moreover, a model transformation can take one or multiple source models and produce one or multiple target models. Furthermore, transformations can be classified into *horizontal* and *vertical.* In the former case, source and target models are at the same level of abstraction whereas in the latter, models reside at different levels of abstraction. In this work we use vertical transformations. More detailed information on model transformations and their classification can be found in [SK03; MCG05].

### 1.4.2 Requirements

In the pursuit of our *domain model*, certain properties are desirable: *high-level*, meaning that no details are exposed about the core application's implementation which this model is built upon; *declarative*, meaning that interest lies in *what* the high-level model specifies and not *how* it is implemented. This model allows expressing rule logic in terms of concepts of the real-world domain instead of entities from the existing implementation. Domain concepts are captured in domain entities. These domain entities are used in the definition of the rules and rule connections. The high-level nature of this domain model allows reusing domain knowledge among different applications of the same domain or among different versions of an evolving application.

### 1.4.3 Domain Entities

The *domain entities* represent the vocabulary of the domain of interest. They are the building-blocks used in the definition of the high-level rules and their connections with the core application. They are abstractions of domain knowledge that is either implicitly represented in the core application's implementation or unanticipated. The *high-level business rules* express relations between terms of the domain that are captured as domain entities. Thus, rules are independent of implementation details. The *high-level business rule connections* specify how the rules are integrated with the core application and typically do so by denoting *events* — also specified as high-level entities — at which the rules need to be applied as well as specifying how the available information matches the information expected by the rules. Domain entities are illustrated by the circles and the relations between them located in the upper-left part of Figure 1.3.

An essential step for achieving executable rules from high-level specifications is the definition of how the domain entities involved in those specifications are mapped to the implementation. In this dissertation we present an approach for making the *mapping* between the domain and the implementation models explicit (depicted by the arrows labeled with (4) on the left part of Figure 1.3). The definition of domain entities can follow two different flavors: *top-down* and *bottom-up*. In both cases the existence of a core OO application is assumed. The distinction between these two flavors lies in the starting point for the identification of the domain entities of interest:

In the *bottom-up* view, the starting point is the *solution domain*. The domain concepts of interest are identified in the existing solution and pulled up to the domain level. During this process, some details that are important in the solution might not be meaningful to be represented at the domain level and therefore might be hidden or adapted. Typically this pulling up process is carried out by a developer or someone knowledgeable about the existing implementation solution, in cooperation with a domain expert who guides the developer in identifying the entities of interest. Once this first pulling up phase has been accomplished, new domain entities can be added which do not necessarily correspond to identifiable existing implementation entities. This second phase in the definition of domain entities can be carried out completely at the domain level, by combining other already defined domain entities. Thus, the domain expert could perform this step without the intervention of a developer. In addition, for domain entities that are completely unanticipated in the existing implementation, new implementation entities are generated automatically. Thus ideas from MDE are applied for this automatic generation.

In the *top-down* view, the starting point is the *problem domain*. The first step consists of defining domain entities of interest from the point of view of the problem domain. This step can be carried out by the problem domain modeler or domain expert. Because the domain expert is not knowledgeable about what is represented in the existing implemented solution, he or she is not biased by the entities that already exist in the implementation. Secondly, once the problem domain entities have been identified, they have to be mapped to the implementation. The process of defining a mapping between a high-level problem domain model and a low-level solution domain implementation will have to involve a collaboration between the problem domain modeler (i.e. the domain expert) and the solution implementor (i.e. the developer). Moreover, as in this scenario both models are defined in isolation, the discrepancies between them might be numerous. Most likely, the domain entities would not necessarily correspond in a one-to-one way to existing implementation entities. This motivates the need for having a powerful mapping language which allows to overcome these discrepancies. Our approach can deal with this scenario as well by providing a powerful mapping language which allows for the definition of more complex mappings. Again, some of these mappings still require the intervention of developers whereas others can be left to the domain expert.

Other business rule approaches exist today which advocate the idea of mapping domain concepts (used in the expression of the high-level rules) to implementation [ILO; YAS;

Inn; JBob; Halb]. However, in these approaches, the domain concepts are simple aliases for implementation entities and thus a one-to-one mapping between them is assumed. As a consequence, the domain models supported by these approaches are tightly coupled with the implementation models. Moreover, when domain concepts have more complex realizations at the implementation level, the need for supporting more complex mappings than the one-to-one mappings arises.

This dissertation proposes and implements a dedicated mapping language that enhances the current domain mapping support found in existing approaches in three innovative directions:

- Domain entities can map to more than one implementation entity.
- Domain entities can explicitly represent derived information.
- Domain entities can be completely unanticipated in the existing implementation.

AOP is used in a transparent way for the realization of some of the mappings in these directions. This dissertation also shows how the proposed mapping language can be used to realize several mapping use cases, for example mappings to many entities, mappings that require calculating values at execution points, anticipated and unanticipated mappings.

Although some mappings still require knowledge about the implementation, others can be completely defined at the high level, in terms of other existing domain entities. In our prototype implementation, mappings expressed in the dedicated mapping language are fully and automatically translated into expressions that only involve implementation entities — either in OOP or AOP. These expressions are used in the rule and rule connection implementations (explained in section 1.4.5). Moreover, using MDE ideas, for some high-level mapping specifications, new implementation entities (aspects) are generated for the realization of the mapping. The mapping ideas described in this dissertation are also presented in [CDJ05; CDJ06a].

### 1.4.4 High-Level Business Rules and Connections

This dissertation proposes the expression of business rules in terms of domain concepts captured as domain entities of a domain model. The idea of expressing rules at a higher-level of abstraction is not new and therefore present in other existing approaches (e.g. JRules [ILO], QuickRules [YAS], VisualRules [Inn], JBoss Rules [JBob] and HaleyRules [Halb]). The same way rules are specified in current approaches, in our approach a high-level rule is defined as an *IF* ⟨*condition*⟩ *THEN* ⟨*action*⟩ statement, meaning that the condition has to be satisfied in order for the action to be performed. The ⟨condition⟩ and ⟨action⟩ parts are expressed in terms of domain entities. This dissertation proposes and implements a dedicated language which allows the expression of such high-level business rules.

We observe that current approaches that allow the expression of business rules in terms of elements of a high-level business or domain model only support anticipated one-to-one mappings from high-level business rules entities to existing implementation entities. As a consequence, a tight coupling exists between the business or domain model and the implementation model. This is a problem since a high-level — and executable — specification of business rules can be discrepant from the implementation of the core application functionality. This is due to business rules not always being anticipated in the original core application. Thus, one-to-one mappings are not enough to realize unanticipated business rules. The real challenge is the realization of unanticipated business rules that requires unanticipated domain vocabulary that is not present in the existing implementation. We tackle this by supporting more sophisticated mappings for the domain entities involved in the high-level rules, as presented in Section 5.2.

Analogously to the high-level business rule language, a second dedicated language is proposed and implemented that allows the expression of rule connections at the domain level. Therefore, rule connections are also separate and explicit entities at the domain level. This language offers features that abstract from the recurrent connection issues captured by the aspect patterns. Moreover, it supports a set of variations for each of these rule connection issues. High-level rules and connections are depicted in the upper-right part of Figure 1.3. The dotted arrows numbered (3) depict the definition of rules and rule connections in terms of domain concepts made explicit as domain entities. Arrow (7) indicates the relation that exists between a high-level business rule connection and the high-level rule it connects. The high-level rule and rule connection dedicated languages are also described in [CDJ05; CDJ06a; CD06c; CD06a].

### 1.4.5 Automatic Transformations

In order to make high-level rules executable, i.e. ready to be integrated with the existing application according to their connections, we follow a *Model-Driven Engineering* approach: the rules and connections are automatically translated to classes and aspects, respectively. These transformations are illustrated in Figure 1.3 by the arrows (5) and (6). A different transformation is proposed per different high-level feature. However, extra challenges appear when translating a complete high-level rule connection specification which combines many of these features: the output is not the result of simply concatenating the outputs of the individual transformations for the involved features. On the contrary, the individual outputs have to be combined in a non-trivial way in order to obtain a correct aspect. This makes the transformation process more complex since these dependencies between the individual transformations need to be taken into account.

Moreover, the transformations explore the mappings of the domain entities involved in order to get an expression only in terms of implementation entities. This implementation is included in the generated code of the aspects and classes. In case of one-to-one mappings — directly pointing to implementation entities — this step is simple, as only existing implementation entities need to be retrieved. However, this process becomes more complex as nested mappings need to be explored.

Our approach maintains separation of concerns at both levels, the domain and the implementation levels, thus facilitating traceability. Moreover, the automatically generated code pertaining to rules and connections remains separated from the existing application code and therefore does not interfere with the development and maintenance of the existing application. The proposed transformations and the challenges of their implementation are also presented in [CD06c; CD06a].

## 1.5 Business Rules in Service-Oriented Applications

In service-oriented computing, applications are often created by integrating third-party Web services. However, in order for client applications to achieve a high flexibility in this integration, advanced support for selection and client-side service management is fundamental. This support is rarely provided in standard state-of-the-art service integration approaches and tools. Moreover, we observe that the selection, integration and management of Web services are driven by criteria based on non-functional service properties and QoS. For instance, the service integration can be guided by rules that prefer fast and reliable services or give priority to services with the least number of failures. Other rules govern the way management should be carried out, e.g. advising the activation of a caching mechanism for services that are too slow. Many of these business rules depend on dynamic service properties that are only known at run time. Our objective is to achieve the decoupling of service criteria that guides the service selection, integration and client-side management as explicit business rules. We think that their explicit specification is crucial to achieve a highly flexible and adaptable integration of services that best fit the client application's needs.

### 1.5.1 Web Services Management Layer (WSML)

As a first step towards achieving a flexible selection, integration and management of web services, we developed a flexible layer in between the services and the client applications. This layer is named *Web Service Management Layer* (WSML). The WSML is an AOP-based management framework that provides support for the dynamic selection and integration of services into client-applications and the client-side service management. It offers a reusable library of selection, management and monitoring concerns — implemented as aspects — that can be customized for different client applications. Additionally, it supports the explicit definition of service criteria based on non-functional properties that govern the selection, integration and management of services. This work is presented in [CVV$^+$07; VCV$^+$04; VCJ04].

### 1.5.2 High-level Business Rules in the WSML

As a second step, we leverage the advantages of this mediation framework by decoupling WSML configuration and customization business rules, and expressing them at the domain level: first of all, many decisions about how the framework must be configured and customized are taken either at deployment time, i.e. at the moment the WSML framework is deployed on a concrete client application, or manually at run-time, i.e. through an interface that requires human interaction. Examples of these decisions are: choosing which aspects need to be plugged in, which parameters need to be used for their configuration and which services are to be composed. Moreover, even though some anticipated selection and management decisions are encapsulated in business rules, they are tangled and scattered in the implementation of the framework, with negative effects on maintainability. Furthermore, adding new unanticipated rules implies manually modifying or adding code to the framework at many places, which is not desirable. Therefore, extra support is needed to — automatically and non-invasively — realize dynamic business rules that can vary at run time and that are unanticipated at deployment time. We then propose using high-level business rules to express and enforce the dynamic business rules that guide the configuration and customization of the WSML. The approach presented in this dissertation is evaluated with two scenarios: i) an *evolution* scenario, which shows that it is possible to add new rules to the existing management framework, and ii) a *refactoring* scenario, which shows that existing rules in the core WSML implementation can be refactored and externalized as high-level business rules. This evaluation is also reported in [CDJ05; CDJ06a; CD06b].

## 1.6 Chapter Summaries

**Chapter 2: Connecting Decoupled Business Rules with Object-Oriented Applications** This chapter introduces the general concepts behind business rules and the considerations that arise when decoupling them from object-oriented applications. A simple example application domain is introduced which is used throughout this dissertation. As the goal of this dissertation is to integrate business rules in existing applications, having to introduce the extra overhead induced by the use of a dedicated rule-based technology might be overkill in some cases. Thus, we take a lightweight approach for implementing decoupled rules which uses standard object-oriented software development, namely the *Rule-Object Pattern*. We present how this pattern can be used in combination with other design patterns to modularize the rule connection. We then identify and discuss fundamental connection issues that are not tackled by this approach. We then identify a set of requirements, which we believe are essential in order to successfully encapsulate the rule integration code. These requirements are independent of a specific object-oriented programming language in which the core application functionality is implemented, and independent of the business rule representation used.

**Chapter 3: Aspect-Oriented Programming for Business Rule Connection** This chapter focuses on demonstrating the suitability of Aspect-Oriented Programming for realizing the technological requirements identified chapter 2. We first introduce the main ideas advocated by AOP as well as give a general overview of some state-of-the-art AOP approaches. We then describe the AOP characteristics that we consider are fundamental for achieving the modularization of rule connection code. Two representative approaches are then selected that adhere to the chosen characteristics, AspectJ and JAsCo. Concrete examples in these approaches are presented along the chapter which show how the encapsulation of crosscutting rule connections can be successfully accomplished.

**Chapter 4: Aspect Patterns for Business Rule Connection** The experiments described in chapter 3 let us observe that an AOP-based solution for the modularization of a rule connection typically involves a set of recurrent connection issues. Moreover, these issues vary in specific circumstances. In this chapter we identify and discuss these issues that we call *rule connection elements*. Moreover, we make the distinction between elements that are mandatory — i.e. need to be part of every rule connection — or optional — i.e. might or might not be part be of a rule connection. Furthermore, we identify how and under which circumstances these elements vary and analyze which AOP features are suitable to implement each of these variations. We also observe that not all the variations of different elements can be always combined as for instance the choice of a certain variation for one element can restrict the set of possible variations for another element. As a result, this analysis lets us distill a set of *aspect patterns* that can serve as guidelines for the implementation of rule connection aspects. These patterns rely on AOP characteristics which are common to all approaches based on the pointcut-advice model. In this chapter JAsCo is employed for illustration purposes.

**Chapter 5: A Domain Model for Domain Entities, High-Level Business Rules and High-Level Business Rule Connections** This chapter is concerned with one of the ultimate goals of this dissertation, which is the consideration of the domain expert as an active participant in the process of understanding, defining business rules and integrating them with the existing application. This chapter proposes building a *high-level domain model* which incorporates ideas from MDE to achieve the integration of high-level and executable business rules in existing applications. First of all, the ideas behind MDE are presented. Secondly, we show how domain concepts can be explicitly captured and how business rules and their connections to an existing core application can be defined in terms of those explicit domain concepts. Thirdly, we present transformations that encapsulate the translation from high-level rules and connections to implementation. High-level rules are transformed into rule-objects whereas rule connections are transformed into aspects (following the patterns defined in chapter 4). These transformations are carried out automatically and transparently by our prototype implementation.

**Chapter 6: Mapping Domain Knowledge To Implementation** An essential step for achieving the automatic generation of executable rules from high-level specifications is the definition of how the domain entities involved in those specifications are mapped to the implementation. This chapter presents an approach for making the *mapping* between the domain and the implementation models explicit. We build on the current support provided by existing approaches and enhance it in many directions. This chapter first motivates the need for having more complex mappings. It then presents the main features of the proposed and implemented mapping language. Finally it shows how this mapping language can be used to realize five different mapping use cases.

**Chapter 7: Implementation** This chapter presents the prototype implementation developed as a proof of concept for the ideas presented in this dissertation. This prototype supports the entire domain model explained in chapters 5 and 6. The core of this implementation is a framework of OO classes for representing business rules, business rule connections, domain entities and their mappings to implementation. Moreover, on top of this core framework, three high-level dedicated languages are implemented: one for the definition of high-level business rules, a second one supporting the definition of high-level business rule connections and a third one for the definition of domain entities and their mappings. Parsers for these languages have been implemented. Semantical checks in charge of validating high-level specifications against the domain entities defined in the domain model are also supported. High-level specifications expressed in the dedicated rule and connection languages are automatically translated to OOP and AOP programs respectively, following the transformations described in chapter 5. Examples taken from the e-commerce case study application are shown throughout this chapter.

**Chapter 8: Evaluation** In this chapter an evaluation of our approach is presented which uses a case study in the domain of Service-Oriented Architectures (SOA), the Web-Services Management Layer (WSML). Unlike the real-world domains (e.g. financial, medical) typically found in state-of-the-art business rules systems, the chosen case study is based on a technical and challenging application domain, which let us show the expressive power of our approach. The same way as real-world domains, this domain suffers from the problems posed by the management of business rules and therefore can benefit from our approach. Many rules need to be taken into account in order to cope with the inherent volatility of service-oriented applications. We particularly focus on QoS criteria that guide the selection and client-side management of Web services. This chapter first introduces the main ideas behind the WSML and describes its general architecture and its approach to service selection, management and redirection. It then identifies current limitations of this layer with respect to changing and adding new configuration business rules based on QoS. The actual evaluation part is done in two scenarios: *evolution* and *refactoring* scenarios, where we show, on the one hand, how our approach can realize the non-invasive addition of unanticipated business rules to the WSML and, on the other hand, how existing selection policies can be refactored and expressed at the high-level.

**Chapter 9: Related Work** This chapter analyzes different approaches that relate, in one way or another, to the work presented in this dissertation. First, several (some commercial) business rules systems are described with respect to their business rule languages, their support for expressing business rules at the domain model and the rule execution model they support. Then, lightweight approaches to business rules as well as other approaches that also investigate the use of AOP for the decoupling of business rules are described. Afterwards, approaches that aim at combining MDE and AOP are presented. We also touched upon approaches that study the mapping between several knowledge representation mechanisms. Some related approaches that also aim at externalizing business rules in the domains considered in this dissertation, i.e. the e-commerce and service-oriented domains are described as well. To conclude, some work on business rule methodologies, vocabularies and rule engine standards is presented.

**Chapter 10: Conclusions** This chapter presents the conclusions of this dissertation. It first summarizes the work presented in this dissertation while stressing our contributions. This dissertation then ends with a discussion on trade-offs and future work.

# Chapter 2

# Connecting Decoupled Business Rules with Object-Oriented Applications

This chapter aims to introduce the general concepts behind business rules and the considerations that arise when decoupling them from object-oriented applications. We first introduce a simple example application domain which is used throughout this dissertation for illustration purposes and give some examples of typical business rules in it (section 2.1). In this dissertation we take a lightweight approach for implementing decoupled rules. As our goal is to integrate business rules in existing applications, having to introduce the extra overhead induced by the use of a dedicated rule-based technology might be overkill in some cases. Therefore, this chapter presents an approach for the decoupling of business rules which uses standard object-oriented software development, namely the *Rule-Object Pattern* (section 2.5). Afterwards, we present a typical approach for connecting rule objects which is based on the use of design patterns (section 2.7). However, we identify many fundamental connection issues that are not tackled by this approach. We analyze these connection issues and come up with a set of connection requirements which are presented and discussed at the end of this chapter (section 2.8).

## 2.1 Running Example: e-commerce

The World Wide Web has become the standard computing platform for the development of next-generation information systems. A new wave of Web-based applications such as corporate portals, supply chain automation, and online marketplaces, is driving the need for a more open, flexible, adaptable, and distributed infrastructure. Besides having to deal with the many technical and complex issues inherent to the Web, in order for these applications to stay competitive, they need to take into account the many heterogeneous needs of their users and adapt accordingly. In this setting, Web applications are becoming increasingly complex.

Examples of this kind of systems are e-commerce applications. These applications need to keep up with an increasing complexity taking into account important issues such as user profiles and personalization concerns among others. In this dissertation — similarly to other existing approaches that also advocate the decoupling of business rules from core applications [RDR$^+$00; RFCS01; RSG01a; KRS00; IBM] — we choose the e-commerce as a representative domain to illustrate our approach. In the next sections we present the basic functionality of a simple e-commerce application and present typical business rules that we encounter in this domain.

### 2.1.1 Basic functionality

This simple e-commerce application allows customers to order and buy products online. The `Shop` class stores the available `Products` and keeps track of `Customers` and their `ShoppingBaskets` containing products selected by them for eventual purchase. A `Customer` refers to its `ShopAccount` which keeps track of the customer's purchase history. Customers belong to a `LoyaltyCategory` which can be either bronze, silver or gold. Customers can also create `orders` for specific products which are processed asynchronously by the store.

When the customer confirms a purchase, the `checkout(aShoppingBasket)` method is invoked on the store. This causes the `CheckoutProcess` to be started. This process consists of different steps, modelled as subclasses of `CheckoutStep`: payment, wrapping, shipping and delivery. The first action taken as part of the checkout is to calculate the total price. This is done by invoking the method `getTotalPrice()` on the `ShoppingBasket` received as parameter. In this method, the individual prices of the selected products are obtained (by invoking the `getPrice()` defined in `Product`) and summarized. Afterwards, the total amount spent and the number of purchased products are used to keep the customer's account up-to-date for auditing purposes. Figure 2.1 shows a class diagram of a possible implementation solution for this basic functionality.

## 2.2 Business Rules

A business rule is defined by The Business Rules Group as a *statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control the behaviour of the business* [BRG01]. Different kinds of business rules can be distilled. For instance, von Halle [vH01] identifies the following categories:

- constraints: a constraint is a mandatory or suggested restriction on the behaviour of the core application, such as a *customer must not purchase more than 25 products at one time.*

- action enablers: an action enabler rule checks conditions at a certain event and upon finding them true applies an action. An example is *if a customer is registered, then show his or her recommended products.*

- derivations: there are two kinds of derivations:

    - computations: a computation checks a condition and when the result is true, provides an algorithm for calculating the value of a term using typical mathematical operations. An example of a computation is *if a customer is a frequent customer then subtract 10% from the purchased products.*
    - inferences: an inference also checks a condition but upon finding it true establishes the truth of a new fact. An example inference states that *if a customer has purchased more than 20 products then he or she is a frequent customer.*

In this dissertation we concentrate on derivation rules.

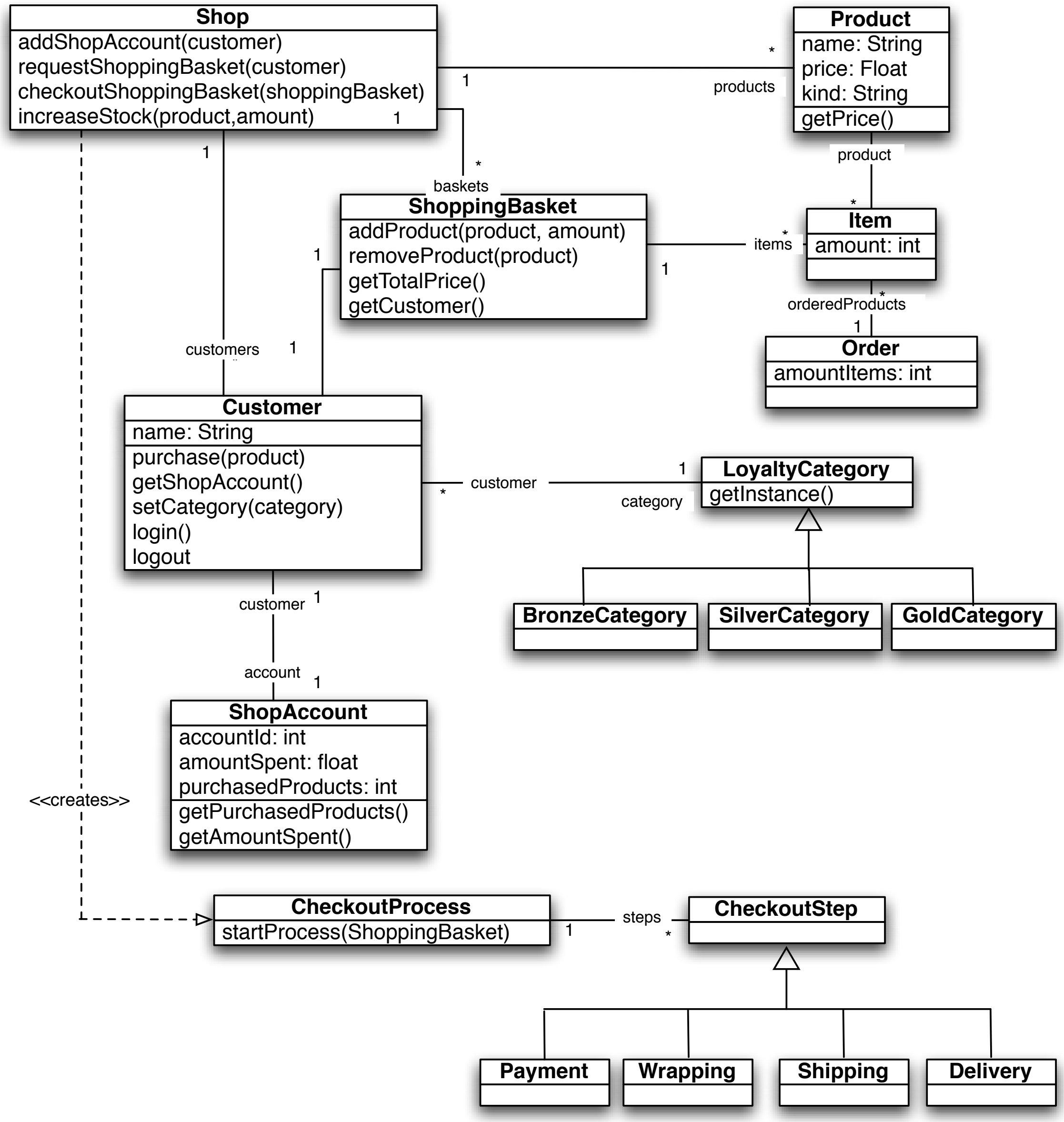


*Figure 2.1:* Class diagram of a possible implementation solution for the e-commerce functionality

## 2.3 Business Rules for Personalization in the e-commerce Domain

In the running example, we focus on business rules for the personalization of e-commerce applications. Personalization has become a very important and pertinent issue in e-commerce applications, as the special issue of the Communications of the ACM [cac02] shows. We can observe nowadays that almost every e-commerce store includes some kind of personalization in order to flexibly accommodate the user's needs. In this section we list several commonly found policies in e-commerce applications that are sources of personalization:

- policies typically found in online stores, e.g. *Amazon* (http://www.amazon.com) and

*Proxis* (http://www.proxis.be web site at amazon.co.uk) typically regulate:

  - price discounting [GLC99]:
  - recommendations
  - availabilities of products
  - returns
  - delivery rates
  - delivery restrictions
  - refunds
  - usage restrictions
  - lead time to place an order
  - canceling of orders
  - creditworthiness, trustworthiness, and authorization
  - customer fidelity categories

- policies for personalizing links, structure, content and behaviour of web pages [RSG01b]
- policies that determine the control flow of online purchases [AA01]

Extensive literature can be found on approaches that tackle some of these personalization aspects. For instance, solutions to derive and explicitly model user profiles based on information gathered from the internet can be found in [PE00] as well as recommendation mechanisms are proposed in [SKR99]. Further requirements for the personalization of web applications can be found in [KRS00].

In the following sections we focus on some of these personalization issues and give concrete examples of business rules typically found in the e-commerce domain.

### 2.3.1 Discount Business Rules

Some examples of business rules for price personalisation of e-commerce applications are:

- ***BRChristmasDiscount**: If today is Christmas then apply a 5% discount on any customer's purchase*
- ***BRPurchasedDiscount**: If a customer has purchased more than 2 products of the same kind then she gets a 10% discount on the next product of that kind*

In the first example, the discount should be applied on the current purchase whereas in the second, the customer is entitled to a discount to be applied to her next purchase.

### 2.3.2 Categorization Business Rules

Other rules can be considered in order to classify customers in loyalty categories. Assume that this decision is based on the amount of money customers already spent in the store. The following rules can be defined:

- ***BRBronzeCustomer**: if a customer spent up to 200 euros then the customer's loyalty category is* bronze

- ***BRSilverCustomer**: if a customer spent between 200 euros and 400 euros then the customer's loyalty category is* silver

- ***BRGoldCustomer**: if a customer spent more than 400 euros then the customer's loyalty category is* gold

The fact that a customer belongs to a certain loyalty category can in turn be the reason for the application of other price discounts. This is specified by the following example rule:

- ***BRGoldCustomerDiscount**: If a customer's loyalty category is* gold *then he or she gets a 20% discount*

## 2.4 Applying Personalization Business Rules

Typically business rules are applied at *events* which are well-defined points in the execution of the core application functionality. They are based on the execution of core methods or property accesses. Example events are:

- ***event1**: After the price of a product is retrieved*
- ***event2**: After the customer has checked out*

Moreover, the application of a given rule can be restricted to certain contexts. For instance, a discount rule — which would typically be applied when the product price is retrieved — can be restricted only to those price retrievals that occur *while the customer is checking out*, or within the period of time *between the moment the customer logs in and the moment he/she adds a product to the shopping cart*, or *not while the customer is browsing the products.* Consider the following definition:

- ***event3**: After the price of a product is retrieved while the customer is checking out*

This last event only captures those price retrievals that occur in the context of the checkout process. Therefore, price retrievals occurring for instance when simply browsing the e-store's catalog are not of interest, since personalized discounts do not apply.

We could for example trigger *BRChristmasDiscount* at *event1*, *BRGoldCustomer* at *event2* and *BRGoldCustomerDiscount* at *event3*. Moreover, *BronzeCustomerRule* and *SilverCustomerRule* can also be triggered at the same *event2*, which shows that different rules can be triggered at exactly the same event. Triggering different rules at the same event might cause rules to interfere if they have conflicting actions — however this is not the case in our example.

## 2.5 Rule Object Pattern

The separation of business rules is pursued along the different phases of the software development process. In this section we present an approach that proposes using object-oriented design patterns to tackle this separation from design to implementation, namely the *Rule Object Pattern*. In this approach business rules are represented as *Rule Objects*. In this dissertation we consider a simple version of this pattern for the representation of the business rules. This representation is relevant also in the coming chapters, when integrating decoupled rules using AOP (chapter 3) and when translating high-level rules to classes (chapter 5).

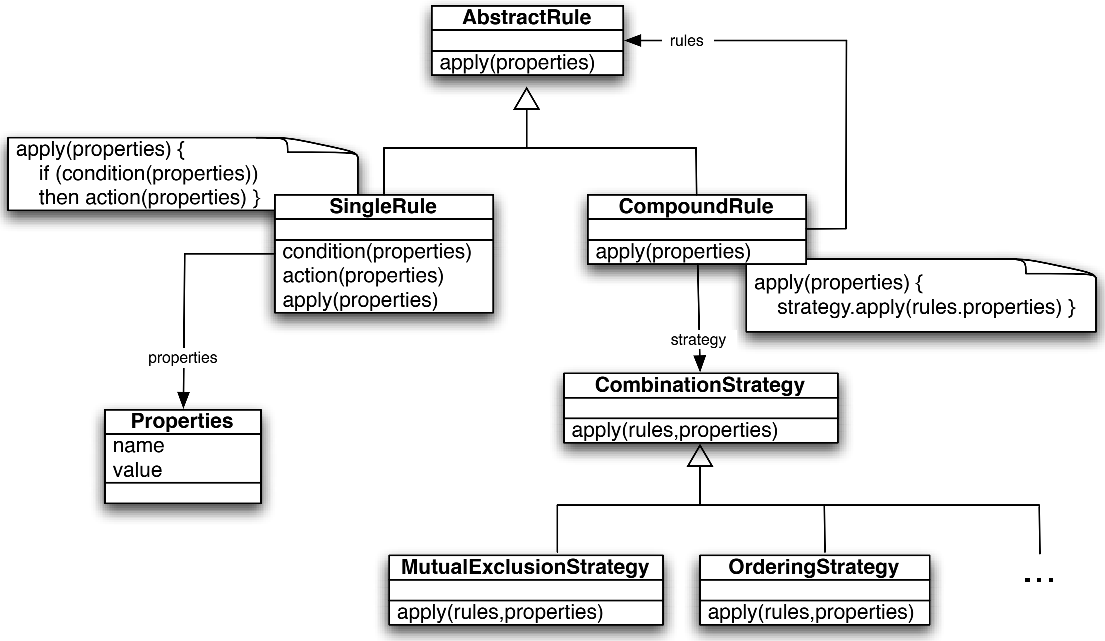


*Figure 2.2:* Rule Object Pattern

### 2.5.1 Simple Rule Object

The *Rule-Object Pattern* is proposed in [Ars01] as a solution to the common problems encountered during the modeling, design and implementation of business rules. This pattern suggests reifying business rules as *Rule Objects*. In its simplest form, a class is defined per business rule implementing methods for its condition and action and another method for triggering the application of the rule. These methods need to be explicitly invoked wherever in the core application this rule logic is needed.

The Rule-Object Pattern can become quite elaborate by employing more sophisticated design solutions. For instance, it advices encapsulating conditions and actions in their own classes when they tend to increase in number and vary very often, making them interchangeable and pluggable; also, rule objects can be parameterized with a *property list* containing the data to be manipulated by the rule. Moreover, it suggests the use of any number of the well-known design patterns [GHJV95]. For example, the *Composite Pattern* can be used to model related business rules that need to be triggered at the same events, the *Strategy Pattern* can be used to model different conflict resolution mechanisms to solve rule interference, such as for instance mutual exclusion or ordering strategies. This basic set-up of the Rule-Object Pattern that uses the *Composite* and *Strategy* patterns is shown in Figure 2.2. Furthermore — however not shown here — the *Factory Pattern* can be employed to instantiate condition and action classes, and the *Mediator Pattern* to control the execution of rule objects, conditions and actions, especially when they are compound.

## 2.6 Implementing Rule Objects

In this section we show how the Rule-Object Pattern can be used to implement some of the example rules presented in section 2.2.

First of all, code fragment 2.1 shows the implementation of an abstract price personalization business rule in Java. The abstract class `BRPriceDiscount` defines a `percentage` attribute and implements an abstract method `condition()` and a concrete method `action(Float price)`. It also implements a method `apply()` that tests the condition, and performs the action if the condition evaluates to true or returns the original price passed as parameter if it does not. The method `action()` subtracts the `percentage` attribute from the original price.

```
abstract public class BRPriceDiscount {
 protected float percentage;
 abstract public boolean condition();

 public Float action(Float price) {
  return(new Float(price-price*percentage/100));
 }

 public Float apply(Float price) {
  if (condition())
   return action(price);
  else return price;
 }
}
```

*Code Fragment 2.1:* Implementation of an abstract price discount rule object in Java.

The `BRPriceDiscount` abstract class can be subclassed with the implementation of concrete price discount rules, such as the *BRChristmasDiscount* rule. This is shown in code fragment 2.2. The `BRChristmasDiscount` class assigns a concrete value to the inherited `percentage` attribute and provides a concrete implementation for the `condition()` method which tests whether today is Christmas day. The class `BRGoldCustomer` (shown in code fragment 2.3) is not a price discount rule and therefore does not inherit from `BRPriceDiscount`. Instead, it determines the logic under which the customer's category must be considered gold. Its `condition(Customer c)` method checks whether the customer received as parameter has spent more than 400 euros (information stored in the customer's shop account and retrieved by invoking the `getAmountSpent()` method on it). The `action(Customer c)` method sets the customer loyalty category to gold by invoking the method `setCategory(LoyaltyCategory c)`.

```
public class BRChristmasDiscount extends BRPriceDiscount {
  public BRChristmasDiscount() {
    percentage = 5;
  }

  private boolean isChristmas() { ... }

  public boolean condition() {
    return isChristmas();
  }
}
```

*Code Fragment 2.2:* Rule object implementing the `BRChristmasDiscount` rule in Java

```
class BRGoldCustomer {
  public boolean condition(Customer c) {
    return (c.getShopAccount().getAmountSpent() > 400);
  }

  public void action(Customer c) {
    c.setCategory(GoldCategory.getInstance());
  }

  public Float apply(ShoppingBasket sb) {
    if (condition(sb.getCustomer()))
      return action(sb.getCustomer());
  }
}
```

*Code Fragment 2.3:* Rule object implementing the `BRGoldCustomer` rule in Java

```
class BRGoldCustomerDiscount extends BRPriceDiscount {
  percentage = 20;

  public boolean condition(Customer c) {
    return (c.category.isGold());
  }

  public Float apply(ShoppingBasket sb) {
    if (condition(sb.getCustomer())) {
      return action(sb.getTotalPrice());
    else
       return sb.getTotalPrice();
    }
  }
}
```

*Code Fragment 2.4:* Rule object implementing the `BRGoldCustomerDiscount` rule in Java

Another concrete price discount rule is implemented by the `BRGoldCustomerDiscount` class which therefore inherits from `BRPriceDiscount` (shown in code fragment 2.4). It defines a new `condition(Customer c)` method that checks whether the loyalty category of the customer received as parameter is gold. It also redefines the inherited `action()` and `apply()` methods as they require a shopping basket object to be received as parameter. This shopping basket object must be available and passed to the rule objects when they are invoked.

## 2.7 Integrating Rule Objects

The usual way of connecting a rule object is by explicitly invoking its `apply` method. This invocation has to be repeated everywhere in the code of the core application where that rule logic is needed. For example, when considering the integration of the *BRChristmasDiscount* at *event1*, the implementation of the `getPrice()` method defined in `Product` needs to be modified to include the invocation of the `apply()` method on an instance of the `BRChristmasDiscount` class. This is depicted in the following code fragment:

```
public Float getPrice() {
 BRChristmasDiscount br = new BRChristmasDiscount();
 return br.apply(this.price);
}
```

However, this way of connecting rule objects is not very flexible since it requires manually changing the implementation of the `getPrice()` method — which is part of the core functionality — every time the requirements for the connection of the business rules change. Approaches have been proposed in [RSG01a; RFCS01] in order to achieve a more flexible integration of business rules in e-commerce applications. They propose flexible design solutions that build on top of the well-known design patterns [GHJV95] with the objective to minimize coupling between rule objects and the core application. This is achieved by introducing a set of intermediate objects (e.g. wrappers and personalizers) in charge of deciding which rules to trigger and how to configure them.

In our example, we can avoid hardcoding the connection of the *BRChristmasDiscount* by introducing a `ProductWrapper` object which delegates this responsibility onto a `ProductPersonalizer` object. Figure 2.3 shows this solution which uses the *Strategy* pattern for modeling the product personalizers and the *Decorator* pattern for the product wrappers. As a consequence, a higher flexibility is achieved as this design allows easily "switching" between different personalized behaviors in different contexts, e.g. discounts applied to a single product, to many products, to products that are part of a special promotional package, etc. The same idea — although not shown here — can be used to personalize other aspects, such as the recommendations and product information. Consequently, this design solution can become quite elaborate as more patterns are employed and combined.

Although these patterns allow for a more flexible rule connection, they fall short when integrating more sophisticated and unanticipated business rules at different dynamic events. In the following section we analyze the encountered limitations and identify the necessary requirements for achieving a fully flexible business rules connection. The connection issues are also presented in [Cib02; CDJ03; CDS$^+$03; CDS$^+$05].

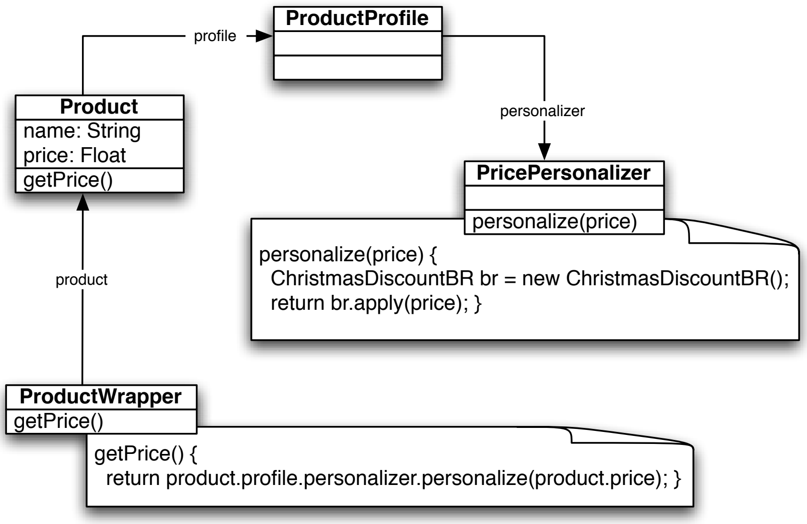


*Figure 2.3:* Integration of price personalization rule objects using the Strategy and Decorator design patterns

## 2.8 Towards a Flexible Rule Connection

We observe that existing approaches that deal with the decoupling of business rules suffer from one common problem: they all focus on separating the business rules from the core application, but do not support at all the encapsulation of the connection of the business rules with the core application. Essentially, by *rule connection code* or *rule connection* we refer to the code in charge of denoting the events at which rules are applied, capturing the required data and making it available for rule manipulation. Using the existing business rules approaches, however, one has to adapt the source code of the core application manually in different places whenever a business rule is plugged in or out. Depending on the approach this is done differently, but essentially the result is the same: the rule connection code is *scattered* among many modules and *tangled* with code addressing other concerns of the core application functionality. This situation is identified as *crosscutting code* in the area of *Aspect-Oriented Programming* [KLM$^+$97]. The encapsulation of crosscutting rule connection code is crucial to achieve maximum business rules configurability.

The goal of this section is twofold:

i illustrate that, even when the rules are successfully decoupled, it is hard to integrate them using standard object-oriented programming. We show this by presenting examples that require the core application code to be manually extended or modified when the business rule connections change. Some examples are coded and others are described, and clearly show that statements need to be added in different places of the core application implementation. Moreover, the new code cannot be always encapsulated in objects.

ii distill a set of requirements for achieving a flexible rule connection. We describe and motivate the set of requirements that are essential for any technology to be suitable to cleanly encapsulate the rule connection code and achieve high flexibility in the integration of the rules. These requirements are described independently of concrete implementation languages and/or technologies. They will be revisited in the chapter 3 in which we show how AOP succeeds in meeting them.

### 2.8.1 Denoting Rule Application Time with Dynamic Events

The rule application time is denoted by events which capture well-defined points in the execution of the core application where business rules are applied. Examples of events are method invocations and property accesses. They can be distributed in the core application, for example in objects with different types. Since business rules change often and new ones are added regularly, it is generally not possible to anticipate all the events at which they are going to be triggered. In current approaches, explicit hooks for the events have to be foreseen in the core application. Moreover, dynamic events can depend on properties only available at run-time, such as control flow in our example *event3*. Expressing this event would typically be resolved by adding a flag which is set to true when the checkout process is started, in other words when the `checkout(aShoppingBasket)` method is invoked on a store. An extra condition evaluates this flag in `ProductWrapper` before delegating the personalization of the product price on the corresponding personalizer. Consequently, the extra flag and condition becomes tangled with and scattered among the core application's implementation. Hence, a mechanism is needed that allows specification of dynamic — and even unanticipated — events that may depend on properties available at run-time, without having to change the source code manually.

### 2.8.2 Exposing and Passing Available Contextual Information

This requirement identifies the need for making sure that the available data — that needs to be manipulated by the rule — is passed to the rule at rule application time. Some rules depend on properties of objects that are in the scope of the dynamic event that activates the rules. In our example, the *BRChristmasDiscount* needs the system date as well as the original product price that needs to be personalized. As the system date is global information, it is always available and directly accessible to the rules. The original product price can be obtained by retrieving the property `price` on the target product. This is only possible because the integration of the rule is hardcoded in the method `getPrice()` of the `Product` class, and because the required product object is the target object of this method which can then be directly passed to the rule. However, as identified by the first requirement, anticipating rule integrations results in crosscutting code and therefore a mechanism based on dynamic events is preferred. These dynamic events must be able not only to denote well-defined points in the execution of the core application, but also to expose and manipulate the information available in the dynamic context of those events in order to pass it to the rules at rule application time.

### 2.8.3 Capturing, Exposing and Passing Unavailable Information

Rules might also require information from specific objects that are outside the scope of the dynamic event which activates the rules. Generally, capturing this unavailable data at the point when it is available and retrieving it at the desired dynamic event involves introducing a global variable or outfitting all methods in the control flow between those points with an extra parameter, resulting in crosscutting code. We illustrate this situation by introducing a simple change in the business requirements: imagine that the *BRChristmasDiscount* does not fix a 5% discount percentage but instead it specifies that a different discount percentage must be applied depending on the actual customer. To incorporate this change, the constructor of the `BRChristmasDiscount` class has to be modified so that the percentage can be received as a parameter. A new design is depicted in Figure 2.4 which shows the changes that have to be introduced in the design of Figure 2.3 in order to realize this requirement. Personalizers are introduced for the customers in order to obtain a different personalized discount percentage per customer. Moreover, wrappers are needed to add the `getDiscount()` functionality to the `Customer` class, again implemented using the *Decorator Pattern* (shown in upper part of Figure 2.4). The `ProductWrapper` class now defines the method `getPrice(Customer)` which delegates this functionality onto a `ProductProfile` object and finally onto the right price product personalizer. The product personalizer then first needs to query the discount percentage from the customer received as parameter and use it to initialize the `BRChristmasDiscount` (shown in lower part of Figure 2.4). The problem of tangled and scattered code is clear: the `ProductWrapper` class had to be modified to receive the customer as parameter; a new `getPrice(aCustomer)` method had to be manually added; the objects requesting product prices now need to invoke the new method `getPrice(aCustomer)` instead of the original `getPrice()`.

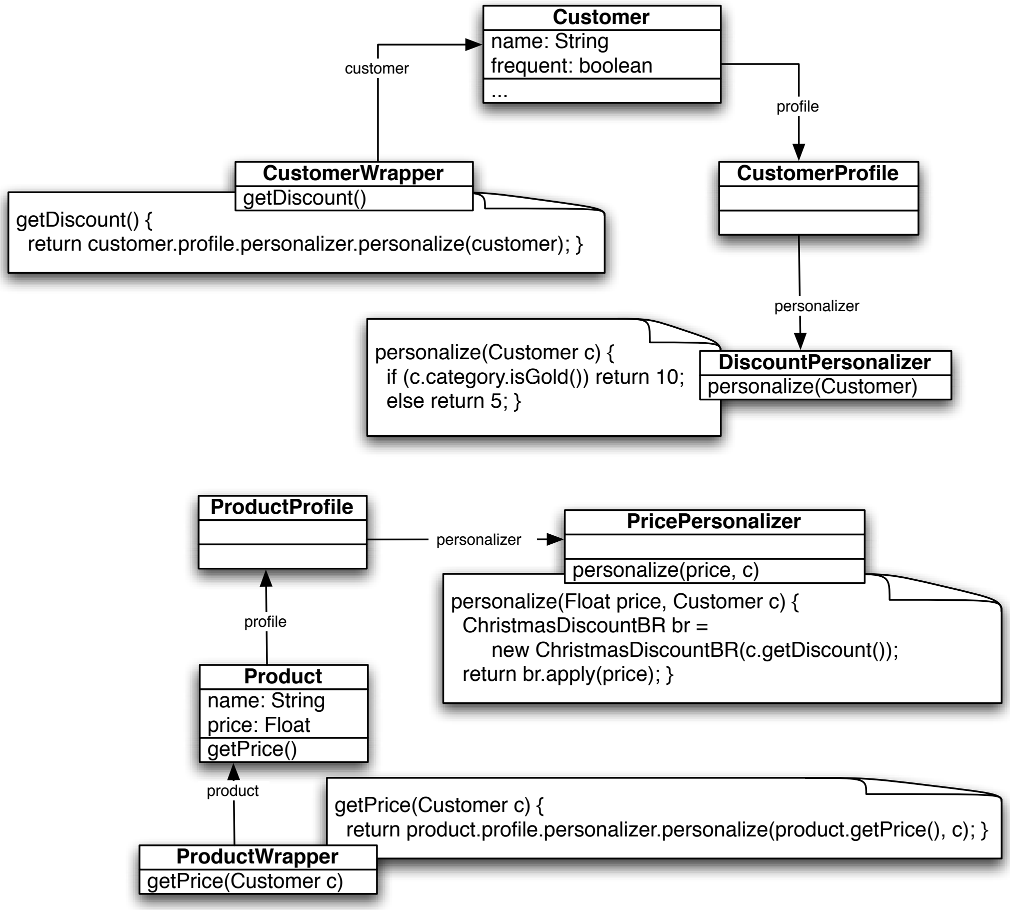


*Figure 2.4:* More complex pattern-based design solution for achieving a price personalization that differs per product and per customer

We observe the need for a mechanism that is able to identify the points in the core functionality where the required objects are available, capture and expose them in order to make them accessible to the rules, without having to change the core application's code manually.

### 2.8.4 Introducing Unanticipated Information

Consider the following rule:

***BRFrequentCustomerDiscount**: If a customer is* frequent *then he or she gets a 10% discount*

This rule requires information — namely the concept of *frequent customer* — that was not explicitly foreseen at the moment the core functionality was designed and implemented. This concept could be incorporated by extending the `Customer` class with some way of determining when a customer is frequent. For instance, a new method `isFrequent()` can be added, which returns whether the customer has purchased more than 20 products. Besides having to add this functionality manually, this solution has the problem that it cannot non-invasively change the way of determining when a customer is frequent. A more flexible solution is to encapsulate the definition of *frequent customer* in a business rule:

***BRFrequentCustomer**: If a customer has purchased more than 20 products then he or she is a* frequent *customer*

However, this rule still expects the class `Customer` to have a boolean attribute `frequent`, which is not anticipated in the original design and thus needs to be added manually. This again results in crosscutting code. Therefore, when the need for unanticipated structure and behaviour arises, a mechanism is needed that enables the non-invasive introduction of new objects, attributes and operations to the existing implementation of the core application. The new code should be encapsulated so that it can be reused or removed easily.

### 2.8.5 Incorporating Rule Results

Once a rule has been applied, the results of its application must be considered back in the context of the core application's execution. Depending on the mechanism used for implementing the rules (whether it is rule objects or a rule-based language for instance) we might need to process these results is different manners. For instance, a rule object can trigger a certain action by means of invoking a method on an object of the core application. This invocation can be side-effect free — in the case of a getter, e.g. `Product.getPrice()` — or it can induce changes in the state of core application's objects — when invoking setters or methods that change object's properties, e.g. the method `Shop.increaseStock(Product, int)` which modifies the shop's `stock`. Rules implemented in a rule-based approach are more declarative since it would not directly change the state of an object but instead new information will be inferred which then needs to be explicitly retrieved and considered — or eventually discarded — in the core application's execution. Of course nothing impedes having more declarative rule objects that besides invoking methods on core application objects also conclude new information. In this case a mechanism is needed in order to retrieve this new information and use it accordingly when proceeding with the core application's execution.

### 2.8.6 Configuring and Reusing Rules and Their Connections

The different parts of the rule connection, i.e. dynamic events, introduced and captured data, must be configurable and reusable. It must be possible, for example, to connect new business rules reusing a dynamic event at which other rules are applied. This might require to configure differently the way the required and available data are mapped. For instance the *BRGoldCustomerDiscount* can be connected at the same event *event3* at which the *BRFrequentCustomerDiscount* rule is applied, reusing also the way the required information is mapped to the available one, as they both require the same objects — i.e. the original price and the customer. Inversely, the same rule could be connected at different events. For instance, *BRChistmasDiscount* can be connected at *event2* instead of at *event3*.

### 2.8.7 Controlling Rule Precedence, Combination and Exclusion

Some rules may specify actions that conflict with the ones specified by other rules. In addition, some rules can have precedence over others or should not be applied when others are deployed. This can be solved by explicitly specifying combination strategies as the ones shown in Figure 2.2. For example, assume *BRFrequentCustomerDiscount* and *BRGoldCustomerDiscount* are applied at the same *event3* and that we want to avoid applying the two discounts on a single purchase. Thus, we can specify that when both rules apply, *BRFrequentCustomerDiscount* mutually excludes *BRGoldCustomerDiscount*. In order to address these complex interdependencies among rules, combining and prioritizing the modules that encapsulate the rule connections is required. Moreover, we need to be able to explicitly control the application of the rules.

### 2.8.8 Controlling Rule Instantiation and Initialization

It should be possible to control the instantiation of rules at different events, and vary their initialization properties from one rule connection to another. For instance, the same rule logic might still be valid but slightly modified, such as the Christmas discount being able to vary from 5% to 10% according to certain conditions. Therefore a mechanism to reuse business logic and configure it accordingly is needed. Considering the volatility of rules, this is a vital requirement, as it allows customizing application-independent rules to conform to a specific integration.

### 2.8.9 Connecting Rules

A flexible mechanism is needed in order to deal with the volatility inherent of business rules: existing rules might become obsolete and new rules might need to be considered. In some application these changes can be done by manually removing or adding the corresponding rule objects, even if this requires stopping the application. An e-commerce application for instance can afford manually changing — e.g. at the end of a season, or during sales period — the rules regulating price discounts. Other more critical or real-time applications might not be able to afford stopping their execution to do these manual changes and therefore it must be possible to dynamically deploy and remove rules.

## 2.9 Summary

State-of-the-art business rules approaches mainly aim at physically separating the rule definitions from object-oriented applications. The integration code for a rule however, still remains tangled in the core functionality itself which impedes the business rules objectives: separate, trace, externalise business rules and position them for change [23]. So far we have seen that rules are applied at different points in the core functionality and that many times the concrete connection code is identical at all these points. It can also occur that the connection code is scattered among different places in the core application. In any case, the connection code is tangled with code addressing other concerns of the core application, and therefore is crosscutting. Therefore, the developer is forced to adapt the existing code manually every time the rule connections change.

In this chapter, we identify a set of requirements, which we believe are essential in order to successfully encapsulate the rule integration code. These requirements are independent of a specific object-oriented programming language in which the core application functionality is implemented, and independent of the business rule representation used — even though rule objects are considered in this thesis.

AOP appears as a promising technique which provides a means to cleanly encapsulate crosscutting code in separate modules. The next chapter digs into how AOP can be used to encapsulate rule connection code successfully at the same time as accomplishing the identified requirements.

# Chapter 3

# Aspect-Oriented Programming for Business Rule Connection

So far we have identified and presented the technological requirements that any suitable technology must support in order to successfully decouple and encapsulate crosscutting rule connections. Those requirements were described independently of any concrete technology. In this chapter we focus on demonstrating the suitability of Aspect-Oriented Programming for realizing those technological requirements. We first introduce the main ideas advocated by AOP 3.1. In section 3.2 we provide a general overview of different AOP approaches and classify them according to two main characteristics: symmetry and weaving time. Section 3.3.1 describes the AOP characteristics that we consider are fundamental for achieving the modularization of rule connection code. We then select two representative approaches that adhere to the chosen characteristics, AspectJ and JAsCo (section 3.3.2). Concrete examples in these approaches are presented along the chapter which show how the encapsulation of crosscutting rule connections can be successfully accomplished.

## 3.1 Aspect-Oriented Programming

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 to low-level notions such as caching, buffering, synchronization and transaction management [EFB01]. In order to deal with all these heterogeneous concerns in a software application, *separation of concerns* (SoC) is fundamental. SoC 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 [Dij76b; OT01]. SoC is a crucial property for realizing comprehensible and maintainable software. It aims at being able to think about the design and implementation of a system in natural units of concerns rather than in units imposed by specific languages or tools. 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 of thinking to the limitations imposed by the languages and tools [KHH$^{+}$01].

Once software systems reach a certain complexity, the modularization constructs provided by current languages and environments fall short in order to separate all these heterogeneous — and even sometimes interrelated — complex concerns. *Aspect-Oriented Programming* argues that some concerns of a system cannot be cleanly modularized using current software engineering techniques, as they 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 [KLM+97]. This problem is identified in [TOHS99] as the "*tyranny of the dominant decomposition*": the program can be modularized in only one way at a time, and as a consequence the many concerns that do not align with that modularization end up *scattered* across many modules and even *tangled* with code that addresses other concerns. Examples of typical decompositions are objects in the object-oriented paradigm, modules in the imperative paradigm, and rules in the rule-based paradigm. In this thesis we concentrate on the object-oriented paradigm as a decomposition mechanism.

The tangled and scattered concerns of a system are referred to as *'system-wide' concerns* as they do not nicely fit into the chosen modularization of the system. Therefore they *crosscut* its main decomposition. Crosscutting concerns are not encapsulated which makes them very hard to add, edit or remove. Typical examples of well-known crosscutting concerns in object-oriented applications are debugging concerns such as logging and contract verification, security concerns such as confidentiality, access control and transactions as well as verification of design or architectural constraints, systemic properties and features.

AOP aims at achieving a better separation of crosscutting concerns in object-oriented software applications. To this end, AOP introduces a separate module — called *aspect* — which is able to encapsulate the implementation of a crosscutting concern. Moreover, AOP allows describing the relationships that exist between the different concerns of a system and the mechanisms to *weave* or *compose* them together into a coherent program. Originally, separation of concerns was oriented towards implementation concerns. More recently, we do not only talk about AOP — aspects at the implementation level — but also about Aspect-Oriented Software Development (AOSD) as the community recognizes the need for separation of concerns throughout the whole software development cycle. This is because crosscutting concerns may arise at any stage of the software development life-cycle, including requirements specification, analysis, design, implementation, debugging, etc.

Although AOP is a rather recent paradigm, numerous aspect-oriented approaches have already been proposed for which advanced tool support has been developed. These include AspectJ [KHH+01], Adaptive Programming [LOO01], Composition Filters [BA01], JBoss/AOP [FR03], Spring/AOP [J+], AspectWerkz [BV] — which has recently merged forces with AspectJ — and JAsCo [SVJ03]. Some of these approaches are currently reaching maturity and are being used also in industrial projects, e.g. AspectJ, JBoss/AOP and Spring/AOP. Another representative approach — radically different to the previous ones — is Hyper/J [OT01].

In this dissertation the separation of crosscutting rule connections at both the specification and implementation levels is of interest. In this chapter the focus is put first on demonstrating the suitability of AOP at the implementation level as we are concerned with the problem of decoupling crosscutting rule connection code. Therefore, in the rest of this chapter we analyze concrete AOP approaches that offer suitable features to accomplish our goal.

## 3.2 Comparing AOP approaches

Many emergent and some mature aspect-oriented approaches exist today [AOS05], each of them offering their own particular technique for separating crosscutting concerns. This means that even though they all are able to encapsulate crosscutting concerns, they can take radically different approaches for doing so. First of all, approaches can differ in the way they carry out weaving. Traditionally, weaving takes place at compile time, which means that the advices are inserted into the target application at the source or byte-code level, but other approaches allow weaving to occur at load or even run time. In the former case we talk about *static* AOP approaches whereas in the latter cases we refer to these approaches as *dynamic*. Examples of the static approaches are AspectJ and AspectWerkz whereas JAsCo, JBoss/AOP and Spring/AOP are examples of dynamic ones.

Another way of classifying AOP languages is according to whether they distinguish between base code and aspect code or not. In the first case, the approach is called *asymmetrical* whereas in the latter, it is referred to as *symmetrical*. Symmetrical AOP approaches are based on *program composition* as they allow elements of both base programs to map onto each other. In fact, symmetrical approaches consider any concern — crosscutting or not — as a program that can be composed with other programs in different ways according to a composition strategy. The best-known symmetrical approach is *Multi-Dimensional Separation of Concerns* [OT01; TOHS99] and the concrete tool *HyperJ* for Java[1]. The authors of HyperJ also recognize the suitability of their approach for the decoupling of business rules and propose a symmetrical solution (discussed in section 9.3.1).

Asymmetrical approaches on the contrary are based on the pointcut/advice model. This model typically considers an aspect as a module that gathers *pointcut* and *advice* definitions. A *pointcut* identifies a set of points in the program's execution where an aspect can be applied. Each of these points is referred to as *joinpoint*. Therefore a pointcut specification is a concise description of a set of joinpoints at which the aspect behavior, i.e. the advice, should be applied. As such, an advice specifies a concrete behavior to be executed at certain joinpoints, typically before, after or around the original behavior intercepted by the joinpoints. In the first two cases, the advice behavior is additional whereas in the last case the advice behavior can completely replace the interrupted behavior but is still able to invoke it if necessary, typically by using a special *proceed* construct. The advice language typically consists of the base language — such as Java — augmented with a limited number of special keywords that offer aspectual reflection and control over the execution of the original joinpoint.

A wide variety of approaches that adhere to the pointcut/advice model is available today. They typically provide a general-purpose aspect language for implementing aspects in a general purpose object-oriented language, most commonly Java. Moreover, some of these approaches focus on expressing aspects in other development environments. Examples are *JAsCo* [SVJ03] and *JAC* [PSDF01], AspectWerkz [BV] and JBoss/AOP [FR03], which provide aspects for component-based development. The last two approaches focus particularly on the J2EE component model. Other approaches improve the original pointcut/advice approach in order to obtain reusable aspects and composition of aspects. In AspectJ, the application context of an aspect is hard-coded in the aspect itself, which severely limits

[1] HyperJ has recently evolved into a broader technology called the Concern Manipulation Environment (CME) available at http://www.research.ibm.com/cme/

reuse of the aspect in other contexts. *Aspectual Components* is one of the first approaches to remedy this, by introducing explicit connectors that connect aspects to a specific context [LLM99]. Newer approaches based on Aspectual Components are *Aspectual Collaborations* [LLO03], *Caesar* [MO03] and *JAsCo* [SVJ03].

## 3.3 Selecting Suitable AOP Approaches

One of the most important requirements (identified in 2.8) for achieving a flexible modularization of the rule connection code is to be able to intercept the application at dynamic points during the core application's execution at which the rules are to be applied. This implies the need for an asymmetrical AOP approach based on the pointcut/advice model.

In our approach there is no actual need for considering core concerns — tackled by the existing core application's implementation — and rule connection concerns at the same level. Moreover, a compositional approach would directly map the two parts that need to be connected — i.e. core application and rules — which is not sufficient to realize a flexible rule connection. This is because a flexible connection implies the issues identified in 2.8 to be tackled as part of the rule connection which cannot be realized in a simple mapping. To this purpose, extra code is needed in between the two connecting parts, i.e. an aspect, and therefore an asymmetrical approach seems more suitable. Such an approach allows us to distinguish between the modules of the existing application and the business rules from the modules of the aspects implementing the rule connections.

Even though asymmetrical approaches are based on a common pointcut/advice model, they can still differ — sometimes even quite radically — in the actual mechanisms and features they offer. Still, we are able to distill a list of AOP characteristics that are fundamental for achieving our goal. This allows us to generalize our findings, not restricting our solution to the features of specific AOP approaches. This way, any suitable AOP approach must adhere to these characteristics in order to be able to modularize rule connections. We explain these general characteristics in the following section.

### 3.3.1 Determining Required AOP Characteristics

In this section we list a subset of these AOP characteristics to which any suitable AOP approach must adhere in order to achieve our goals:

- *Dependency inversion*: AOP allows reversing the typical dependency relation that exists in traditional object-oriented languages, that is the explicit reference that a program must include in order to call a module whenever the functionality encapsulated by that module needs to be executed. Aspects on the contrary do not need to be explicitly invoked in order to get executed. It is the responsibility of the aspects to 'observe' the static structure or dynamic execution of the base program and to 'react' accordingly — by executing the additional crosscutting behavior — at the desired places [NI01]. This way, the base program remains untouched. The AOP mechanism of 'observing' and 'reacting' is based on the definition of *pointcuts* which designate places in the core application's execution at which aspects need to be executed. When the execution of the core application reaches those places, the control is transferred from the core application to the applicable aspects. Moreover, those aspects are able to decide whether and under which conditions the intercepted execution must be resumed.

- *Pointcut* context exposure: it refers to the ability to expose, inspect, access and manipulate the information available in the context of a dynamic point in the execution of the core application. The first property — dependency inversion — allows us to actually intercept the application at dynamic points. The context exposure property allows us to have control over the available information at those points — e.g. executing objects and method parameters — and even change their values, possibly influencing the way the intercepted application is resumed.

- *Inter-type declaration*: It refers to the ability to express crosscutting concerns affecting the structure of existing classes. This feature allows declaring additional members — attributes and methods — on certain core classes without having to anticipate these extensions in their original implementation. This feature is also known as open classes or mixins.

- *Pointcut composition*: AOP provides mechanisms for relating pointcuts together in order to:

  i) Capture more finely grained dynamic events of the core application execution. This is achieved by defining composite pointcuts which compose individual pointcuts using certain composition strategies (such as *and*, *or* and *not*).

  ii) Enable pointcuts to share the contextual information and make it available to other pointcuts.

- *Aspect interaction*: Aspects that are triggered at the same joinpoints might define conflicting behaviors and therefore their execution must be controlled. AOP provides mechanisms for controlling the execution of conflicting aspects.

These AOP characteristics are depicted in Figure 3.1. This figure also depicts the general set up of our approach where we can distill three layers: the left-hand side shows the existing core application developed in object-oriented paradigm, the right-hand side shows the business rules implemented as rule objects, and the middle part shows the aspects implementing the connection of the rules with the core application. The arrow going from *ConnectionAspect1* to the *Customer* class depicts the *inter-type declaration* property: the *ConnectionAspect1* extends the definition of the core class `Customer`. The dotted arrows going from *ConnectionAspect2*, *3* and *4* to dynamic points in the execution of the core application represent pointcuts and thus illustrate the *dependency inversion* property by showing that the aspects observe the system's execution at those points. Also, at any of those points, the execution context is exposed by the corresponding pointcuts and therefore it is made available to the aspects, as it is illustrated with the line that borders the activation frames for the methods `checkoutShoppingBasket(sb)` and `getPrice()`. The star in *ConnectionAspect4* illustrates the *pointcut composition* property (case (i)) as it shows that pointcuts labeled *a* and *b* can be combined in order to specify finer-grained joinpoints. The stars in *ConnectionAspect2* and *ConnectionAspect3* and the arrow between them illustrate that pointcuts can share their contextual information (case (ii) of the same pointcut composition property). Finally, the arrow between *ConnectionAspect3* and *ConnectionAspect4* depicts the fact that the interaction between aspects that apply at the same joinpoints need to be explicitly controlled.

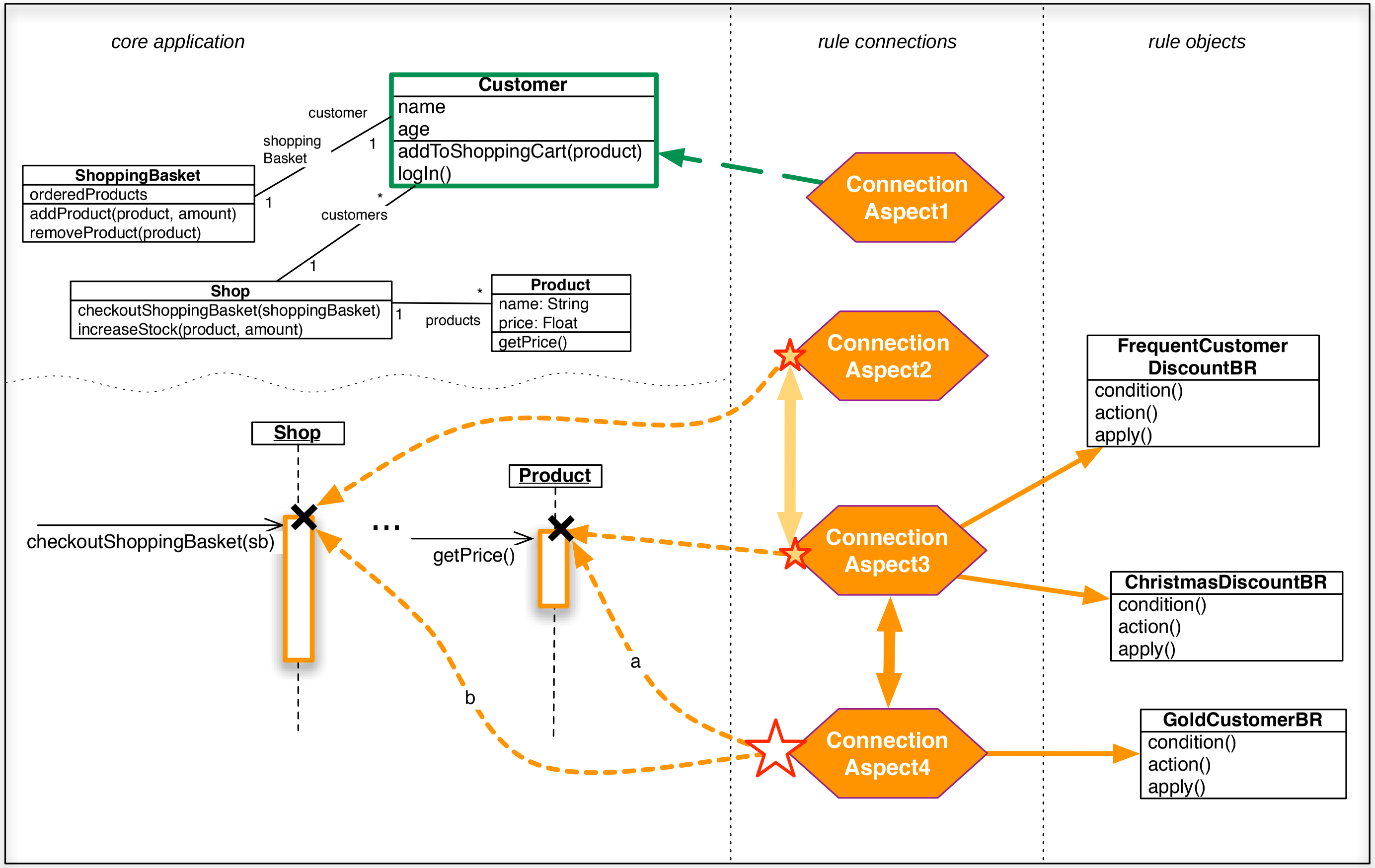


*Figure 3.1:* General architecture of the use of AOP for the modularization of rule connections

### 3.3.2 Selecting Representative AOP Approaches

For illustration purposes, we select two representative approaches among the ones based on the pointcut/advice model: AspectJ and JAsCo.

The first experiments were carried out using AspectJ, one of the most mature approaches which is considered the "mother" of all pointcut/advice approaches. AspectJ is a general-purpose extension to Java that provides support for modular implementation of crosscutting concerns [16]. In addition to containing fields and methods, AspectJ's aspect declaration contains pointcut and advice members. AspectJ's main advantage is the expressiveness of its joinpoint model. AspectJ has been successfully used for clean modularization of crosscutting concerns such as tracing, contract enforcement, display updating, synchronisation, consistency checking, protocol management and others [KHH+01]. For an overview of AspectJ which discusses these constructs thoroughly, we refer the reader to [KHH+01] and the web site aspectj.org. The results from this first experiment are also reported in [Cib02; CDJ03; CD03].

The second experiments were carried out using the dynamic AOP approach JAsCo [SVJ03]. JAsCo provides a pointcut model which is as expressive as AspectJ's and in addition it provides features for the explicit specification of aspect combination, precedence and instantiations — very rudimentary in AspectJ — which are fundamental for achieving a high flexibility in the connection of the rules (as identified by the requirements listed in 2.8). The JAsCo language is also an aspect-oriented extension for Java and stays as close as possible to its original syntax and concepts. Moreover, JAsCo aspects are highly-reusable as they are not tightly coupled to the base implementation. This is achieved by decoupling the aspect logic and the deployment logic in two different constructs: *aspect beans* and *connectors*.

An *aspect bean* allows describing crosscutting behavior in an abstract way, independent of the base application. It is an extended version of a regular Java bean and is specified independently of concrete component types and APIs, making it highly reusable. An aspect bean can group the definition of one or more logically related *hooks* that describe the crosscutting behavior itself. A *hook* includes a special constructor that defines — in an abstract way — when the hook has to be triggered. A constructor receives several abstract method parameters that are bound to one or more concrete methods at the moment the aspect is deployed. The constructor body specifies when the hook needs to be triggered. The advices — before, after and around — of a hook are used to specify the various actions a hook needs to execute when the hook is triggered. The concrete places where the hook functionality has to be executed are specified in the *connectors*. Therefore connectors are used to deploy aspect beans onto a concrete context. Also connectors can specify explicit combinations among two or more aspect beans.

Finally, the JAsCo technology excels at providing dynamic integration and removal of aspects with a minimal performance overhead. For more detailed information about JAsCo, the interested reader is referred to [SVJ03; VSV+05; VSCF05]. We conducted an elaborate experiment for connecting rule objects to object-oriented applications using JAsCo, which is reported in [CDS+03; CDS+05; CSD+04].

These two experiments let us observe that the concrete features of both AOP languages complement each other for the successful decoupling and modularization of rule connections. We also review other representative asymmetrical aspect-oriented approaches —

also based on the pointcut/advice model —, namely JAC [PSDF01], AspectWerkz [BV] and JBoss/AOP [FR03], and compare their features to the ones of the selected approaches to evaluate their suitability for the decoupling of rule connections. The investigated approaches are also general-purpose and are considered quite mature in terms of tool support and adoption [Ker05]. All the selected asymmetrical AOP approaches adhere — in one way or another — to the required AOP characteristics and offer mechanisms to reproduce the architecture shown in Figure 3.1, as it is explained in the following section.

## 3.4 AOP for Rule Connection

The goal of this section is to illustrate the suitability of AOP for modularizing crosscutting rule connection code by showing how the chosen representative approaches accomplish the requirements of section 2.8. Organized per requirement, this section points out which AOP characteristics are needed to accomplish it — from the list of characteristics identified in section 3.3.1 — and illustrates this with examples in AspectJ and JAsCo.

In the rest of the chapter examples in the e-commerce application are shown in both selected languages. We do this by showing code (sometimes simplified to avoid cluttering the solution with unnecessary details), pseudo-code or diagrams.

The following naming conventions are used in the implementation of the examples:

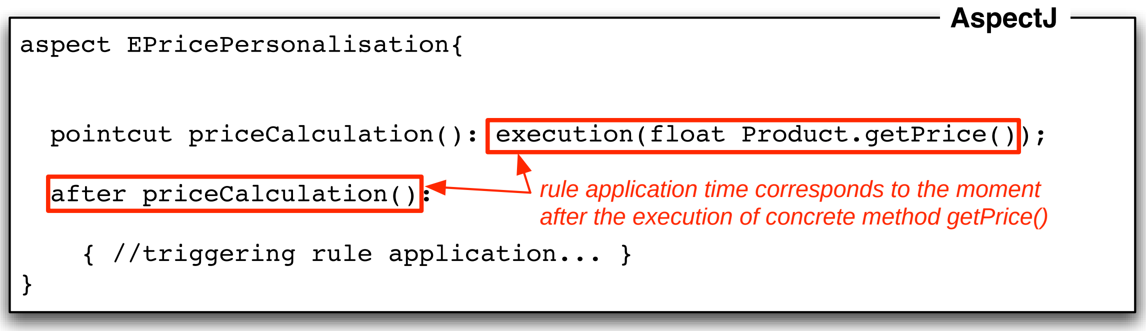
- names of business rule classes start with *BR*
- names of event aspects start with *E*
- names of data exposing and introducing aspects start with *Capture* and *Extend* respectively
- names of business rule application aspects start with *Apply*

### 3.4.1 Denoting Rule Application Time with Dynamic Events

One of the first requirements identified in section 2.8 is the need for denoting dynamic events in the execution of the core application at which rules need to be connected. Examples of events are method invocations and instance variable accesses. As we want to avoid hardcoding these dynamic events in the implementation of the core application, support for *dependency inversion* is fundamental. Pointcut are able to capture dynamic points in the execution of the core application without having to manually anticipate this in the core application's code and therefore are a suitable solution for designating the events at which rules need to be connected.

#### 3.4.1.1 A Simple Event

As a first example of how AspectJ can be used to specify the rule application time, we present an implementation of the event *event2* introduced in chapter 2 which denotes the moment after the price of a product is retrieved. A pointcut `priceCalculation(Product p)` is defined which captures every call to the method `getPrice()`. This definition is included in the aspect `EPricePersonalisation` shown in code fragment 3.1.



AspectJ

```
aspect EPricePersonalisation{


  pointcut priceCalculation(): execution(float Product.getPrice());

  after priceCalculation():

    { //triggering rule application... }
}
```

rule application time corresponds to the moment after the execution of concrete method getPrice()

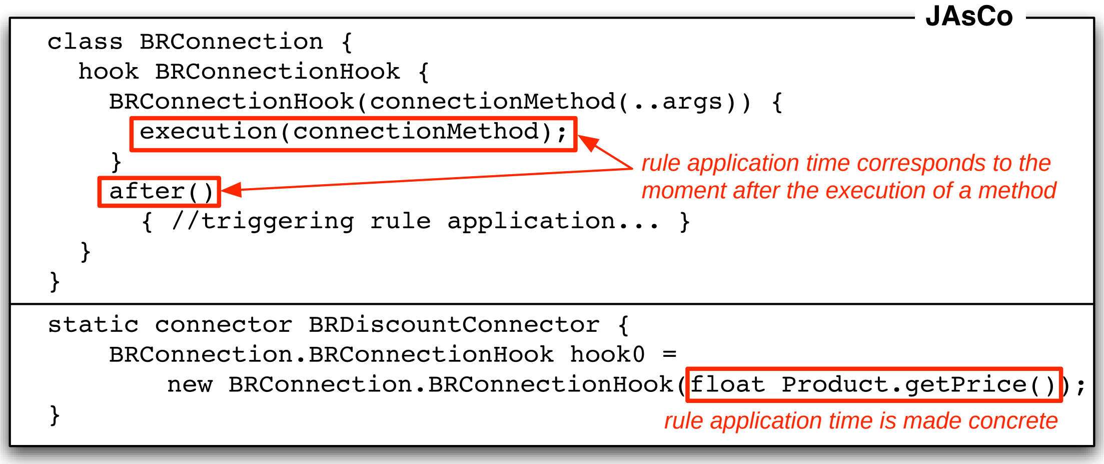
*Code Fragment 3.1:* AspectJ's solution for denoting a simple event capturing the moment after the invocation of the method `Product.getPrice()`

In JAsCo, pointcuts can be used in a similar way as in AspectJ to capture dynamic events. The only difference is that JAsCo pointcuts can only capture method executions and not property accesses. Therefore the existence of getters and setters for the involved properties is required, in the case of events that involve property accesses. The upper part of code fragment 3.2 shows the definition of an abstract pointcut capturing a very generic rule application time, which corresponds to the invocation of any method with any number of parameters. This pointcut is defined in the hook `BRConnectionHook` which is defined in the aspect bean `BRConnection`.

JAsCo's pointcuts are more reusable than AspectJ's as they do not refer to a concrete method signature but are defined in an abstract way. This is a very general example but note that it is also possible to define certain restrictions on the signature of the methods that are to be bound to the abstract methods calls, such as restriction on the type of the arguments, the number of arguments, etc. The concrete binding between abstract method calls and concrete methods — what we call aspect *deployment* — is done in a separate module, the *connector*. The lower part of code fragment 3.2 shows a JAsCo connector deploying the abstract pointcut `connectionMethod(..args)` on the concrete method `getPrice()`.

#### 3.4.1.2 A More Sophisticated Event

Moreover, both AspectJ and JAsCo provide more advanced pointcut designators that allow expressing execution points that depend on more sophisticated dynamic properties of the application's execution, such as control flow. These more advanced pointcut designators can be used to realize more sophisticated dynamic events at which to trigger rules. Consider the example *event3* introduced in section 2.3.1 which denotes the moment in the execution of the core application after the product's price is retrieved while the customer is checking out. A solution in AspectJ for realizing this event consists of defining a second pointcut `priceCalcInCheckout(Product p)` that filters the joinpoints captured by the first pointcut `priceCalculation()` according to whether they occur in the control flow of execution of the method `checkoutShoppingBasket(ShoppingBasket)`. This is achieved by using the AspectJ's primitive pointcut designator `cflow`, as it is shown in code fragment 3.3. As a result the pointcut `priceCalcInCheckout(Product p)` only picks up those invocations to the method `getPrice()` that occur in the control flow of the method `checkoutShoppingBasket(ShoppingBasket)`.



JAsCo

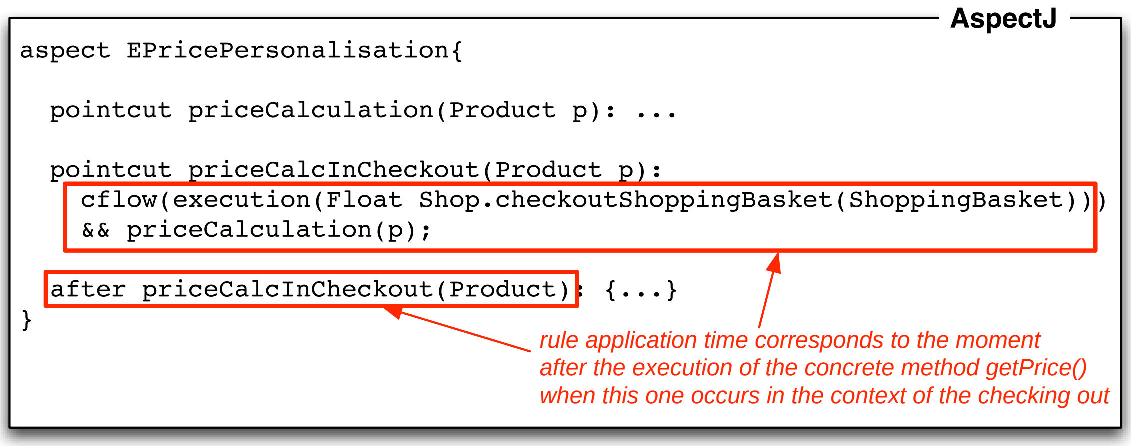
```
class BRConnection {
  hook BRConnectionHook {
    BRConnectionHook(connectionMethod(..args)) {
      execution(connectionMethod);
    }
    after()
      { //triggering rule application... }
  }
}
```

rule application time corresponds to the moment after the execution of a method

```
static connector BRDiscountConnector {
    BRConnection.BRConnectionHook hook0 =
        new BRConnection.BRConnectionHook(float Product.getPrice());
}
```

rule application time is made concrete

*Code Fragment 3.2:* JAsCo's solution for denoting a simple event capturing the moment after the invocation of the method `Product.getPrice()`: the upper part shows the abstract aspect bean whereas the lower part shows the concrete deployment

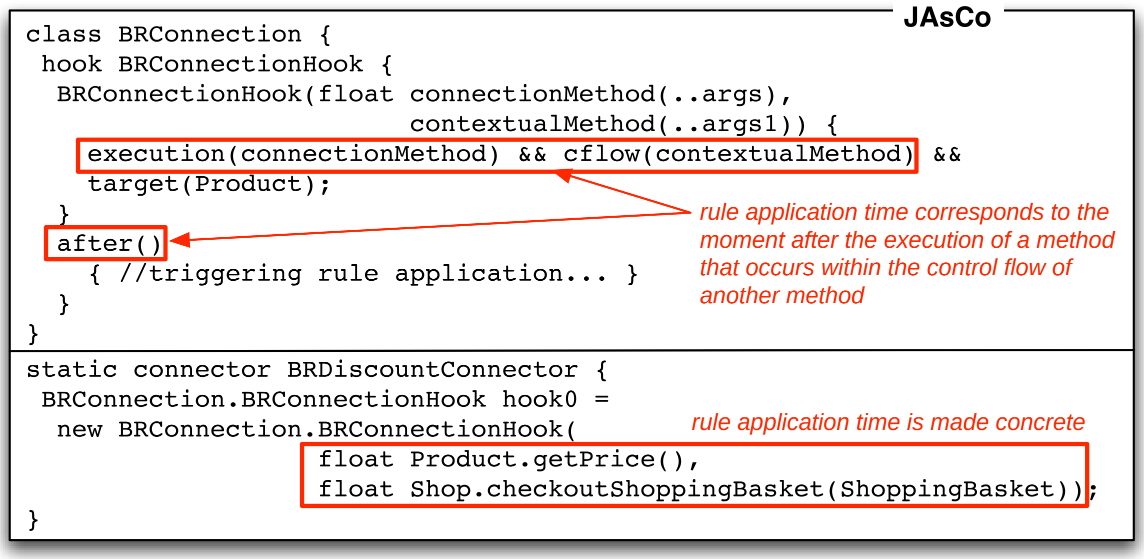


AspectJ

```
aspect EPricePersonalisation{

  pointcut priceCalculation(Product p): ...

  pointcut priceCalcInCheckout(Product p):
    cflow(execution(Float Shop.checkoutShoppingBasket(ShoppingBasket)))
    && priceCalculation(p);

  after priceCalcInCheckout(Product): {...}
}
```

rule application time corresponds to the moment after the execution of the concrete method getPrice() when this one occurs in the context of the checking out

*Code Fragment 3.3:* A solution in AspectJ for denoting a more complex event capturing the moment after the invocation of the method `Product.getPrice()` in the control flow of the `Shop.checkoutShoppingBasket(ShoppingBasket)` method



JAsCo

```
class BRConnection {
 hook BRConnectionHook {
  BRConnectionHook(float connectionMethod(..args),
                         contextualMethod(..args1)) {
    execution(connectionMethod) && cflow(contextualMethod) &&
    target(Product);
  }
  after()
    { //triggering rule application... }
  }
}
```

rule application time corresponds to the moment after the execution of a method that occurs within the control flow of another method

```
static connector BRDiscountConnector {
 BRConnection.BRConnectionHook hook0 =
  new BRConnection.BRConnectionHook(
                   float Product.getPrice(),
                   float Shop.checkoutShoppingBasket(ShoppingBasket));
}
```

rule application time is made concrete

*Code Fragment 3.4:* A solution in JAsCo for denoting a more complex event capturing the moment after the invocation of the method `Product.getPrice()` in the control flow of the `Shop.checkoutShoppingBasket(ShoppingBasket)` method

Similarly to AspectJ, JAsCo also supports the `cflow` pointcut designator which allows expressing more sophisticated dynamic events. A solution similar to the AspectJ's one is shown in code fragment 3.4. The upper part corresponds to the aspect bean. Note that `connectionMethod` and `contextualMethod` are abstract method calls that need to be deployed on the concrete methods `getPrice()` and `checkoutShoppingBasket(ShoppingBasket)`, as shown in the lower part of the code fragment 3.4.

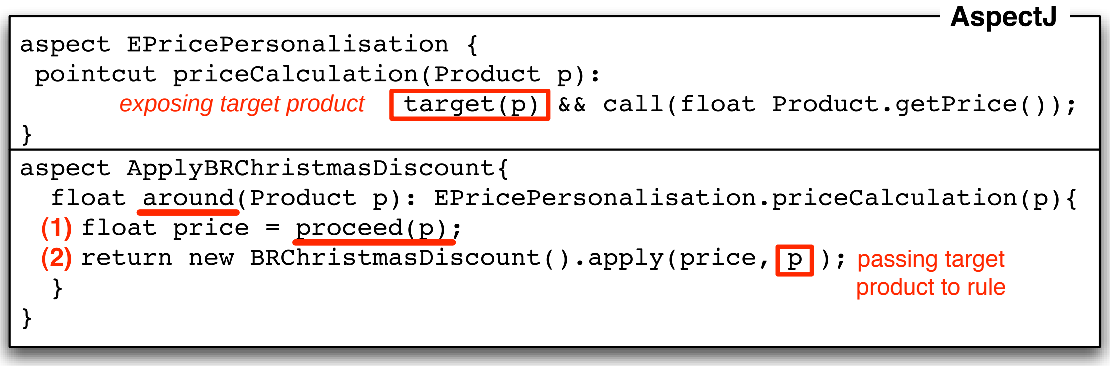
### 3.4.2 Exposing and Passing Available Contextual Information

Rules often require information which depends on the dynamic context. Therefore, it should be possible for a rule to access and manipulate the data available in the context of the event that triggered its execution. All investigated AOP approaches allow to introspect the context of the joinpoint that triggers the execution of the aspect behaviour. This is in fact a consequence of the *dependency inversion* property. For instance, for a joinpoint that corresponds to a method execution or call, one can query the name of the method, the supplied arguments and the target object on which the method is called. In many cases this expressive power suffices for providing the business rules with the necessary information. As an example consider varying the *BRChristmasDiscount* which applies the discount only on purchases of some product kind, for instance Christmas articles. This can be formulated in the following rule:

***BRChristmasDiscount:*** *If today is Christmas then apply a 5% discount on the purchases of Christmas articles*

Suppose this rule is triggered at the same *event2* after the price of a product is retrieved. This implies the need for checking an extra condition on the product that is being purchased. Therefore the product has to be obtained and passed to the rule at rule application time.

Pointcuts can expose the information directly available in the context of the execution point. In AspectJ the target object can be exposed using the **target** pointcut designator, as shown in the upper part of code fragment 3.5. In our example, the product is the target object and therefore can be directly exposed by the **priceCalculation(Product p)** pointcut.



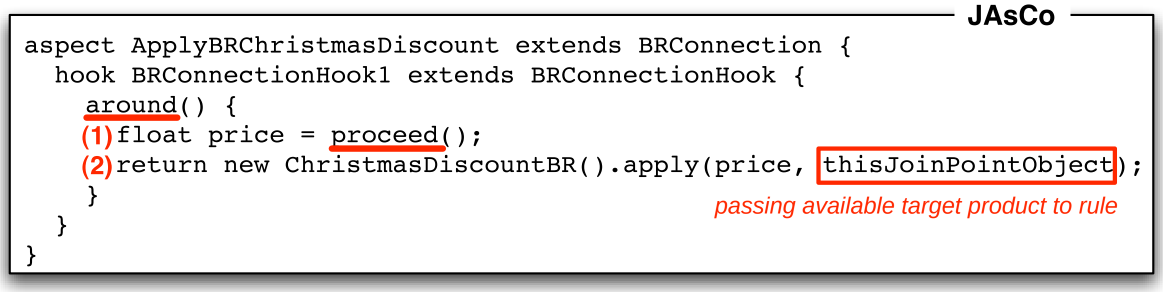
```
aspect EPricePersonalisation {
 pointcut priceCalculation(Product p):
                                 target(p) && call(float Product.getPrice());
}
aspect ApplyBRChristmasDiscount{
  float around(Product p): EPricePersonalisation.priceCalculation(p){
 (1) float price = proceed(p);
 (2) return new BRChristmasDiscount().apply(price, p );
  }
}
```

AspectJ
exposing target product
passing target
product to rule

*Code Fragment 3.5:* A solution in AspectJ for exposing (upper part) and passing to the rule (lower part) information available in the context of the triggering pointcut

This exposed information has to be passed to the rule. In both AOP approaches, the information exposed by a pointcut can be used in any advice defined on that pointcut. Therefore, the actual triggering of the rule has to be performed from within an advice defined on the pointcut capturing the rule application time and exposing the desired information available in that context. In AspectJ this can be done as shown in the lower part of code fragment 3.5. Note that now instead of an **after** advice, an **around** advice has to be defined on the pointcut **priceCalculation(Product p)**. This is because the original price — which is the return value of the interrupted method — has to be manipulated. This price needs to be obtained by explicitly resuming the original behavior which is done by calling **proceed(p)**. This is the first action of the advice which is followed by the actual triggering of the rule. Therefore, the rule is still applied 'after' the original behavior, even though an around advice is employed. The instance of **Product** which is the target object — exposed by this pointcut — is passed to the rule to be used in its condition. This around advice can be defined in the same aspect where the pointcut is defined, i.e. **EpricePersonalisation**, or in a separate aspect. The latter solution allows for a better reusability of rule connections, as a better separation of the different connection parts is achieved. In the lower part of code fragment 3.5 a separate aspect **ApplyBRChristmasDiscount** is shown which holds the definition of this around advice.

In JAsCo exposing information available in the context of the pointcut is done in a similar way as in AspectJ. The only difference is that both the arguments and the target object of the interrupted behavior are directly accessible from within any advice by referring to **args** and by using the **thisJoinPointObject** keyword respectively, contrary to AspectJ where they need to be explicitly captured by the pointcut. The JAsCo solution is shown in code fragment 3.6. The connector for this aspect bean deploys the aspect behavior in the same way as it is shown in code fragment 3.2 (therefore it is not shown here).



JAsCo

```
aspect ApplyBRChristmasDiscount extends BRConnection {
  hook BRConnectionHook1 extends BRConnectionHook {
    around() {
    (1)float price = proceed();
    (2)return new ChristmasDiscountBR().apply(price, thisJoinPointObject);
    }
  }
}
```

passing available target product to rule

*Code Fragment 3.6:* A solution in JAsCo for passing to the rule the information available in the context of the triggering pointcut

### 3.4.3 Capturing, Exposing and Passing Unavailable Information

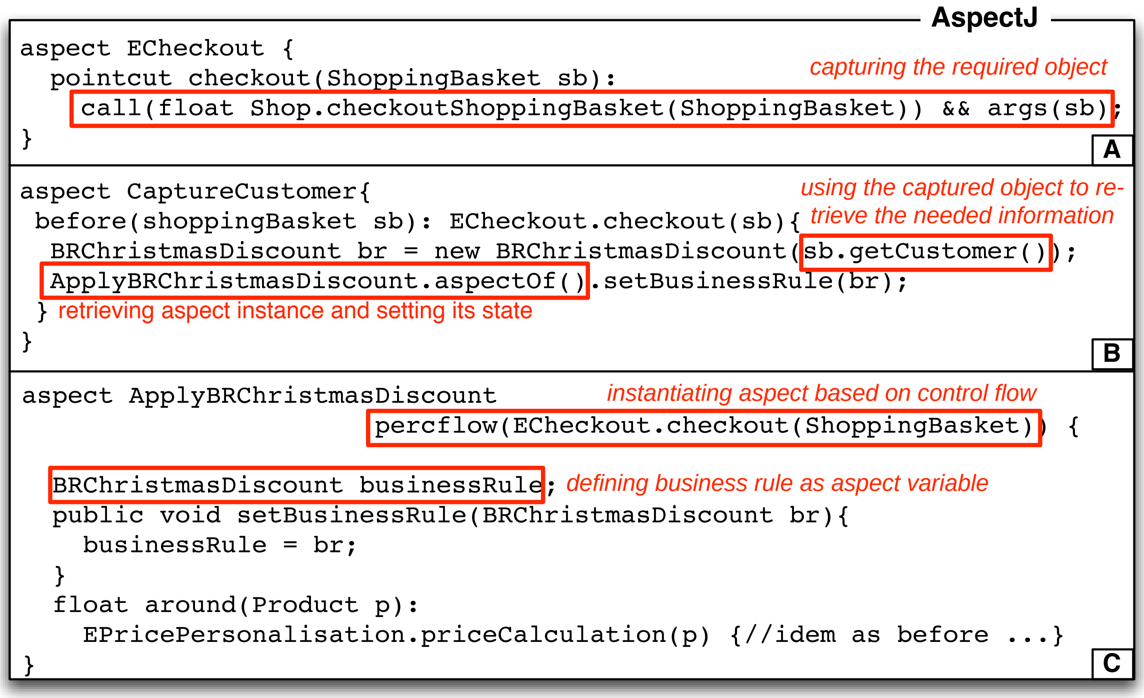
If the data objects required by the rule are outside the scope of the joinpoint that triggers the rule, other pointcuts can be defined in order to intercept the application at the moment the needed objects are available. Those pointcuts would then expose the objects available in their context. Once this is accomplished, the challenge is then to pass those captured objects to the event that triggers the rule, so that they can be passed to the rule.

Imagine a new variation of the *BRChristmasDiscount* rule that applies a discount to the product price which depends on the actual customer that is purchasing the product. This implies the need for having the customer object at rule instantiation time in order to obtain the discount percentage. However, this object is not available at the moment the product price is retrieved, the event at which the rule is instantiated and applied. A solution is to instantiate the `BRChristmasDiscount` at the moment the `checkoutShoppingBasket-(Shoppingbasket)` method is invoked, since the customer is available at that point. Still we want to do this non-invasively and therefore *dependency inversion* is fundamental. A possible solution in AspectJ consists of the definition of a set of new collaborating aspects, as follows:

First of all, a new aspect `ECheckout` defines a new `checkout` pointcut intercepting the calls to `checkoutshoppingBasket(ShoppingBasket)` and exposing the target shopping basket object (shown in part A of code fragment 3.7).

Secondly, a separate aspect named `CaptureCustomer` defines an advice on the `checkout` pointcut and uses the exposed shopping basket to create a new instance of the `BRChristmas-Discount` rule (shown in part B of code fragment 3.7). The separation of these two parts in different aspects aims at enhancing reusability of the different parts of the rule connections. Once the extra information is captured, it has to be made available at the event where the rule is applied. To this end, support for sharing the aspect context information is needed. In AspectJ however — as well as in other approaches such as AspectWerkz and JBoss/AOP — sharing information between several aspects is not straightforward. The `CaptureCustomer` aspect needs to pick the correct aspect instance of the `ApplyBRChristmas-Discount` class to be associated with the correct business rule instance. This is done by invoking `ApplyBRChristmasDiscount.aspectOf()` (shown in part B of code fragment 3.7).

Finally, a new instance of the aspect `ApplyBRFrequentDiscount` is needed for each joinpoint denoted by the pointcut `checkout(ShoppingBasket)`. This is achieved by using the `percflow` feature which instantiates an aspect for each join point designated by the pointcut it takes as parameter (shown in part C of code fragment 3.7). Therefore the two involved aspects need to collaborate in order for the rule to be successfully applied. In AspectJ the collaboration between aspects is implicitly represented in the implementation of the involved aspects themselves. Typically, when aspect needs to communicate with another aspect, the former has to explicitly grab the desired aspect instance[2]. This is done by referring to the aspect class and invoking the method `aspectOf()` of it (as shown in part B of code fragment 3.7). As a consequence, it is not trivial to understand how aspects collaborate by simply looking at the aspect code. This is especially the case when complex (and even nested) control flows are involved, since the developer is forced to have a good understanding of the order in which the methods — involved in the control flows — are executed, and the impact this order has on the way aspects are instantiated. As a consequence of the implicit way aspects are related, it becomes hard to write and understand AspectJ's aspects. This AspectJ's limitation is explained in more detail in section 3.4.8.



AspectJ

```
aspect ECheckout {
  pointcut checkout(ShoppingBasket sb):
    call(float Shop.checkoutShoppingBasket(ShoppingBasket)) && args(sb);
}
```

capturing the required object

A

```
aspect CaptureCustomer{
 before(shoppingBasket sb): ECheckout.checkout(sb){
  BRChristmasDiscount br = new BRChristmasDiscount(sb.getCustomer());
  ApplyBRChristmasDiscount.aspectOf().setBusinessRule(br);
 }
}
```

using the captured object to retrieve the needed information

retrieving aspect instance and setting its state

B

```
aspect ApplyBRChristmasDiscount
                        percflow(ECheckout.checkout(ShoppingBasket)) {

  BRChristmasDiscount businessRule;
  public void setBusinessRule(BRChristmasDiscount br){
    businessRule = br;
  }
  float around(Product p):
    EPricePersonalisation.priceCalculation(p) {//idem as before ...}
}
```

instantiating aspect based on control flow

defining business rule as aspect variable

C

*Code Fragment 3.7:* A solution in AspectJ for capturing information available at events other than the triggering event and making it available to the rule

JAsCo has the advantage of being able to gather collaborating aspects — the hooks — in an extra module — the aspect bean. Furthermore, an aspect bean can hold the definition of information — structure and behavior — which is shared among its hooks. This facilitates sharing information among aspects. In our example, two hooks can be defined, one for

[2] By default there is only one instance of an aspect; many instance can exist and even co-exist depending on the strategy used to instantiate the aspects (such as percflow, perobject, etc.).

capturing the customer and another one for applying the rule, as part of the same aspect bean. The `businessRule` variable is defined in the aspect bean and therefore shared among the two hooks. This simplifies aspect communication since it avoids having to explicitly obtain the right aspect instances by querying the aspect classes. Moreover, related hooks can be instantiated in the same connector which ensures the correspondence between their instances.

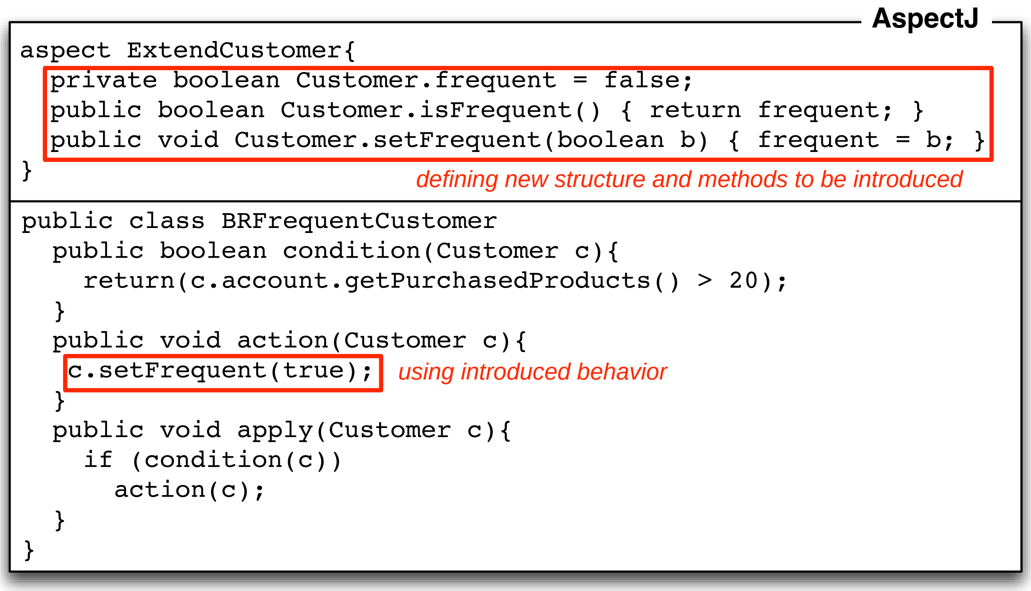
### 3.4.4 Introducing Unanticipated Information

Some rules require information that is not present in the existing application. To this end, the introduction of new data and behavior is crucial (third fundamental AOP characteristic, as listed in 3.3.1). Several AOP approaches support a mechanism to introduce new structure and behavior. For instance, the *open classes* feature (previously named *introductions*) supported by AspectJ allows the insertion of attributes and methods. It also allows extending classes from specific superclasses and interfaces from specific superinterfaces. Consider again the following rule:

***BRFrequentCustomer****: If a customer has purchased more than 20 products then he or she is a* frequent *customer*

This rule expects the class `Customer` to have a boolean attribute `frequent` and to be able to set that attribute to a new value. In order to realize this extension, an AspectJ aspect `ExtendCustomer` is defined which introduces the unanticipated attribute `frequent` to the `Customer` class and the methods `isFrequent()` and `setFrequent(boolean)`. This is shown in the upper part of code fragment 3.8. The introduced structure and behavior can be used in other classes, in this case in the `BRFrequentCustomer` class, as shown in the lower part of code fragment 3.8. An aspect for the application of this rule must trigger the application of the rule after the event designated by the `ECheckout.checkout(ShoppingBasket)` pointcut using the shopping basket object exposed by this pointcut (not shown here).

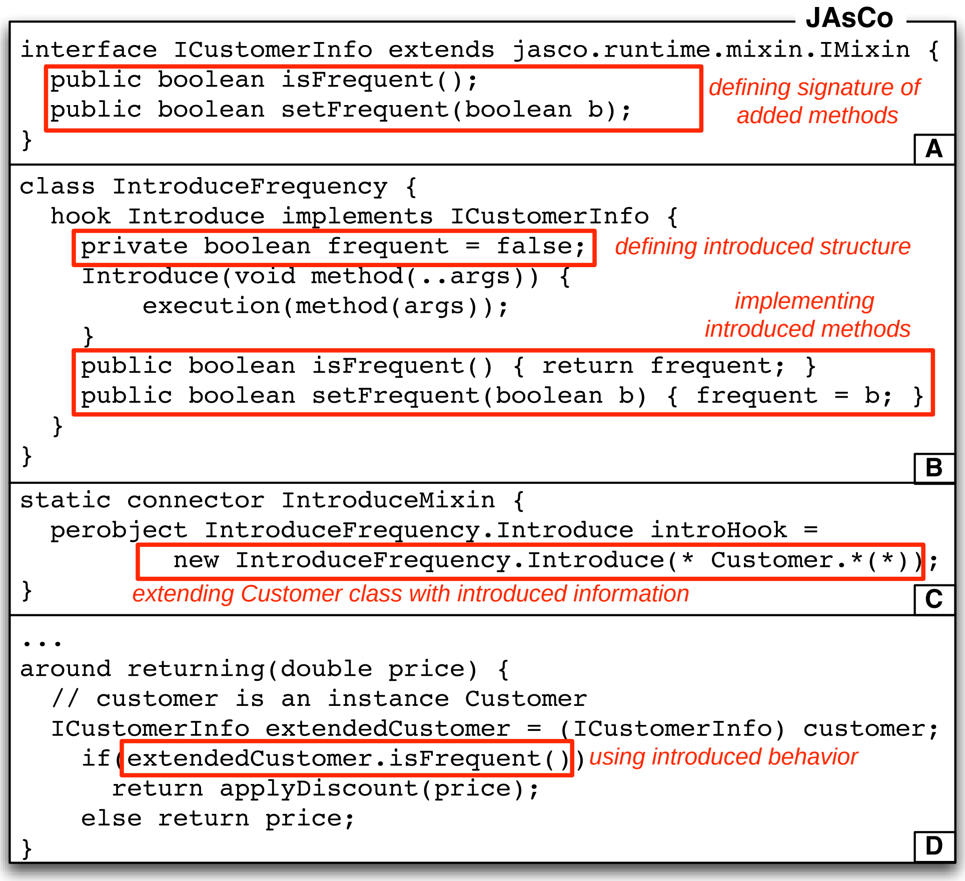
Other approaches support introducing new behavior to the context of an aspect by forcing it to implement an interface. This is the case in approaches such as JAC, JBoss/AOP and JAsCo. A mixin class is provided that handles the implementation of a new interface and is automatically attached to the concerned classes. The new methods can be invoked from other aspects. Therefore, this feature is useful for communicating information between aspects. A solution using JAsCo's virtual mixins is shown in code fragment 3.9: an interface defining the methods to be added to the `Customer` class is defined in part A; an implementation for this interface is provided by the hook `Introduce`, shown in part B; note that the hook defines an attribute `frequent` which holds a boolean value. Contrary to AspectJ's introductions that allow extending a target class with not only behavior but also structure, JAsCo virtual mixins can only add behavior. The state — the `frequent` variable — is kept in the aspect instead of in the target class. In order to actually introduce the two methods defined in the `Introduce` hook to all the `Customer` objects, the `Introduce` hook must be instantiated `perobject` of the class `Customer`, as it shown in the connector named `IntroduceMixin` shown in part C. This instantiation makes sure that a different hook exists per customer, ensuring that a different `frequent` variable exists per customer. Finally, the added behavior can now be invoked from other aspect code as shown in part D.


AspectJ
defining new structure and methods to be introduced
using introduced behavior

```
aspect ExtendCustomer{
  private boolean Customer.frequent = false;
  public boolean Customer.isFrequent() { return frequent; }
  public void Customer.setFrequent(boolean b) { frequent = b; }
}
```

```
public class BRFrequentCustomer
  public boolean condition(Customer c){
    return(c.account.getPurchasedProducts() > 20);
  }
  public void action(Customer c){
    c.setFrequent(true);
  }
  public void apply(Customer c){
    if (condition(c))
      action(c);
  }
}
```

*Code Fragment 3.8:* A solution in AspectJ based on open classes for introducing structure and behavior


JAsCo
defining signature of added methods
A
defining introduced structure
implementing introduced methods
B
extending Customer class with introduced information
C
using introduced behavior
D

```
interface ICustomerInfo extends jasco.runtime.mixin.IMixin {
  public boolean isFrequent();
  public boolean setFrequent(boolean b);
}
```

```
class IntroduceFrequency {
  hook Introduce implements ICustomerInfo {
    private boolean frequent = false;
    Introduce(void method(..args)) {
        execution(method(args));
    }
    public boolean isFrequent() { return frequent; }
    public boolean setFrequent(boolean b) { frequent = b; }
  }
}
```

```
static connector IntroduceMixin {
  perobject IntroduceFrequency.Introduce introHook =
        new IntroduceFrequency.Introduce(* Customer.*(*));
}
```

```
...
around returning(double price) {
  // customer is an instance Customer
  ICustomerInfo extendedCustomer = (ICustomerInfo) customer;
    if(extendedCustomer.isFrequent())
      return applyDiscount(price);
    else return price;
}
```

*Code Fragment 3.9:* A solution in JAsCo based on virtual mixins to introduce new behavior to core classes

Although JAsCo mixins allow realizing the extension of classes with new methods, they have the limitation that added methods can only be invoked from within aspect code. Thus, in order for the rule to be triggered, the condition and action methods must be invoked directly from the aspect's advice (instead of invoking the `apply` method on the rule).

### 3.4.5 Incorporating Rule Results

All analyzed AOP approaches can be used in a similar way in order to resume the original application after the application of rules, using — when needed — the results produced by the rules. In the case of an around advice, the interrupted execution can be continued by employing the `proceed` keyword. In case other aspects are also applicable to the same joinpoint, the `proceed` actually triggers the next aspect's around advice, this way realizing a chain of around advices which ends at the original replaced method. In some AOP approaches such as JAsCo, the `proceed` keyword can take parameters which represent the target object and parameters that need to be considered when proceeding with the original interrupted method. Therefore, in order to change the original behavior, the proceed can be invoked using other objects that result from having applied the rule. In the case a rule is triggered before or after an event, rules can affect the core application by invoking methods that change the state of core objects, such as `Shop.increaseStock(Product, int)`. When the execution of the aspect is finished, the core application is resumed. No special keywords are needed for doing so — contrary to the case of an around advice.

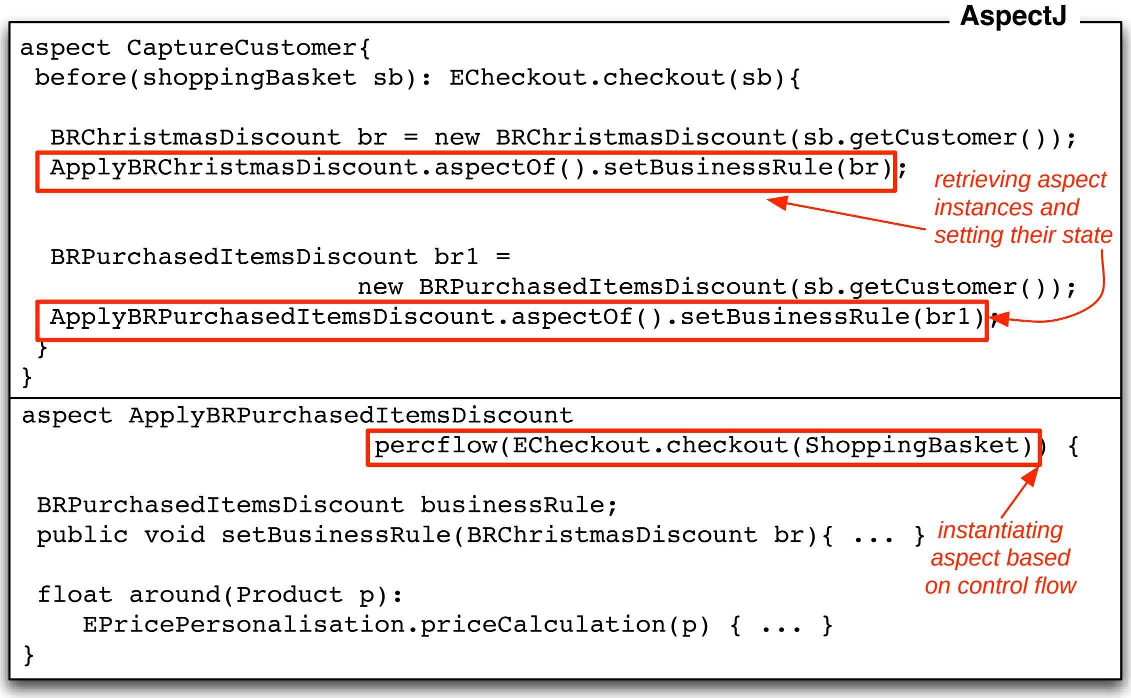
### 3.4.6 Configuring and Reusing Rules and Their Connections

As we mentioned before, we pursue maximum configurability by separating each of the parts that form the connectivity layer in different aspects. This makes it possible to reuse each part of the rule connection separately. For instance, a new rule might appear that needs to be connected at the same event at which the *BRChristmasDiscount* is applied, for instance:

***BRPurchasedItemsDiscount:*** *if the customer has bought more than 10 products then he or she gets a 10% discount on the current purchase.*

Assume the existence of a `BRPurchasedItemsDiscount` class implementing this new rule. In order to connect this rule at *event2*, an aspect — named for instance `ApplyBRPurchasedItemsDiscount` — must be defined that triggers the application of the rule at the same pointcut `EPricePersonalisation.priceCalculation(p)`. The before advice of the `CaptureCustomer` aspect (part B of code fragment 3.7) has to be extended to also instantiate the `BRPurchasedItemsDiscount` using the captured customer. Moreover, the `ApplyBRPurchasedItemsDiscount` must be instantiated `percflow` on the `ECheckout.checkout(ShoppingBasket)` pointcut. The code implementing these extensions is shown in code fragment 3.10. This example shows that it is possible to connect different rules at the same event. In the case the new rule connected at the existing event requires extra information not exposed by the pointcut implementing that event, the pointcut can be extended to expose the extra needed information (if it is available in that context); otherwise, the corresponding `Capture` aspect can be extended with extra pointcuts exposing the required information, as before.

The other way round is also possible: we might want to connect the same *BRChristmasDiscount* at a different event, as for instance at *event3*. This is simply achieved by triggering the application at the `EPricePersonalisation.priceCalcInCheckout` pointcut instead of the `EPricePersonalisation.priceCalculation(p)` pointcut. Eventually the kind of the



```
aspect CaptureCustomer{
 before(shoppingBasket sb): ECheckout.checkout(sb){

  BRChristmasDiscount br = new BRChristmasDiscount(sb.getCustomer());
  ApplyBRChristmasDiscount.aspectOf().setBusinessRule(br);

  BRPurchasedItemsDiscount br1 =
                      new BRPurchasedItemsDiscount(sb.getCustomer());
  ApplyBRPurchasedItemsDiscount.aspectOf().setBusinessRule(br1);
 }
}
aspect ApplyBRPurchasedItemsDiscount
                       percflow(ECheckout.checkout(ShoppingBasket)) {

 BRPurchasedItemsDiscount businessRule;
 public void setBusinessRule(BRChristmasDiscount br){ ... }

 float around(Product p):
    EPricePersonalisation.priceCalculation(p) { ... }
}
```

retrieving aspect instances and setting their state
instantiating aspect based on control flow

*Code Fragment 3.10:* A solution in AspectJ for applying the `BRChristmasDiscount` and the `BRPurchasedItemsDiscount` rules at the same event: the upper part shows the extensions to the `CaptureCustomer` aspect in order to make the captured customer available to the instance of `ApplyBRPurchasedItemsDiscount`; the lower part shows the aspect for applying the `BRPurchased-ItemsDiscount` on the `EPricePersonalisation.priceCalculation` pointcut advice that triggers the rule application might change as well (even though it is not the case in this example). These changes are done in the corresponding `Apply` aspect.

Decoupling the different parts of a rule connection in different aspects improves reusability but at the same time implies the need for adequately composing the several aspects together in order to achieve the correct behavior. Thus, this might require significant coordination among aspects. In AspectJ, limited support for expressing the relations between aspects is provided. To illustrate this it is enough to look at the `ApplyBRChristmasDiscount` aspect (lower part of code fragment 3.5). This aspect needs to refer to another aspect — the `ECheckout` — in order to specify how it must be instantiated, introducing a dependency between the two: if the second aspect changes its name, or its pointcut implementation, the first aspect might become invalid. This is because in AspectJ most relations between aspects are implicitly specified in the same aspect code (an exception is the *precedence* relation available from AspectJ 1.1, which replaces the original *dominates* feature). As a consequence, the coordination and synchronization between aspects are also implicitly specified. The software engineer is responsible for controlling and relating the aspects manually, typically by hard-coding the references and dependencies in the aspect code itself.

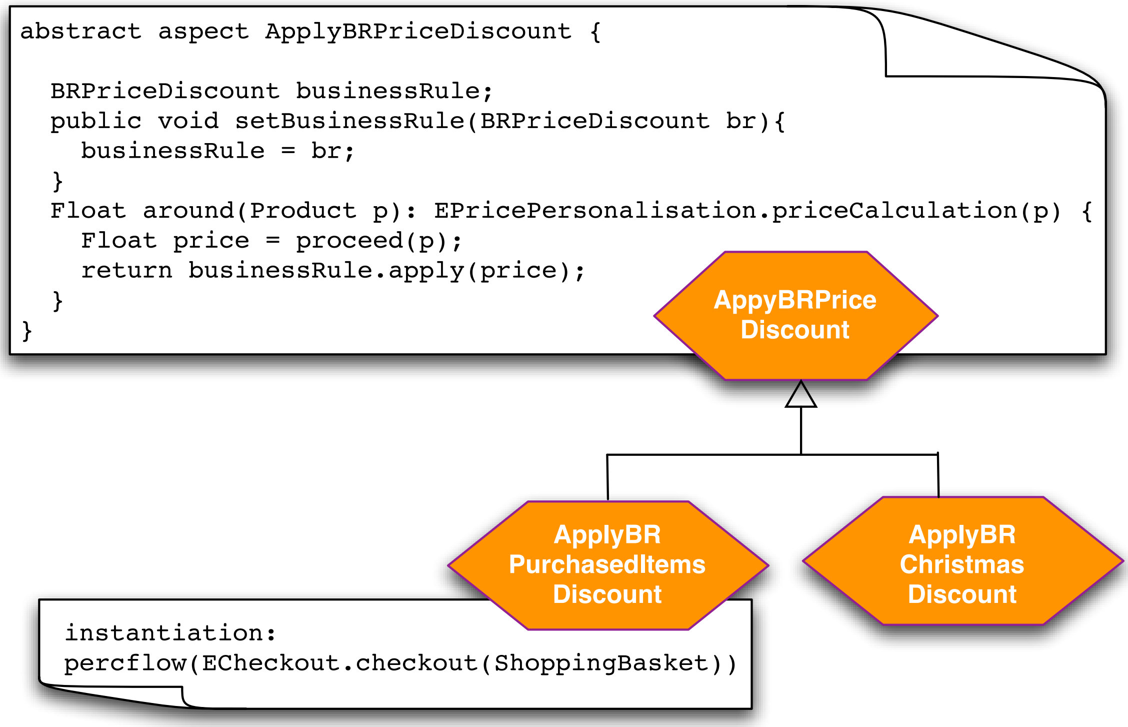
In JAsCo, applying the `BRPurchasedItemsDiscount` at the same event as `BRChristmas-Discount` would imply defining a new aspect bean extending `BRConnection` (analogous to the one for triggering the `BRChristmasDiscount` shown in code fragment 3.6) and a corresponding connector in order to trigger the application of the `BRPurchasedItemsDiscount` at the same *event2*. It is also required that `BRConnection` defines a second hook which captures and stores the customer object as a global variable of the aspect bean (as explained in section 3.4.3), which is then inherited by the concrete subaspects extending `BRConnection`. The captured object is then used in the triggering hooks in order to instantiate both `BRChristmasDiscount` and `BRPurchasedItemsDiscount` rules respectively.

Both approaches allow reusing aspect code though inheritance, which helps reusing rule connections as well. For instance, we observe that the around advice in both AspectJ's aspects `ApplyBRChristmasDiscount` and `ApplyBRPurchasedItemsDiscount` share common parts. Therefore, it is possible to pull up those parts in an abstract aspect `ApplyBR-PriceDiscount`. This is shown in Figure 3.2. Both concrete aspects `ApplyBRChristmas-Discount` and `ApplyBRPurchasedItemsDiscount` extend this abstract aspect, inheriting this way the `around` advice. In JAsCo, inheritance of aspects is achieved in a similar way and therefore it is omitted here.

### 3.4.7 Controlling Rule Precedence, Combination and Exclusion

An important part of the rule connection is the specification of how to combine several rules that are triggered at the same events. This is crucial to avoid rule interference. This is a well-known issue in AOP which is referred to as *feature interaction problem*[3] [PSC+02; BMV02; DFS02; NBA04; KPRS01]. AOP approaches tackle this problem is different ways.

[3] In the last years this problem has received special attention in the community, as demonstrated for example by the workshop on Aspects, Dependencies, and Interactions organized as part of ECOOP'06



```
abstract aspect ApplyBRPriceDiscount {

  BRPriceDiscount businessRule;
  public void setBusinessRule(BRPriceDiscount br){
    businessRule = br;
  }
  Float around(Product p): EPricePersonalisation.priceCalculation(p) {
    Float price = proceed(p);
    return businessRule.apply(price);
  }
}
```

```
instantiation:
percflow(ECheckout.checkout(ShoppingBasket))
```

*Figure 3.2:* AspectJ's solution illustrating the inheritance of aspects for the application of rules

In AspectJ, an aspect may declare a precedence relationship between concrete aspects with the declare precedence form:

*declare precedence : TypePatternList;*

This signifies that if at any join point, advices of the same kind are encountered — belonging to aspects that are matched by the TypePatternList — then the order in which those advices are executed is determined by the order in which the aspects are listed in TypePatternList.

In our previous example, both rules *BRChristmasDiscount* and *BRPurchasedItemsDiscount* are triggered at the same pointcut. The declare feature can be used for instance to specify a certain order in which discount rules must be applied, as follows:

```
aspect Ordering {
  declare precedence : ApplyBRChristmasDiscount, ApplyBRPurchaseDiscount;
}
```

This ensures that whenever a joinpoint occurs that triggers the application of both aspects, the around advice of `ApplyBRChristmasDiscount` is executed before the one of `ApplyBRPurchaseDiscount`. In AspectJ, the precedence clause is the only support for aspect combination.

A more fine-grained approach to aspect precedence is the one where the precedence is not defined at the level of the entire aspect but at the level of the advice. This is supported by AOP approaches such as JAC, JBoss/AOP and AspectWerkz. These approaches allow specifying explicit sequences of aspect deployments by means of stacks. Whenever a joinpoint is encountered, the deployed aspects are executed in the order specified by the stack. JAsCo enhances this support as it provides a powerful, reusable and extensive system for specifying the precedence and the combination of aspects. Whenever two or more hooks interfere, the order in which their behaviour must be executed is derived from the connector. This is useful to specify the order in which business rules must be triggered.

Being able to specify the sequence in which the various business rules are executed is in many cases not sufficient. In some cases more advanced techniques to specify the combination of the various business rules that are deployed within the system is required. In the previous section for instance, an additive discount strategy is employed. However, the business policy could specify that only one discount is offered for a given product. For instance we could restrict the application of the frequent customer to the period outside Christmas. The JAsCo language provides a solution to be able to specify this kind of advanced aspect-combinations, by providing a mechanism called combination strategies. A combination strategy acts like a kind of filter that validates the list of applicable hooks, which are obtained at run-time. Each specific combination strategy implements the `CombinationStrategy` interface which defines the `public HookList validateCombinations(HookList aHookList)` signature. The interface itself only specifies the `validateCombinations` method, which is used to describe the specific logic of a combination strategy. This mechanism of combination strategies allows maximum flexibility, as user-defined relationships between the various aspects can be implemented.

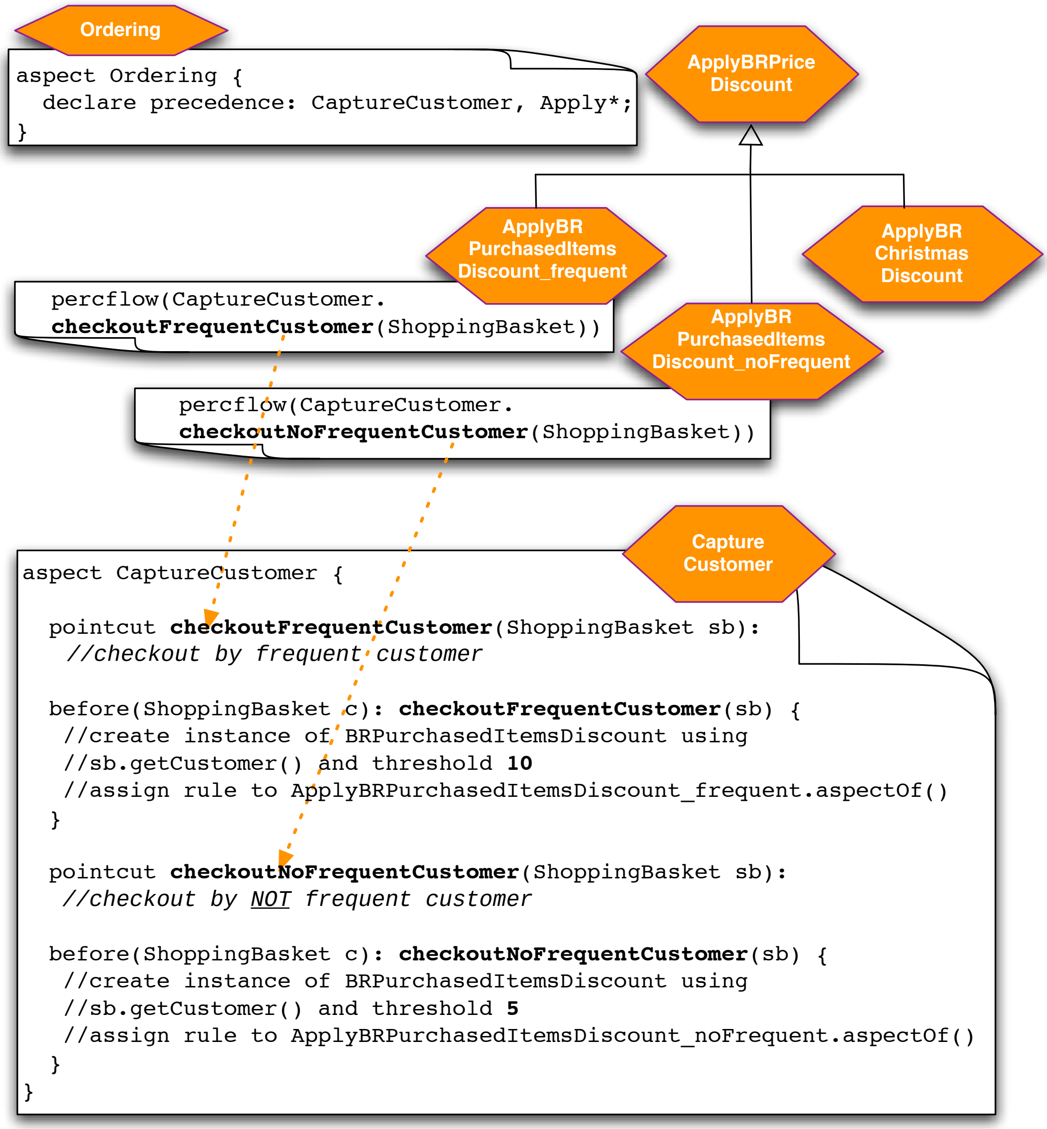
For instance, an exclude combination strategy can be defined which makes sure that a certain hook, e.g. *hookB*, is not executed whenever *hookA* is encountered. Such a combination strategy can be used to specify the relationship between the Christmas and the frequent customer discount business rules for instance to avoid the application of the latter whenever the first one is triggered. This is done by instantiating the exclude combination strategy giving as parameters the hooks in charge of triggering the application of the corresponding rules and adding this strategy to the connector. This is shown in the following lines of code (consider that `chDiscount` and `frDiscount` are the names of the hook variables for the Christmas and frequent rules respectively and that `ExcludeCombinationStrategy` is the name of the class implementing the exclude combination strategy):

```
connector ChristmasFrequentCustomerDiscountDeployment {
  ...
  ExcludeCombinationStrategy strategy =
    new ExcludeCombinationStrategy(chDiscount,frDiscount);
  addCombinationStrategy(strategy);
}
```

### 3.4.8 Controlling Rule Instantiation and Initialization

As the rules themselves are defined as reusable as possible, it is required to customise the rules towards the specific environment in which they are being applied. This implies the need for having control and being able to customize the instantiation of rule connection aspects. Most aspect-oriented technologies however do not allow sophisticated control for initializing aspects, as the aspect instantiation is done implicitly when the aspect is woven into the core functionality.

In AspectJ, as well as in AspectWerkz, aspects are not explicitly instantiated with `new` expressions. Rather, aspect instances are automatically created and controlled by the aspect framework. By default, an aspect is a singleton. However, other ways of instantiating an aspect are possible by using special keywords. An aspect `A` can be defined `percflow(Pointcut)` or `percflowbelow(Pointcut)`, meaning that an object of 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 object of type `A`. An instance of the aspect is created upon entry into each such control flow. The aspect instantiation mechanism based on pointcuts allows creating instances at very fine-grained points in the execution of the core application. To illustrate this consider a slight variation of the previous example rule *BRPurchasedItemsDiscount* in which we assume that the threshold is not fixed at 10, but instead it can vary according to the customer's frequency, as follows: if the customer is frequent, the threshold must be 5 whereas if the customer is not frequent, the discount must be applied only when at least 10 products have been bought. In order to implement this change, two concrete subaspects must be defined distinguishing between the application of the rule for a frequent and non frequent customer, which are shown in Figure 3.3. Two different pointcuts are defined in the `CaptureCustomer` aspect which capture the invocations to the method `checkoutShoppingBasket(ShoppingBasket)` but distinguish between the cases of shopping baskets belonging to frequent and non-frequent customers respectively. These pointcuts are used in order to initialize the two different `Apply` aspects using the `percflow` feature. This is needed because the aspects trigger rule objects that differ in state, namely the customer and the threshold.


Ordering
ApplyBRPrice Discount
ApplyBR PurchasedItems Discount_frequent
ApplyBR Christmas Discount
ApplyBR PurchasedItems Discount_noFrequent
Capture Customer

```
aspect Ordering {
  declare precedence: CaptureCustomer, Apply*;
}
```

```
percflow(CaptureCustomer.
checkoutFrequentCustomer(ShoppingBasket))
```

```
percflow(CaptureCustomer.
checkoutNoFrequentCustomer(ShoppingBasket))
```

```
aspect CaptureCustomer {

  pointcut checkoutFrequentCustomer(ShoppingBasket sb):
   //checkout by frequent customer

  before(ShoppingBasket c): checkoutFrequentCustomer(sb) {
   //create instance of BRPurchasedItemsDiscount using
   //sb.getCustomer() and threshold 10
   //assign rule to ApplyBRPurchasedItemsDiscount_frequent.aspectOf()
  }

  pointcut checkoutNoFrequentCustomer(ShoppingBasket sb):
   //checkout by NOT frequent customer

  before(ShoppingBasket c): checkoutNoFrequentCustomer(sb) {
   //create instance of BRPurchasedItemsDiscount using
   //sb.getCustomer() and threshold 5
   //assign rule to ApplyBRPurchasedItemsDiscount_noFrequent.aspectOf()
  }
}
```

*Figure 3.3:* A solution in pseudo-AspectJ code for the application of the `BRPurchasedItemsDiscount` according to whether a customer is frequent of not

Other AspectJ keywords allow having an instance of the aspect `perthis(Pointcut)` and `pertarget(Pointcut)` that create an instance of the aspect per-executing object or per-target object respectively.

Likewise to adding and removing aspects, altering properties of aspects at run-time is also a desired feature when they represent volatile rule integration code. Altering properties is in most approaches as simple as invoking methods defined in the aspects. However, in order to be able to invoke methods, the aspects have to be found first. This uncovers a fundamental AOP feature that is the reference of aspect instances, which is related to the *aspect interaction* characteristic listed in section 3.3.1. In AspectJ, it is only possible to fetch an aspect instance by name. This is not always a good solution as the name of the actual aspect class that got instantiated might not be known until run time. AspectWerkz allows fetching aspects on a per joinpoint basis, but requires obtaining every joinpoint by name. Fetching all aspects disregarding the concrete joinpoint they are attached to is not possible.

JAsCo and JBoss/AOP are the only two reviewed approaches that allow to explicitly instantiate and initialize aspects and also allow fetching all aspects in the system. In JAsCo the instantiation of an aspect with a specific context is described explicitly in the connector. The connector also allows customising the instantiated aspects and supports the full expressiveness of the Java language to this end. Also, the execution of the behavior of the business rules is specified explicitly in the connector, allowing even more finely grained control. JBoss/AOP introduces the novel concept of aspect factories, allowing fine-grained control over aspect instantiation. Aspect customisation happens through an XML connector that describes the deployment details. This XML file allows specifying a set of properties that are passed as input for aspect initialization. The aspect itself is responsible for parsing the XML tree, which makes it somewhat more cumbersome. In contrast to JAsCo connectors, no static type checking is possible for these XML property definitions.

### 3.4.9 Connecting Rules

Rules constantly evolve to cope with changes in the business requirements, other rules become obsolete and new ones are added. Thus, the aspects that encapsulate their links should be pluggable at run-time to reflect that volatility. Not all existing AOP technologies however allow easy and dynamic plugging in and out business rules. Approaches such as HyperJ only allow to statically plug in and out aspects, i.e. aspects can only be added at compile-time and it is not possible to plug them in or out at run-time. This is mainly because an aspect loses its identity when it is woven into the base-application. Approaches like JAC and JAsCo solve this issue, by also providing a run-time separation between the aspects and the base implementation of the system. This way, aspects remain first class entities even at run time and their logic is not weld together with the base functionality of the application. This is a valuable property in the context of business rules, as this run-time separation allows dynamic reconfiguration of business rules, without the need to shut down business-critical applications.

Approaches like AspectJ, AspectWerkz and JBoss/AOP provide support for adding and removing aspects at load-time and some support for their addition and removal at run-time. In the last two approaches, an XML "connector" is employed for connecting the aspects to concrete joinpoints. However, this XML connector cannot be employed any longer during run time and aspects have to be attached and removed programmatically. Because both approaches rely on traps at every joinpoint for aspect execution, aspects can only be added at joinpoints where traps are placed. In AspectWerkz, these traps are only inserted at joinpoints where aspects are applied at start-up time of the application. As such, only at those joinpoints, aspects can be attached and removed. In JBoss/AOP, it is possible to declare joinpoints as advisable in the XML connector. Even though no aspects are applied, a trap is still installed and aspects can be dynamically attached at those advisable joinpoints.

## 3.5 Discussion

Aspects are meant to encapsulate the implementation of crosscutting concerns and as such seem suited to modularize the crosscutting rule connections. We observe that all analyzed approaches offer features that can be used to accomplish the identified requirements and — even though sometimes the solutions are fundamentally different — we can conclude that the supported AOP features are very well-suited to accomplish the distilled requirements. We were able to express suitable solutions in both concrete representative approaches, and we can point out from them the following main conclusions:

AspectJ allows decoupling the different parts that constitute the rule connection in separate aspects that can be reused independently. However, this results in a proliferation of aspects which is hard to manage. Generally aspect relations are expressed in the same aspects that are being related (an exception is the explicit precedence relation) which reduces aspect reusability and composability. On the other hand, reusability of aspect code is possible through inheritance. Moreover, we observe that AspectJ has some very powerful and low-level features that are used for solving a wide range of problems, for example percflow. However, sometimes the same features are used to solve semantically different concerns, thus impeding program understandability and portability. AspectJ's pointcuts are fragile as they directly point to concrete places in the core application's execution, and therefore less reusable than JAsCo's. Additionally, instantiation and initialization is controlled by the framework itself which can be an advantage — like in situations where the instantiation depends on complex pointcuts — but can also be restrictive when more controlled instantiation is desired.

We observe that JAsCo is able to improve on AspectJ for connecting business rules on several essential points. First of all, JAsCo offers higher-level abstractions to express the composition of aspects which allows specifying more complex interrelations between rules. JAsCo allows specifying reusable business rules that can be dynamically plugged in and out to fit the application at hand. Secondly, the connector concept of JAsCo allows controlling in a more detailed way the instantiation and initialization of the business rules. An additional advantage of the connector is that it allows specifying and managing more advanced and fine-grained business rule combinations than in AspectJ. Whereas the introduction of new structure is not possible in JAsCo, virtual mixins can be used to extend core classes with additional methods.

We have also analyzed other emergent AOP approaches against the requirements. Other dynamic AOP approaches, such as AspectWerkz and JBoss/AOP allow adding and removing rules at run time, which is an essential requirement due to the volatile nature of business rules. In addition, these approaches make use of a separate connector concept, which allows separating the identification of an event and the application of rules upon those events. As a result, rules can be instantiated explicitly, customized towards the context upon which they are being applied and their mutual interaction can be managed. However, some dynamic AOP approaches might induce a rather big performance penalty at run-time and their joinpoint model is at the moment less expressive than the ones provided by their static counterparts. Although static AOP approaches, such as AspectJ, do not allow the dynamic pluggability of rule integration code, they provide a more fine-grained description of the events upon which rules can be applied. In addition, these approaches allow introducing unanticipated data required by rules quite easily in the application at hand.

Another way of comparing the different AOP approaches is with respect to how aspects are treated both at compile and run time. In JAsCo, JAC, AspectWerkz and JBoss/AOP aspects are implemented as fully independent modules. They are completely independent and reusable entities. Even at run-time, the aspects remain first-class entities independent from the core functionality. AspectJ supports load time weaving of aspects. This allows aspects to be compiled separately as no details about the core application on which the aspects are to be woven are needed at compilation time. In HyperJ aspects are physically woven into the core functionality, embedding the advices in the base behaviour. This makes the aspect again crosscutting at run-time. In this latter case, aspects loose their identity at run-time and it is in principle impossible to refer to the aspect entity directly. In addition, when the aspect logic has to be altered, the complete application has to be rewoven; this gives raise to scalability issues when a multitude of aspects are present in large scale applications. To conclude, in JAsCo, JAC, AspectWerkz, JBoss/AOP, aspects can change independently and reflect those changes directly in the core functionality, without the need to be reintegrated; it suffices to recompile the aspects. In AspectJ, changes in the aspect logic are reflected by recompiling and reloading the aspects.

## 3.6 Summary

In this chapter we pointed out which are the fundamental AOP characteristics that are required in order to successfully decouple and modularize crosscutting rule connections. Besides being able to encapsulate and declare crosscutting behavior in a localized and explicit way, AOP features are also suitable to accomplish the extra set of requirements identified in chapter 2. First of all we show this in a general way by distilling the fundamental AOP features that are needed to accomplish the identified requirements. Second, a more concrete contribution is to show how representative AOP approaches — namely JAsCo and AspectJ — actually achieve this.

We observe while carrying out the experiments that it is difficult to generalize the results to all aspect-oriented approaches because the concrete features these approaches provide are sometimes fundamentally different. However, these experiments let us observe that, independently of the concrete mechanisms and features, an AOP solution for decoupling rule connections follows a certain structure that can be abstracted in patterns. We distill these patterns and their variations and present them in the coming chapter. These patterns build on top of the common AOP characteristics and therefore are applicable to all approaches that adhere to those characteristics.

We also observe that because of the general purpose nature of all the analyzed AOP approaches, the offered support is sometimes too low-level for the kinds of problems we need to address when modularizing rule connections. This motivates the need for having higher-level abstractions, which is a further topic of this thesis and is presented in chapter 5.

# Chapter 4

# Aspect Patterns for Business Rule Connection

In our previous chapter we have demonstrated the suitability of AOP for decoupling business rules connections and illustrated this with concrete examples in AspectJ and JAsCo, two representative AOP approaches that succeed in accomplishing the requirements identified in section 2.8. In this chapter we are concerned with abstracting the commonalities of these solutions in aspect patterns. We first motivate the need for aspect patterns (section 4.1). We then observe that an AOP-based solution for the modularization of a rule connection typically involves a set of recurrent connection issues. Moreover, these issues vary in specific circumstances. We identify and discuss these issues that we call *rule connection elements* (section 4.2). Moreover, we make the distinction between elements that are mandatory — i.e. need to be part of every rule connection — or optional — i.e. might or might not be part be of a rule connection. Furthermore, we identify how and under which circumstances these elements vary and analyze which AOP features are suitable to implement each of these variations. We also observe that not all the variations of different elements can be always combined as for instance the choice of a certain variation for one element can restrict the set of possible variations for another element. As a result, this analysis lets us distill a set of *aspect patterns* that can serve as guidelines for the implementation of rule connection aspects (section 4.3). These patterns rely on AOP characteristics to which all approaches based on the pointcut-advice model adhere — in one way or another. In this chapter JAsCo is employed for illustration purposes.

## 4.1 Towards Aspect Patterns

Domain-specific reuse is gaining more and more attention in the software engineering research. Advancements in the area of domain-specific languages, software features and product lines are some examples of the efforts taken in this direction. Also in the AOP field, domain-specific reuse is starting to be a main research area. It is recognized by the community that reusable aspect patterns in general [HUS03] and domain-specific aspect patterns in particular can boost the adoption of AOP in industry[1]. Aspect patterns can encapsulate planned development and expert knowledge, reducing the amount of testing required for the deployment of aspects in industrial applications. Thus, aspect patterns enable a more

[1] Interview by Adrian Colyer on Domain Specific Aspects. Available at: http://www.infoq.com/interviews/Adrian-Colyer

reliable use of AOP, this way facilitating the incorporation of this (rather new) technology in production phases of the software development process. The aspect patterns presented in this chapter contribute to this line of research. They document best practices in applying AOP to a particular problem, the connection of business rules. Other efforts in the field of aspect patterns can be found in [Völ05], where the author proposes patterns for handling crosscutting concerns in the context of model-driven software development. This research is explained in more detail in section 9.4.

## 4.2 Identifying Rule Connection Elements

A first necessary element that any rule connection aspect has to include is the specification of *when* the rule application needs to be triggered. We refer to this element as *rule application time*. In the simplest scenario, the application time identifies a dynamic event that occurs during in the execution of the core application. In a more complex scenario, applying a rule might also require restricting the rule application to those dynamic events that occur within a specific context, such as the control flow of execution of another behavior. For instance, a discount rule — which is typically applied when the product price is retrieved — can be restricted only to those price retrievals that occur *while the customer is checking out*, or within the period of time *between the moment the customer logs in and the moment he/she adds a product to the shopping cart*, or *not while the customer is browsing the products*. We refer to this extra restriction on the rule applicability as a second connection element that we call *activation time*.

Another connection issue that needs to be tackled as part of a rule connection solution refers to *making the required information available to the rule* at rule application time. Different kinds of information might be required by the rules: core application objects that are reachable at the moment the rules are applied, core application objects that are not reachable at rule application time — in which case they have to be captured at other points in the execution where they are still reachable —, global information always available form any point of the system, and finally unanticipated information which was not foreseen in the existing core application.

The actual *triggering of the rule* is of course a mandatory element of any connection solution. Moreover, once the rule finished executing, the control must be returned to the core application. In order to achieve this, a mechanism that allows *resuming with the interrupted behavior* is needed. The process of resuming the execution of the core application can be done either implicitly or explicitly, depending on the kind of connection. Moreover, this step might imply the need for explicitly *retrieving rule results* from the rule after its application. When rule results are needed in the process of resuming the core application's execution, they have to be retrieved from the rule right after the rule's execution whereas they can be retrieved at a later point in time otherwise.

A typical AOP implementation of a rule connection consists of one or more related aspects. Every connection aspect is in charge of encapsulating one or more connection elements. Therefore, when looking at the code of the different aspects implementing a rule connection, different code fragments can be distilled corresponding to the realization of different connection elements. However, the code realizing each of the involved elements is not always well-localized in the aspect's implementation but tangled with code addressing other connection elements — in the same connection aspect or in others. This phenomenon is referred to as *tangled aspect code* [Fab05]. We observe that this problem is due to the existence of dependencies between the connection elements which impede the definition of a separate and well-modularized AOP implementation for each of them. By dependency we mean the fact that opting for a variation of a certain element — such as the rule application time — restricts the set of possible variations for other elements — such as the kind of information that can be passed to the rule at that point in time. These dependencies challenge the implementation of the rule connection aspects.

In the rest of this chapter we propose AOP implementations per element and per variation of the elements, taking into account the mentioned dependencies that exist between the elements. These dependences shape the kind of AOP solution that is needed for realizing the connection of a given rule. The connection elements and their variations are also discussed in [Cib02; CD06a; CDJ06b] (the first two papers consider rules implemented also as rule objects whereas the latter considers a rule-based language for the implementation of the rules). The identified dependences are also discussed in [CD06a].

To summarize, we have identified six elements which are inherent to the connection of the rules with the core application, namely:

A) determining the rule application time

B) determining the period of time in which the rule is considered active

C) making the required information available to the rule

D) triggering the rule

E) retrieving the rule results

F) proceeding with the interrupted core application's execution

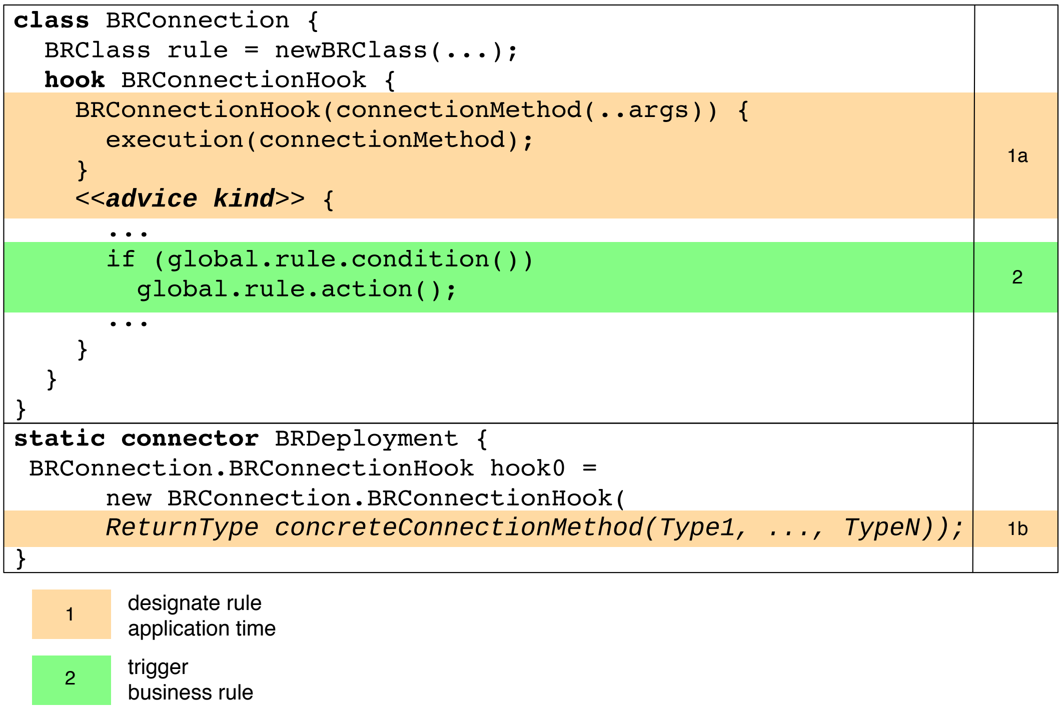
The coming sections are going to dig into the possible AOP solutions for the realization of these connection issues and their variations.

## 4.3 Aspect Patterns for Rule Connection

In order to provide a general overview of how a typical JAsCo solution looks, we first consider the simple set-up of a rule that needs to be triggered at an event, identifying for instance the moment a method *ReturnType concreteConnectionMethod(Type1, ..., TypeN)* is executed. When no other elements are involved, a simple JAsCo aspect bean is needed as shown in code fragment 4.1. This basic solution specifies an aspect bean containing the definition of a hook which defines an abstract pointcut specification (capturing the execution of any method) and an advice on that pointcut. The kind of advice will be made concrete depending on the specific variation of elements *A* and *B* (this is analyzed in detail in the coming sections 4.3.1 and 4.3.3). The advice first checks the rule's condition by invoking the `condition()` method on it, and in the case the condition is satisfied, it triggers the rule's action by invoking its `action()` method[2]. A connector is needed to deploy the abstract

[2] Note that triggering the rule's condition in the *isApplicable()* method of a JAsCo aspect bean is not equivalent to the solution presented here. This is because the moment the *isApplicable()* method is triggered differs from the moment the advices of that aspect are executed. In between these two execution points, the data upon which the condition is checked can change — as a result of the execution of other aspects — resulting in invalid rule applications.

pointcut with the signature of the concrete method: *returnType concreteMethod(Type1, ..., TypeN)*.



```
class BRConnection {
  BRClass rule = newBRClass(...);
  hook BRConnectionHook {
    BRConnectionHook(connectionMethod(..args)) {
      execution(connectionMethod);
    }
    <<advice kind>> {
      ...
      if (global.rule.condition())
        global.rule.action();
      ...
    }
  }
}
static connector BRDeployment {
 BRConnection.BRConnectionHook hook0 =
      new BRConnection.BRConnectionHook(
      ReturnType concreteConnectionMethod(Type1, ..., TypeN));
}
```

*Code Fragment 4.1:* Overview: JAsCo aspect bean for rule connection

### 4.3.1 Dynamic rule application time

A rule can conceptually be applied before, after or instead of the behavior captured by a dynamic event designating its rule application time:

a) *before* an event, meaning that the rule is applied just before the execution of the method captured by the event. The interrupted behavior is resumed after the rule is applied. For example, a rule can be connected before the method `Shop.checkoutShoppingBasket(ShoppingBasket)` is executed.

b) *after* an event, meaning that the rule is applied just after the execution of the method captured by the event, for instance after the execution of the `Customer.logIn()` method. The interrupted behavior is resumed after the rule is applied.

c) *instead of* an event, meaning that the rule's application completely replaces the behavior captured by the event. For instance, a payment rule encapsulating a new payment policy can be connected in replacement of the standard way of processing payment, implemented in the method `Shop.proceedPayment()`.

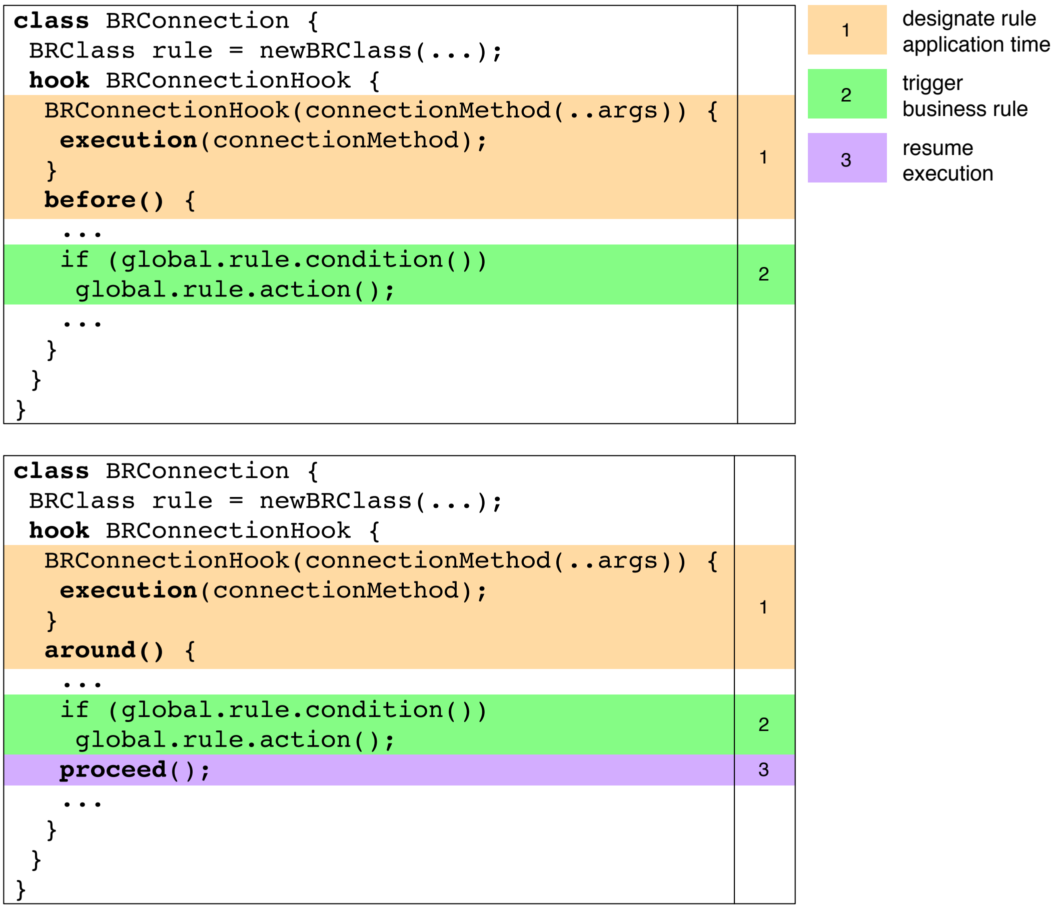
**Solution**

In JAsCo, independently of the connection case — before, after or instead of — an abstract pointcut needs to be defined capturing the execution of a generic method (as shown in code fragment 4.1). In order to actually distinguish between the cases, a different kind of advice on the abstract pointcut has to be defined. A different solution exists per case. Moreover, for some cases, different solutions are possible which differ in the kind of advice and the way the advice is actually implemented. The choice between these possible solutions depends on the concrete variation of connection element $C$ and also on how the rule manipulates the passed information. In what follows, we present solutions for the possible cases of the rule application time element, and we leave the discussion on which concrete solution to select for the coming section 4.3.3.

a) ***Before* rule application time**

Two solutions are possible:

1) a *before advice* is defined which triggers the rule (shown in upper part of code fragment 4.2).
2) an *around advice* is defined which first triggers the rule and then resumes the core application's execution by invoking `proceed()` (shown in lower part of code fragment 4.2).



```
class BRConnection {
 BRClass rule = newBRClass(...);
 hook BRConnectionHook {
  BRConnectionHook(connectionMethod(..args)) {
   execution(connectionMethod);
  }
  before() {
   ...
   if (global.rule.condition())
    global.rule.action();
   ...
  }
 }
}
```

```
class BRConnection {
 BRClass rule = newBRClass(...);
 hook BRConnectionHook {
  BRConnectionHook(connectionMethod(..args)) {
   execution(connectionMethod);
  }
  around() {
   ...
   if (global.rule.condition())
    global.rule.action();
   proceed();
   ...
  }
 }
}
```

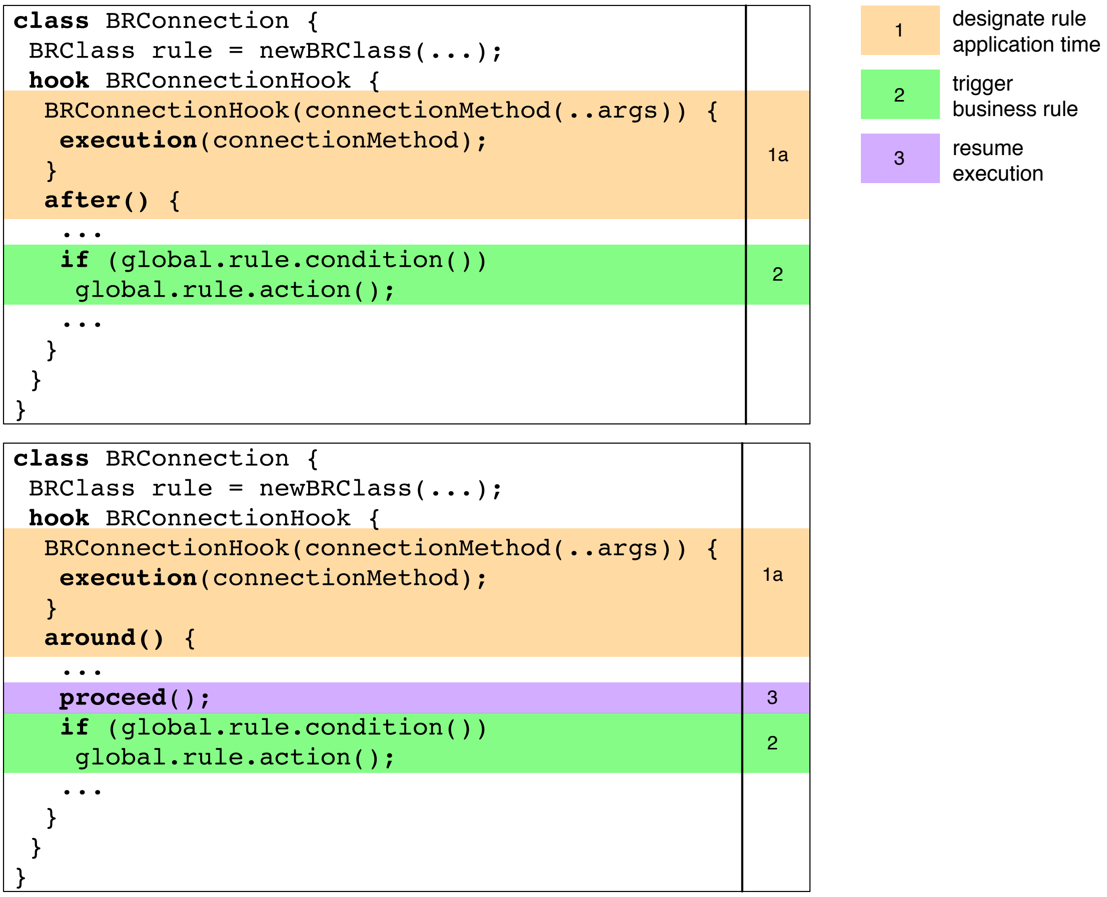
*Code Fragment 4.2:* Two possible realizations of a 'before' connection

b) ***After* rule application time**

Two solutions are possible:

1) an *after advice* is defined which triggers the rule (shown in upper part of code fragment 4.3).

2) an *around advice* is defined which first resumes the interrupted behavior and then triggers the rule (shown in lower part of code fragment 4.3). Note that two more specific kinds of around advice could be used in this case: *around returning* and *after returning*. An *around returning* advice is executed after the joinpoint that triggers the aspect and allows manipulating and changing the original return value. An *after returning* advice behaves in a similar way as the around returning advice with the exception that the original return can only be used and not modified.



```
class BRConnection {
 BRClass rule = newBRClass(...);
 hook BRConnectionHook {
  BRConnectionHook(connectionMethod(..args)) {
   execution(connectionMethod);
  }
  after() {
   ...
   if (global.rule.condition())
    global.rule.action();
   ...
  }
 }
}
```

```
class BRConnection {
 BRClass rule = newBRClass(...);
 hook BRConnectionHook {
  BRConnectionHook(connectionMethod(..args)) {
   execution(connectionMethod);
  }
  around() {
   ...
   proceed();
   if (global.rule.condition())
    global.rule.action();
   ...
  }
 }
}
```

1 designate rule application time

2 trigger business rule

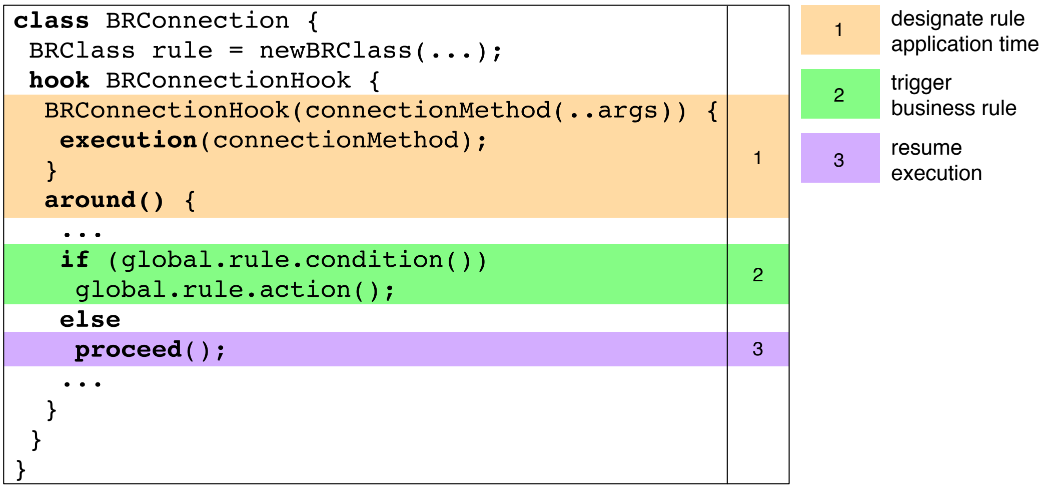
3 resume execution

*Code Fragment 4.3:* Two possible realizations of an ‘after’ connection

A rule applied before or after is typically additive in the sense that the rule’s behavior is executed in addition to the original interrupted behavior. However note that such a rule can also modify the way the interrupted behavior is resumed. This can be achieved only in the case the rules are triggered from within an around advice, and by means of changing the parameters of the `proceed()` invocation. This is explained in more detail in section 4.3.3.1.

c) ***Instead of* rule application time**

Only one solution is possible: an around advice is defined which triggers the application of the rule (shown in code fragment 4.4). In the case the rule applies (i.e. its condition is satisfied), the `proceed()` must not be invoked, this way causing the interrupted behavior not to be resumed (part 2). If the rule is not applicable, then we just simply need to resume the interrupted application by invoking `proceed()` (part 3).



```
class BRConnection {
 BRClass rule = newBRClass(...);
 hook BRConnectionHook {
  BRConnectionHook(connectionMethod(..args)) {      1
   execution(connectionMethod);
  }
  around() {
   ...
   if (global.rule.condition())                     2
    global.rule.action();
   else
    proceed();                                      3
   ...
  }
 }
}
```

1 designate rule application time

2 trigger business rule

3 resume execution

*Code Fragment 4.4:* Realization of an 'instead of' connection

### 4.3.2 Rule Activation Time

The activation time is defined in terms of one or more dynamic events. Four base cases are identified (other cases can be obtained by combining these base cases):

a) a rule is considered active while a certain event *activationEvent* executes, meaning that the rule is only considered during the period of time while the *activationEvent* is being executed

b) a rule is considered active always except during the execution of the *activationEvent*

c) a rule is considered active while a certain event *activationEvent1* executes and while a second event *activationEvent2* is not executed.

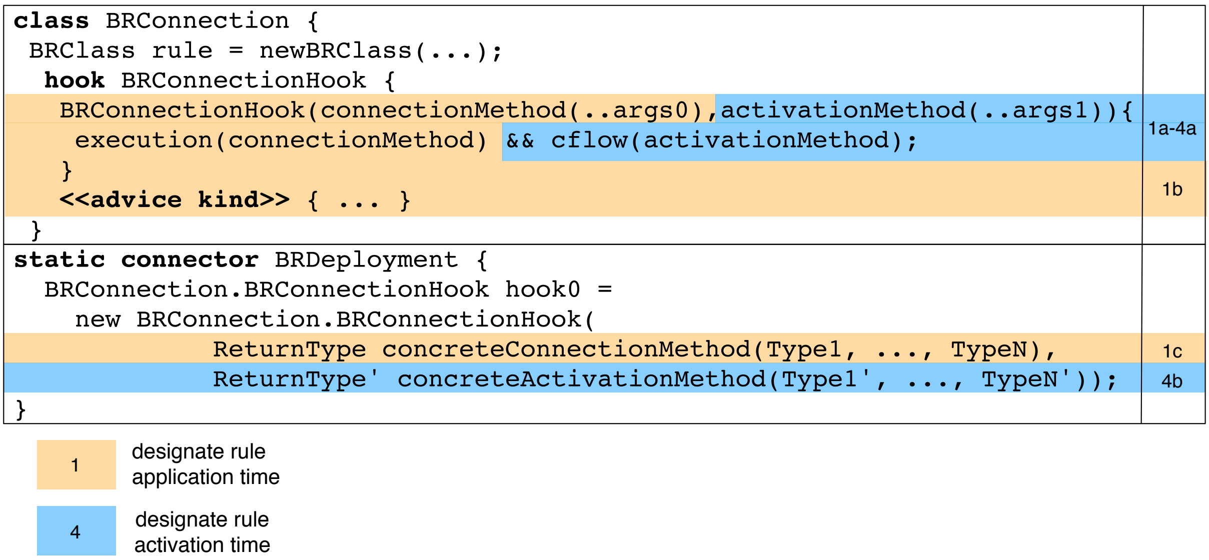
d) a rule is considered active in the period of time initiated by the execution of a first event *activationEvent1* and terminated by the execution of a second later event *activationEvent2*

The activation time is an optional element in a rule connection. When not explicitly specified it is assumed that the rule is always active (its application is not restricted to any context).

### Solution

A different AOP solution is proposed per case of activation time, as follows (consider that *activationEvent* captures the execution of a concrete method *concreteActivationMethod*):

a) **activation while *activationEvent***
An extra abstract method parameter `activationMethod(..args1)` must be added to the constructor of `BRConnectionHook`, representing the activation method. A `cflow(activationMethod)` pointcut must be added to the pointcut definition of `BRConnectionHook` shown in code fragment 4.1. These additions are shown in code fragment 4.5. The abstract *activationMethod* is deployed on the concrete method *concreteActivationMethod* (as done in the modified connector `BRDeployment` shown in the lower part of code fragment 4.5). This deployment ensures that the core application is only interrupted when the concrete methods bound to `connectionMethod` are executed in the control flow of the concrete methods bound to `activationMethod`.



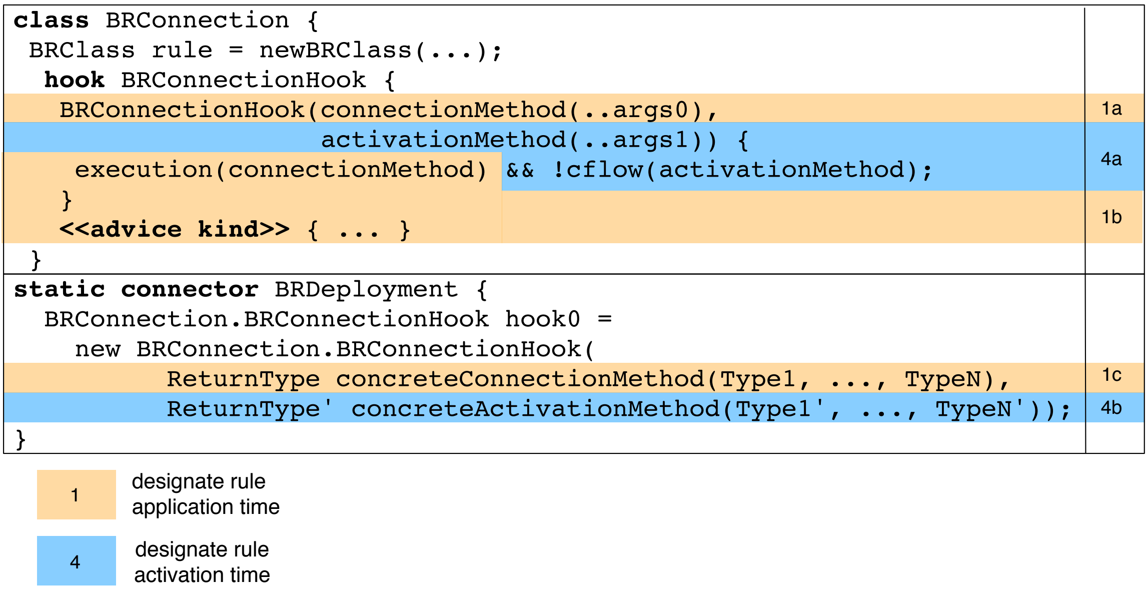
```
class BRConnection {
 BRClass rule = newBRClass(...);
  hook BRConnectionHook {
   BRConnectionHook(connectionMethod(..args0),activationMethod(..args1)){
    execution(connectionMethod) && cflow(activationMethod);
   }
   <<advice kind>> { ... }
 }
static connector BRDeployment {
  BRConnection.BRConnectionHook hook0 =
    new BRConnection.BRConnectionHook(
              ReturnType concreteConnectionMethod(Type1, ..., TypeN),
              ReturnType' concreteActivationMethod(Type1', ..., TypeN'));
}
```

1a-4a
1b
1c
4b
1
designate rule application time
4
designate rule activation time

*Code Fragment 4.5:* Rule activation considered 'while' execution of event

b) **activation not while *activationEvent***
Analogously to the previous case, an extra abstract method parameter `activationMethod(..args1)` must be added as an abstract method parameter to the constructor of `BRConnectionHook`, representing the activation method. A `!cflow(activationMethod)` pointcut is added to the pointcut definition of `BRConnectionHook`. These additions are shown in code fragment 4.6. The deployment of this hook on the concrete methods is identical to the previous case a) (lower part of code fragment 4.5). As a result, the core application is only interrupted when the concrete methods bound to `connectionMethod` are executed except those in the control flow of the concrete methods bound to `activationMethod`.

c) **activation while *activationEvent1* and not while *activationEvent2***
A solution for this case combines the two solutions presented for the cases *a* and



```
class BRConnection {
 BRClass rule = newBRClass(...);
  hook BRConnectionHook {
   BRConnectionHook(connectionMethod(..args0),
                    activationMethod(..args1)) {
    execution(connectionMethod) && !cflow(activationMethod);
   }
   <<advice kind>> { ... }
 }
static connector BRDeployment {
  BRConnection.BRConnectionHook hook0 =
    new BRConnection.BRConnectionHook(
          ReturnType concreteConnectionMethod(Type1, ..., TypeN),
          ReturnType' concreteActivationMethod(Type1', ..., TypeN'));
}
```

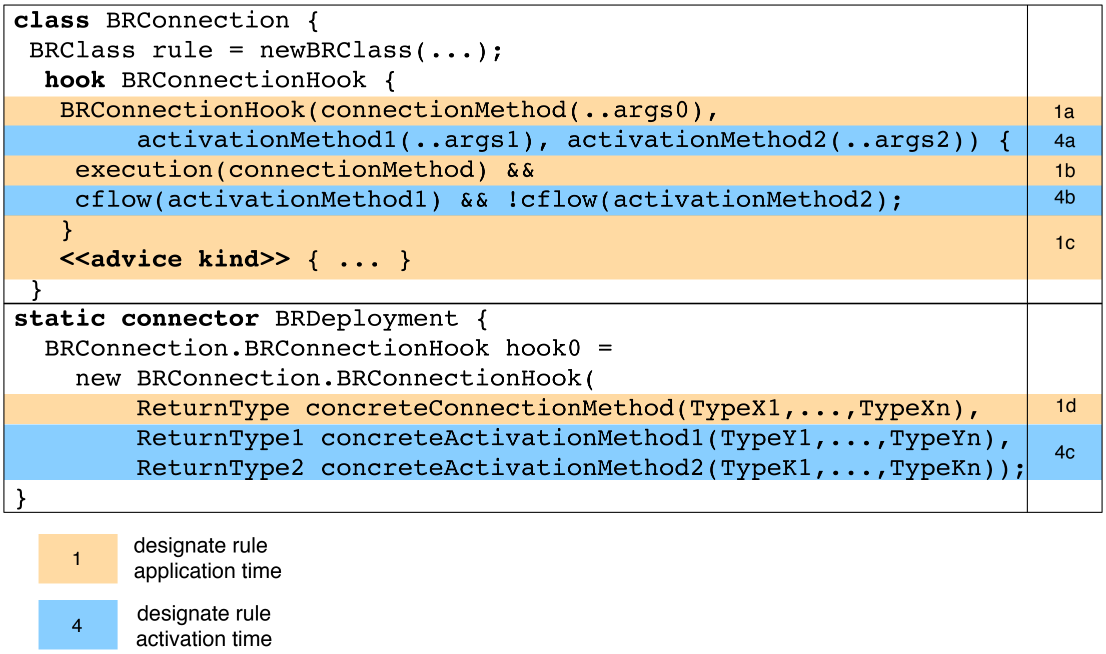
1a
4a
1b
1c
4b
1
designate rule application time
4
designate rule activation time

*Code Fragment 4.6:* Rule activation considered 'not while' execution of event

*b*: two abstract method parameters `activationMethod1(..args1)` and `activationMethod2(..args2)` must be added as abstract method parameters to the constructor of `BRConnectionHook`. Furthermore, a `cflow(activationMethod1) && !cflow(activationMethod2)` pointcut is added to the pointcut definition of `BRConnectionHook`. These additions are shown in code fragment 4.7. As a result, the core application is only interrupted when the concrete methods bound to `connectionMethod` are executed within the control flow of `activationMethod1` and not in the control flow of `activationMethod2`.

d) **activation between *activationEvent1* and *activationEvent2***
The execution of the core application must be interrupted at two specific moments: *activationEvent1* denoting the start of the activation period and *activationEvent2* denoting the end of it. A stateful hook [VSCF05] must be added to the aspect bean which defines two transitions (`p1` and `p2`) capturing the execution of two abstract method parameters `activationMethod` and `deactivationMethod` (which will then be deployed in the connector on the concrete methods whose executions are denoted by *activationEvent1* and *activationEvent2* respectively) (shown in code fragment 4.8). A `before` and an `after` advice are defined on `p1` and `p2` respectively in charge of setting and unsetting a flag (which is a local variable defined in the aspect bean) indicating whether the rule is active or not. Additionally, the `isApplicable` method of the connection hook must return the value of this flag. The `isApplicable` method is a special JAsCo method that is executed when a joinpoint is encountered and determines whether the aspect should be applied. As a consequence, in the case the value is true, the connection advice will be executed and therefore the rule will be triggered.

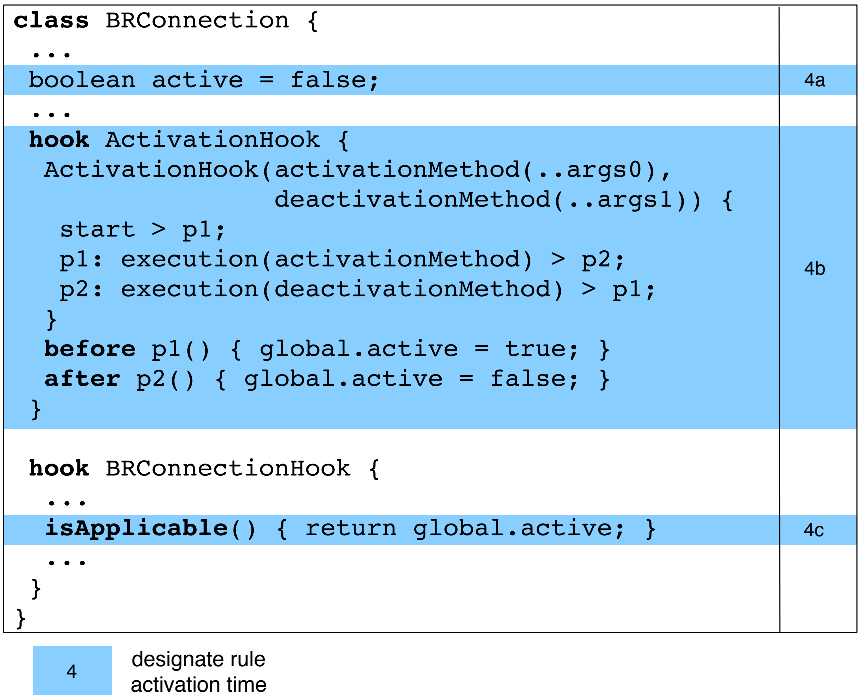
The presented solution assumes the core application to be single-threaded. When a multi-threaded application is considered, the connector must precede the instantiation



```
class BRConnection {
 BRClass rule = newBRClass(...);
  hook BRConnectionHook {
   BRConnectionHook(connectionMethod(..args0),
        activationMethod1(..args1), activationMethod2(..args2)) {
    execution(connectionMethod) &&
    cflow(activationMethod1) && !cflow(activationMethod2);
   }
   <<advice kind>> { ... }
 }
static connector BRDeployment {
  BRConnection.BRConnectionHook hook0 =
    new BRConnection.BRConnectionHook(
        ReturnType concreteConnectionMethod(TypeX1,...,TypeXn),
        ReturnType1 concreteActivationMethod1(TypeY1,...,TypeYn),
        ReturnType2 concreteActivationMethod2(TypeK1,...,TypeKn));
}
```

1a
4a
1b
4b
1c
1d
4c
1
designate rule application time
4
designate rule activation time

*Code Fragment 4.7:* Rule activation considered ‘not while’ execution of event



```
class BRConnection {
 ...
 boolean active = false;
 ...
 hook ActivationHook {
  ActivationHook(activationMethod(..args0),
                 deactivationMethod(..args1)) {
   start > p1;
   p1: execution(activationMethod) > p2;
   p2: execution(deactivationMethod) > p1;
  }
  before p1() { global.active = true; }
  after p2() { global.active = false; }
 }

 hook BRConnectionHook {
  ...
  isApplicable() { return global.active; }
  ...
 }
}
```

*Code Fragment 4.8:* Rule activation considered ‘between’ the execution of two events of the `BRConnectionHook` with the keyword `perthread`. Note that the use of stateful aspects might be overkill for implementing an activation period denoted by two events. It is also possible to use two stateless hooks that keep track each of them of the start and end of the activation period by setting to true and false the variable `active` of the aspect bean. Although this solution is valid, manually keeping track of the aspect state might result in complex aspect code [VSCF05]. Thus, a stateful solution as the one presented here is desired in those cases.

### 4.3.3 Passing/Retrieving Information to/from the Rule

For the rules to be able to apply, required information needs to be made available to the rules. Some rules only need global information always available in the system, such as the current system date. Global system information is directly reachable by the rules. When non-global information is required, several issues need to be taken into account: first we need to identify *which kind of information is required by the rules.* Different cases are identified:

- the information is reachable at the rule application event or not:
    - *contextual*: it represents core application's objects reachable from the context in which rules are triggered
    - *non-contextual*: it represents core application objects that are not reachable in the context of the event at which rules are executed
- the data structures are present in the core application or not:
    - *anticipated*: it exists in the core application
    - *unanticipated*: it is not present in the current implementation

Note these two cases are orthogonal to each other, and thus different combinations are possible: contextual-anticipated, contextual-unanticipated, captured-anticipated and captured-unanticipated information.

A second important issue is deciding *when the required information is passed to the rule.* For this issue, it is not important whether the information is anticipated or unanticipated but we do need to distinguish between the cases of contextual and captured information: the contextual information needs to be passed right before the rule application is triggered, whereas the captured information can be passed both at rule application time or it can be made available to the rule at a different point in time.

Also, contextual and non-contextual information that gets modified by the rule has to be retrieved from the rule right after its application, because it has to be used when proceeding with the interrupted behavior.

In the following sections we analyze these issues in detail and present AOP solutions for each variation. For the first two cases we assume we are dealing with anticipated information.

#### 4.3.3.1 Contextual Information

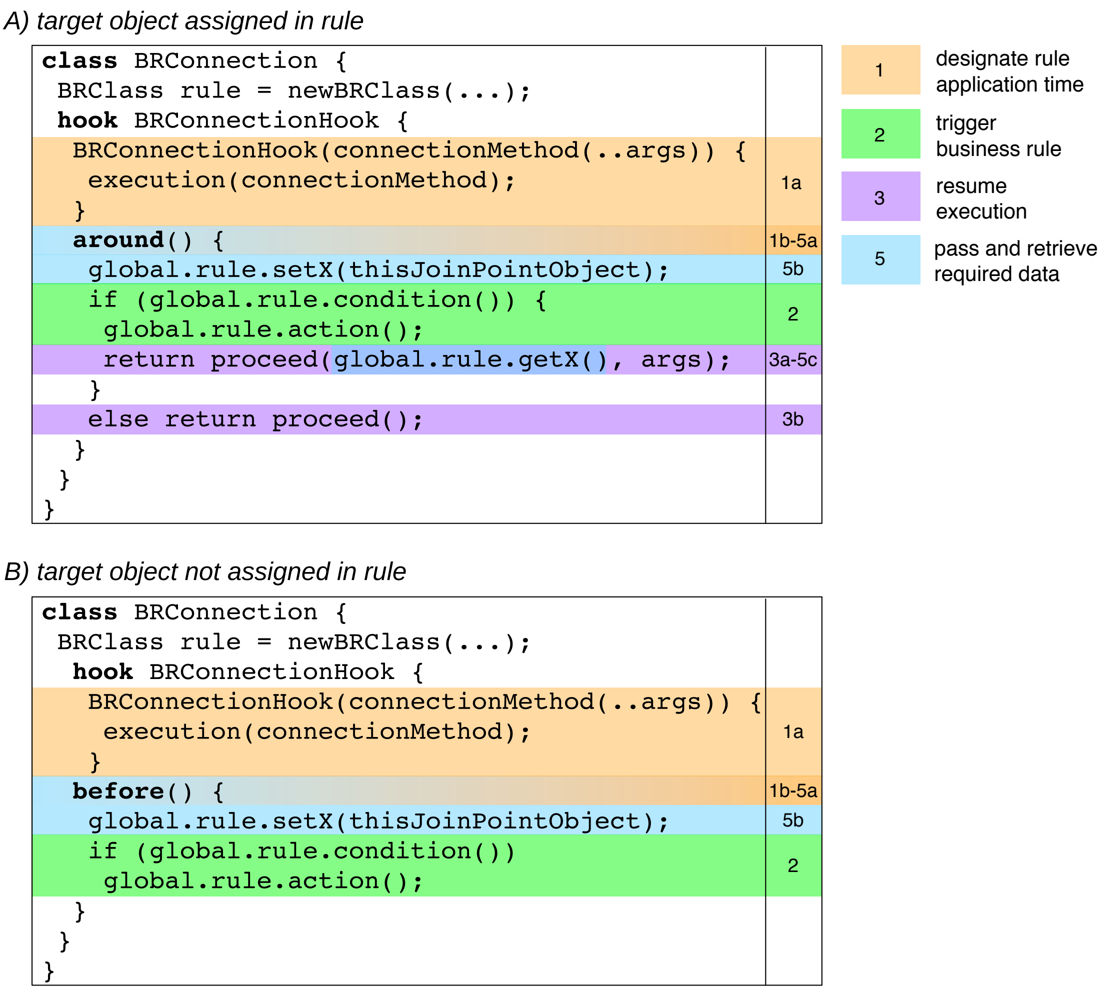
In AOP, a pointcut can expose data that is present in the context of a join point at which the application is interrupted. In JAsCo, contextual data is directly exposed by pointcuts, without the need for having to explicitly capture it first (this is not the case in other AOP approaches, such as AspectJ). The possible objects that can be exposed by a pointcut correspond to the target object and the parameters of the interrupted behavior. These objects are directly accessible from within any advice defined on that pointcut. In JAsCo, the target object is directly accessible using the *thisJoinPointObject* construct and the arguments of the abstract method parameter are reachable by referring to the corresponding variable (e.g. `args` in code fragment 4.1). The return value can only be obtained in an around advice by explicitly invoking `proceed()`. We can use this *pointcut context exposure* mechanism of AOP in order to access data available in the context of the pointcut designating the rule application time (shown in code fragment 4.1). This exposed data can then be passed to the rule as parameters of its `apply()` method from within the advice that triggers the rule application.

However, the actual solution for passing the required objects to the rule depends on which kind of information is needed and how it is manipulated by the rule[3]. Depending on the case, a different kind of advice is needed. We analyze all the possibilities per connection case:

- **before *connection***: the available contextual information is the target object and the parameters. Two cases are possible depending on whether the rule assigns new values to the passed objects or not (depicted in code fragments 4.9 and 4.10):

  1) *the rule assigns the passed contextual information*: this means that the rule has determined new values for the original target object or parameters. These new values need to be explicitly retrieved after the application of the rule and need to be taken into account when resuming the invocation of the interrupted behavior. Therefore, an *around advice* is needed, since it allows interrupting the application at a certain point, adding some extra business logic and proceeding with the original execution, eventually considering a different target object and parameters. In this around advice we first pass the information to the rule, trigger the rule, retrieve results and finally proceed. This case is depicted in the upper part of code fragments 4.9 and 4.10 (the former shows the case in which the target object is passed to the rule and assigned a new value, whereas the latter shows the same situation for the jth parameter of the interrupted method).

[3] The different cases that are listed here are mainly driven by the way parameter passing is supported in Java, since this is the object-oriented language our approach is based upon. In Java parameters are passed by value. Therefore, anything passed to the rule remains unchanged in the caller's scope when the rule returns. However, we are interested in changing the caller's context with results produced by the rule. Therefore, whenever the rule assigns a new value to the passed parameters, in order to make these new values available in the caller's context, we need to explicitly retrieve them from the rule. This can be done by invoking methods on the rule. If the parameters are not primitive values but actual objects, the rule can also assign values to some attributes of those objects — that are obtained through attribute navigation — in which case there is no need anymore for retrieving results, as the modified objects are the same objects available in the caller's scope. When the base object-oriented language supports call-by-reference, the need for explicitly retrieving rule results from the rules becomes obsolete.



*A) target object assigned in rule*

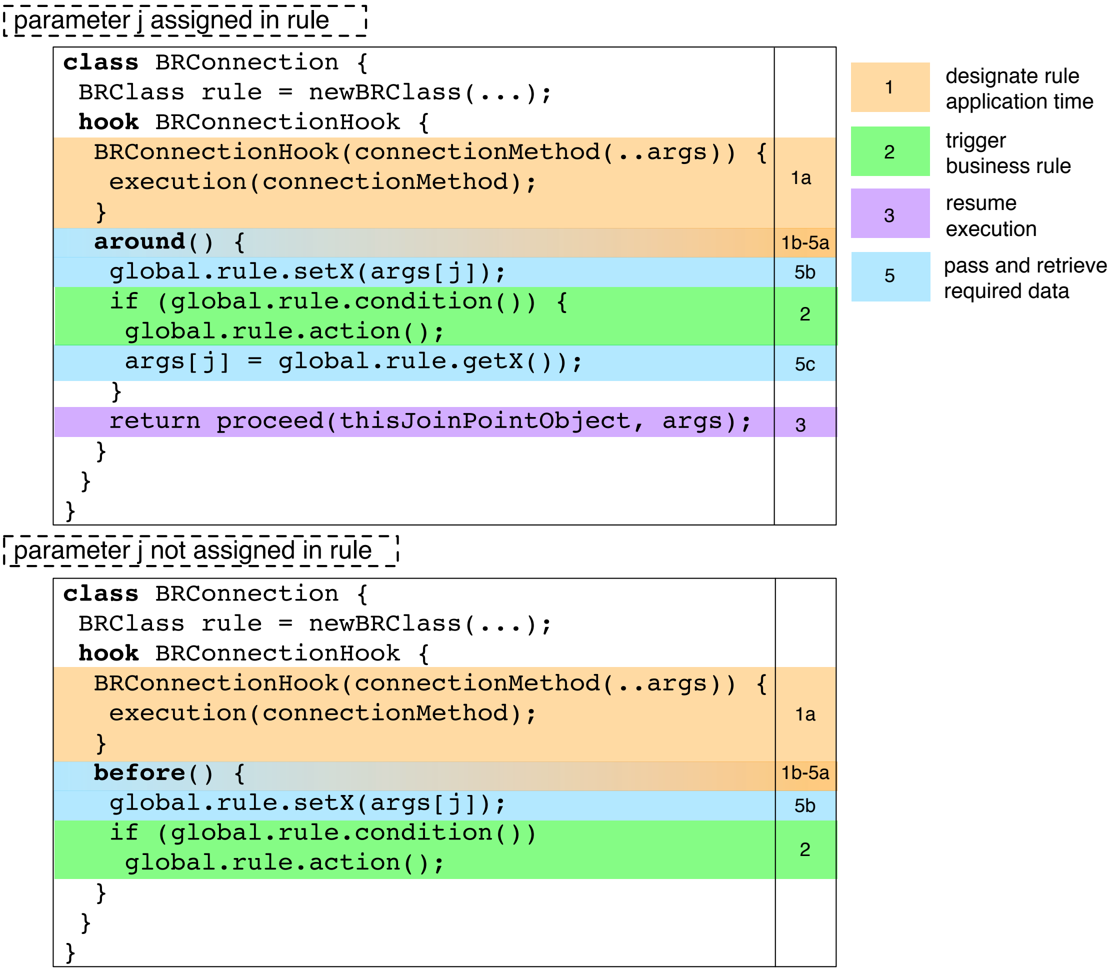
```
class BRConnection {
 BRClass rule = newBRClass(...);
 hook BRConnectionHook {
  BRConnectionHook(connectionMethod(..args)) {
   execution(connectionMethod);
  }
  around() {
   global.rule.setX(thisJoinPointObject);
   if (global.rule.condition()) {
    global.rule.action();
    return proceed(global.rule.getX(), args);
   }
   else return proceed();
  }
 }
}
```

*B) target object not assigned in rule*

```
class BRConnection {
 BRClass rule = newBRClass(...);
  hook BRConnectionHook {
   BRConnectionHook(connectionMethod(..args)) {
    execution(connectionMethod);
   }
  before() {
   global.rule.setX(thisJoinPointObject);
   if (global.rule.condition())
    global.rule.action();
  }
 }
}
```

1a
1b-5a
5b
2
1 designate rule application time
2 trigger business rule
3 resume execution
5 pass and retrieve required data

*Code Fragment 4.9:* Before connection: exposing and passing target object to rule


parameter j assigned in rule
1a
1b-5a
5b
2
5c
3
1
designate rule application time
2
trigger business rule
3
resume execution
5
pass and retrieve required data
parameter j not assigned in rule
1a
1b-5a
5b
2

```
class BRConnection {
 BRClass rule = newBRClass(...);
 hook BRConnectionHook {
  BRConnectionHook(connectionMethod(..args)) {
   execution(connectionMethod);
  }
  around() {
   global.rule.setX(args[j]);
   if (global.rule.condition()) {
    global.rule.action();
    args[j] = global.rule.getX());
   }
   return proceed(thisJoinPointObject, args);
  }
 }
}
```

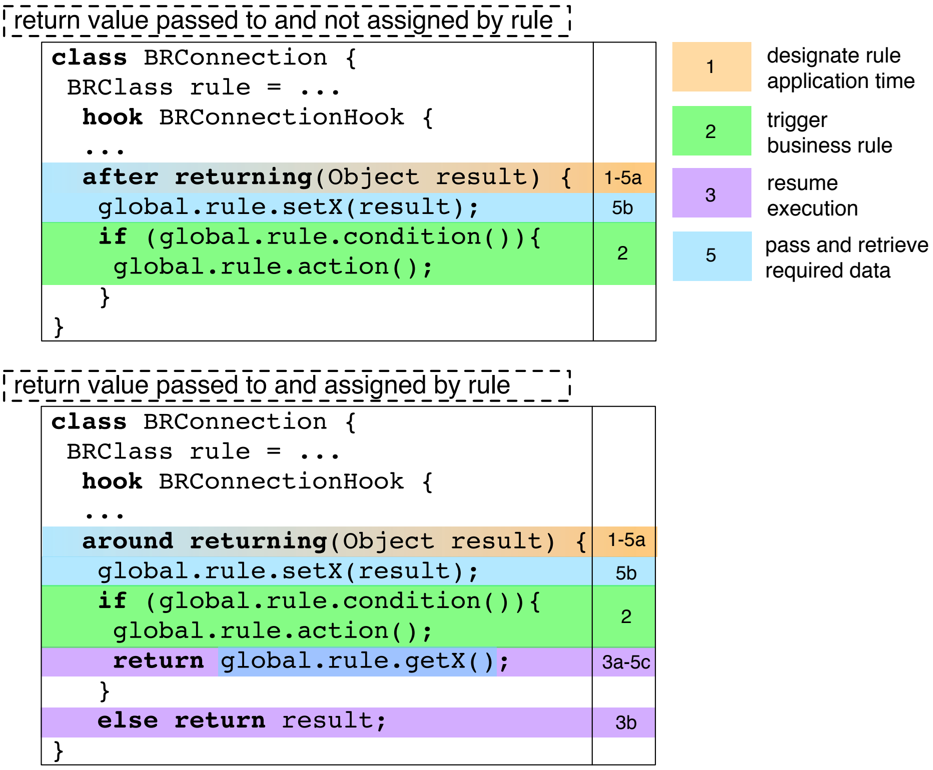
```
class BRConnection {
 BRClass rule = newBRClass(...);
 hook BRConnectionHook {
  BRConnectionHook(connectionMethod(..args)) {
   execution(connectionMethod);
  }
  before() {
   global.rule.setX(args[j]);
   if (global.rule.condition())
    global.rule.action();
  }
 }
}
```

*Code Fragment 4.10:* Before connection: exposing and passing parameter to rule

2) *the rule does not assign the passed contextual information*: in this case, a *before advice* suffices to trigger the rule's action, as the original event execution does not need to be modified. This case is depicted in the lower part of code fragments 4.9 and 4.10 (the former assumes the target object is passed to the rule but not assigned, whereas the latter assumes the same for the *jth* parameter of the interrupted method).

- **after *connection***: the available contextual information is the target object and the parameters and eventually the original return value. Two cases are possible depending on whether the rule assigns new values to the passed objects or not:

1) *the result of proceeding with the interrupted behavior is passed to the rule*: Two cases are possible: i) in the first case the original value is passed to the rule where it is not assigned a new value. Then, an *after returning advice* is needed which passes the original return value to the rule, triggers the rule and returns the original result as a result of the advice's execution (shown in upper part of code fragment 4.11); ii) in the second case, the original value is passed to the rule where it is assigned a new value. Then, an *around returning advice* is needed which passes the original return value to the rule, triggers the rule and queries the rule in order to retrieve the new assigned value. This value is then returned as the result of the advice's execution (shown in lower part of code fragment 4.11).



return value passed to and not assigned by rule

```
class BRConnection {
 BRClass rule = ...
  hook BRConnectionHook {
  ...
  after returning(Object result) {     1-5a
   global.rule.setX(result);           5b
   if (global.rule.condition()){       2
    global.rule.action();
   }
}
```

return value passed to and assigned by rule

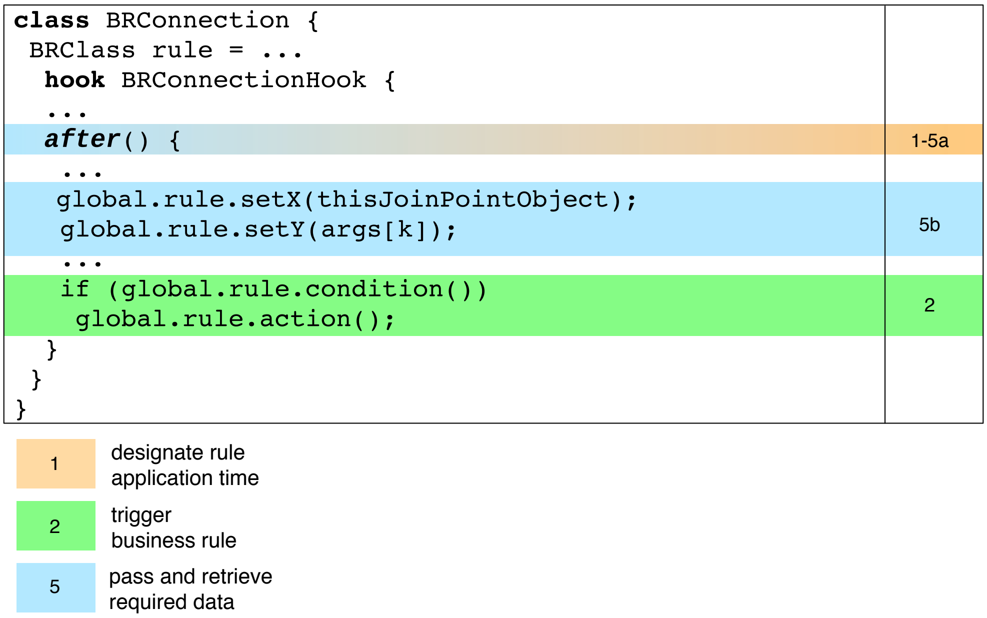
```
class BRConnection {
 BRClass rule = ...
  hook BRConnectionHook {
  ...
  around returning(Object result) {    1-5a
   global.rule.setX(result);           5b
   if (global.rule.condition()){       2
    global.rule.action();
    return global.rule.getX();         3a-5c
   }
   else return result;                 3b
}
```

1 designate rule application time
2 trigger business rule
3 resume execution
5 pass and retrieve required data

*Code Fragment 4.11:* After connection: exposing and passing return value object to rule

2) *the result of proceeding with the interrupted behavior is not passed to the rule*: an *after advice* suffices to trigger the rule's action, after the execution of the

connection event. This solution is shown in code fragment 4.12. An after advice is suitable independently of whether the parameters or the target object are passed to the rule (in which case they are passed before the actual rule triggering, as shown in part 5b of Figure 4.12). As we are in the case of an after connection, the execution of the interrupted behavior is completed before the rule is actually triggered. Therefore, whether or not the rule assigns new values to the objects it received — parameters or target object — is irrelevant, as the new values can no longer be taken into account in the execution of the interrupted behavior.



```
class BRConnection {
 BRClass rule = ...
  hook BRConnectionHook {
  ...
  after() {
   ...
   global.rule.setX(thisJoinPointObject);
   global.rule.setY(args[k]);
   ...
   if (global.rule.condition())
    global.rule.action();
  }
 }
}
```

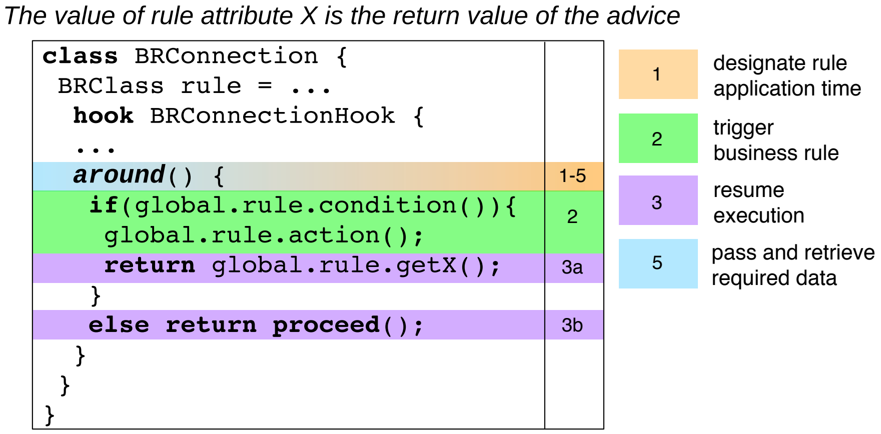
1-5a
5b
2
1
designate rule application time
2
trigger business rule
5
pass and retrieve required data

*Code Fragment 4.12:* After connection: not passing return value to rule

- **instead of *connection***: this means that when the rule's condition is satisfied, the interrupted behavior must be dropped and replaced by the behavior specified by the rule's action. To this purpose, an *around* advice is needed which triggers the rule and does not proceed with the original execution. This is shown in code fragment 4.13. When the rule's condition is not satisfied, a simple `proceed()` must be executed to resume the interrupted behavior.

#### 4.3.3.2 Non-Contextual Information

In order to capture the required unavailable object, we first need to identify the moment when it is still reachable. We should then be able to interrupt the execution of the core application at this moment, which can be any point in the execution of the core application after the creation of the pertinent object and before the point in time at which the rule is triggered. At that point, the object needs to be captured and made available to the event that triggers the rule. To this purpose a second JAsCo hook — that we call `BRCaptureHook` — is added to the same `BRConnection` aspect bean, as shown in code fragment 4.14. `BRCaptureHook` defines a pointcut designating the points in time at which the required objects are reachable and an advice that stores the exposed required objects in global variables of the aspect bean. Once the required objects are captured and stored in



*The value of rule attribute X is the return value of the advice*

```
class BRConnection {
 BRClass rule = ...
  hook BRConnectionHook {
  ...
  around() {
   if(global.rule.condition()){
    global.rule.action();
    return global.rule.getX();
   }
   else return proceed();
  }
 }
}
```

1-5
2
3a
3b
1 designate rule application time
2 trigger business rule
3 resume execution
5 pass and retrieve required data

*Code Fragment 4.13:* Instead of connection: the rule's attribute X represents the new return value to be consider when proceeding with core application's execution

the aspect bean, they can be accessed and used by other hooks defined in the same aspect bean, as for instance the connection hook from where the rule is actually triggered. It is in the advice of this last hook that the captured objects are passed to the rule. This is done in the same way as for the contextual information (thus not shown here). Note that the presented solution assumes the core application to be single-threaded. When a multi-threaded application is considered, the connector must precede the instantiation of the `BRConnectionHook` and the `BRCaptureHook` with the keyword `perthread`.

#### 4.3.3.3 Unanticipated Information

When the information does not exist in the current implementation, a mechanism is needed so that we are able to extend the core application with the new information. JAsCo's *virtual mixins* allow extending the core application objects with new methods. Code fragment 4.15 shows a possible solution: an interface defining the methods to be added to the class `AClass` is defined in part labeled with 7a; an implementation for this interface is provided by the hook `Introduce`, shown in part labeled with 7b; this hook can also define attributes that might be needed in order to implement the added methods. In order to actually introduce the two methods defined in the `Introduce` hook to all the instances of the class that is extended, the `Introduce` hook must be instantiated `perobject` of that class, as it shown in the connector named `IntroduceMixin` shown in part labeled with 7c. This instantiation makes sure that a different hook exists per instance of `AClass`. The added methods can be invoked from either the advice that triggers the rule application, i.e. from within `BRConnectionHook`, or from hooks that capture non-contextual information, e.g. `BRCaptureHook`, as shown in the lower part of code fragment 4.15.

Once the required information is added, it can be used the same way as anticipated information. Therefore, the cases contextual-unanticipated and captured-unanticipated are analogous to the cases contextual-anticipated and captured-anticipated respectively.

```
class BRConnection {
  ...
  Object obj;                                                  6a
  ...
  hook BRConnectionHook {
    BRConnectionHook(connectionMethod(..args)) {
      execution(connectionMethod);
    }
    <<advice>> {                                               1a
      ... //use obj to trigger rule
    }
  }
  hook BRCaptureHook {
    BRCaptureHook(captureMethod(..args)) {
      execution(captureMethod);
    }
    <<advice>> {                                               6b
      ... //store contextual object in variable obj
    }
  }
}

static connector BRDeployment {
  BRConnection.BRConnectionHook hook0 =
    new BRConnection.BRConnectionHook(                          1b
     <<signature concrete connection method>>);
  ...
  BRConnection.BRCaptureHook hook1 =
   new BRConnection.BRCaptureHook(                              6c
    <<signature concrete capturing method>>);
}
```

| | |
|---|---|
| 1 | designate rule application time |
| 6 | capture non-contextual data |

*Code Fragment 4.14:* Capturing non-contextual information

```
interface IExtendedObject extends jasco.runtime.mixin.IMixin {
  public Type1 method1(...);
  ...
  public TypeN methodN(...);
}
```

7a

```
class IntroduceUnanticipatedInfo {
  hook Introduce implements IExtendedObject {
    TypeX attributeX = valueX;
    Introduce(void method(..args)) {
        execution(method(args));
    }
    public Type1 method1(...) { //concrete implementation 1 }
    public TypeN methodN(...) { //concrete implementation N }
  }
}
```

7b

```
static connector IntroduceMixin {
  perobject IntroduceUnanticipatedInfo.Introduce introHook =
   new IntroduceUnanticipatedInfo.Introduce(* ConcreteClass.*(*));
}
```

7c

```
class BRConnection {
  hook BRConnectionHook {
    BRConnectionHook(connectionMethod(..args)) {
      execution(connectionMethod);
    }
    <<advice>> {
      //obj is contextual object of Type ConcreteClass
```

1a

```
      IExtendedObject extendedObj = (IExtendedObject) obj;
      extendedObj.method1(...);
      ...
      extendedObj.methodN(...);
```

1b-7a

```
    }
  }
  hook BRCaptureHook {
    BRCaptureHook(captureMethod(..args)) {
      execution(captureMethod);
    }
    <<advice>> {
      //obj is contextual object of Type ConcreteClass
```

6

```
      IExtendedObject extendedObj = (IExtendedObject) obj;
      extendedObj.method1(...);
      ...
      extendedObj.methodN(...);
```

7b

```
    }
  }
}
```

| | |
|---|---|
| 1 | designate rule application time |
| 6 | capture non-contextual data |
| 7 | add and access unanticipated data |

*Code Fragment 4.15:* Extending the core application with unanticipated information

## 4.4 Summary

In this chapter we have presented aspect patterns for the implementation of crosscutting rule connections. We have identified and presented several recurrent connection elements that can be combined in different ways and vary under certain circumstances. These connection elements are solution-independent. Moreover, we propose solutions to these connection elements that only rely on the AOP characteristics identified in the previous chapter (section 3.3.1) and therefore are generic. Although JAsCo was used to illustrate these solutions, the aspect patterns do not rely on the concrete features of this language. Aspect patterns respect the same structure independently of the concrete AOP approach adopted and thus, given another AOP approach that adheres to those AOP characteristics, implementing these aspect patterns could be done in a similar way with minimal effort.

We observe that, although aspects are a good solution to the problem of decoupling crosscutting rule connection code, they exclude the domain expert. Moreover, writing these aspects by hand is a complex task, as they need to consider and tackle all the recurrent issues identified in this chapter. The next chapter provides a solution to the expression of rule connections at the domain level and removing the need for having to implement these connection aspects by hand.

# Chapter 5

# A Domain Model for Domain Entities, High-Level Business Rules and High-Level Business Rule Connections

So far we have presented solutions based on AOP for the decoupling of the crosscutting rule connection code from core applications. We have also shown that AOP is a suitable technology for achieving our goal and presented reusable AOP-based patterns as guidelines for tackling several recurrent connection issues that we identified (chapter 4). In this chapter we are concerned with one of our ultimate goals, which is the consideration of the domain expert as an active participant in the process of understanding, defining business rules and integrating them with the existing application.

In this chapter we propose moving to a higher level of abstraction by building a *high-level domain model* which incorporates ideas from Model-Driven Engineering (MDE) to achieve the integration of high-level and executable business rules in existing applications. First of all, the ideas behind MDE are presented in section 5.1. Afterwards, the domain model is presented: we show how domain concepts can be explicitly captured (section 5.2) and how business rules and their connections to an existing core application can be defined in terms of those explicit domain concepts (sections 5.3 and 5.4 respectively). We pursue this domain model to be *high-level*, meaning that no details are exposed about the concrete implementation of the core application where the rules are applied. As a consequence, reusing domain knowledge among different applications on the same domain or among different versions of an evolving application becomes possible. Also, new domain vocabulary that appears due to domain evolution can be represented. The translation from high-level rules and connections to their implementation is achieved automatically and transparently for the domain experts (section 5.6). High-level rules are transformed into rule-objects whereas rule connections are transformed into aspects (following the patterns defined in chapter 4). An important requirement for the implementation of these transformations, is the existence of a mapping which defines how domain entities — involved in the high-level definitions — are realized at the level of the implementation. This mapping will be the subject of our next chapter 6 and thus in the rest of the current chapter we make abstraction of the way domain entities are defined and mapped, and we concentrate on their use as part of the definition of the business rules and their connections.

## 5.1 Model-Driven Engineering

*Model-driven engineering* is an approach to software engineering which aims to raise the level of abstraction, and to develop and evolve complex software systems by means of manipulating models. Therefore, models are its primary assets. The manipulation of models is achieved by means of model transformation, which is considered to be the heart and soul of model-driven engineering [SK03]. A model transformation can encode a refinement step, an evolution step, and even a code generation step and can take one or multiple source models and produce one or multiple target models.

The ultimate goal of MDE is to have a software development environment at our disposal with off-the-shelf models and mapping functions that *transform* one model into another. Mellor et al. [MCF03] state that *"Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing"*. This definition remarks the importance of both MDE components, models and transformations: models are meant to be transformed in order to obtain the expected result. Adopting this idea in our work, we envision having a *model* that can raise the level of abstraction by means of explicitly capturing domain concepts and expressing business rules and their connections in terms of those domain concepts. The *real thing* is then having 'executable' implementations for the business rules and connections that can be directly integrated in existing — and even running — applications.

### 5.1.1 Models

A model describes a certain view of the software system at a certain level of abstraction. For example in a bank application, different models can be defined to represent customer management, transaction management and account management. Models can be specified at different levels of abstraction and sometimes also in different languages. Mellor et al. [MCF03] define a model as:

> *A coherent set of formal elements describing something (e.g., a system, bank, phone or a train) built for some purpose that is amenable to a particular form of analysis, such as:*
>
> - *communication of ideas between people and machines,*
> - *completeness checking,*
> - *race condition analysis,*
> - *test case generation,*
> - *viability in terms of indicators such as cost and estimation,*
> - *standards,*
> - *transformation into an implementation.*

#### 5.1.1.1 Domain Modeling

When building a domain model, a first required step consists of getting acquainted with the domain of interest. In software engineering the term *domain* is typically associated with two broad interpretations: a *business domain* and an *application domain* [WPD92]. The first one refers to a domain as a subset of knowledge about some area in the real world confined to a particular business. For example domain knowledge about the insurance domain include the concepts of insurance policy, claim, and policy holder. The second interpretation of domain does not refer to the real-world domain concepts but to knowledge about software applications in a certain field, for instance in the field of middleware or distributed applications. Therefore, this interpretation of domain focuses on the solution space, that is the domain of computing technologies themselves, instead of the problem space.

Over the past five decades, software researchers and developers have been focusing on the solution space by means of creating suitable abstractions on top of the underlying computer environment that could help them concentrate on their design intent rather than the complex low-level details. In [Sch06], Schmidt observes that *even though current languages and platforms have considerably raised the level of abstraction, they still have a "computing-oriented" focus rather than a "domain-oriented" focus. This means that current languages and platforms succeed in providing abstractions of the solution space rather than abstractions of the problem space that express designs in terms of concepts in problem domains, such as telecom, aerospace, healthcare, insurance, and biology.* Therefore, *MDE technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively.*

#### 5.1.1.2 Gathering and Representing Domain Knowledge

The process responsible for creating a domain model is referred to as domain analysis. Obtaining a comprehensive body of domain knowledge is done in part by applying several knowledge acquisition techniques. Such techniques are mainly targeted at the elicitation, the analysis, and the organisation of knowledge coming from different sources. Some of the most common approaches to identify the core concepts are interviews, observations, and document analyses. This process has as objective getting used to the common jargon or domain vocabulary in which experts describe the concepts and rules that govern the domain of interest. It is outside the scope of this dissertation to give a detailed description of the different knowledge acquisition techniques. For more information on the process of engineering knowledge-intensive software systems the interested reader is referred to the Common KADS methodology [SAA$^+$00].

In order for the domain knowledge to be of use, it has to be encoded in a suitable knowledge representation mechanism. This is not a trivial issue as it has to do with the central problem of encoding human knowledge in all its various forms. Knowledge Representation (KR) has long been considered one of the principal elements of Artificial Intelligence, and a critical part of all problem solving [New82]. One of the most important developments in the application of KR has been the so-called frame-based KR languages or systems (proposed by Minsky in 1981 [Min81]). Frame-based systems are knowledge representation systems that use frames as their primary means to represent domain knowledge. A frame is a structure for representing a concept or situation. Attached to a frame are several kinds of information, for instance, definitional and descriptive information and how to use the frame. While frame-based KR languages vary from each other in some degrees, they share some common characteristics: (1) frames are organized in hierarchies; (2) frames are composed out of slots (attributes) for which fillers (scalar values, references to other frames or procedures) have to be specified or computed; and (3) properties (fillers, restriction on fillers, etc.) are inherited from superframes to subframes in the hierarchy according to some inheritance strategy. It is important to note that, besides the inheritance relation, frames do not provide the explicit concept of relations between frames but they use the concept of slot for representing these relations.

Based on the original proposal, several knowledge representation systems have been built and the theory of frames has evolved. Important descendants of frame-based representation formalisms are description logics that capture the declarative part of frames using a logic-based semantics. Moreover, the object-oriented paradigm has adopted the organizational principles introduced by frame-based systems.

An important branch in AI that is concerned with KR mechanisms deals with the so-called *ontologies* [Gru93]. An ontology is a data model that represents a set of concepts within a domain and the relationships between those concepts. It is used to reason about the objects within that domain. Ontologies are typically used to describe domain vocabularies and are actively used in artificial intelligence, the semantic web, software engineering and information architecture as a form of knowledge representation about the world or some part of it. The typical elements described by an ontology are:

- Individuals: the basic or "ground level" objects
- Classes: sets, collections, or types of objects
- Attributes: properties, features, characteristics, or parameters that objects can have and share
- Relations: ways that objects can be related to one another

Typically attributes are used to represent primitive values whereas relations are used to relate non-primitive classes. Some common relationships between concepts are classification (instance-of, member-of), aggregation (part-of), generalisation (is-a, subclass-of, a.k.a. subsumption), and partitioning (group, context) of the concepts.

From a software engineering point of view, and more in particular the MDE point of view, domain modelling is concerned with describing the conceptual view of the problem domain by means of a modeling language. The most representative general-purpose modeling language is the Unified Modeling Language (UML), which has become the 'de facto' modeling language for modeling software systems. The UML can also be used as a basis for domain-specific extensions — by means of the definition of stereotypes and profiles — and reuse. Other possible modeling languages can be mentioned, which might be preferred over UML for specific domains or applications: Object Role Modelling (ORM [Hal01]) and Entity Relationship Modeling (ER [Che76]).

#### 5.1.1.3 Domain-Specific Languages

An important path in MDE which focuses on addressing the issues of a particular domain is the one on *domain-specific languages (DSLs)* [vDKV00]. A DSL is designed to offer appropriate notations and abstractions inherent to a particular domain. It is tailored for a particular domain and therefore captures precisely the semantics of that domain. A DSL allows software development to be done quickly and effectively, yielding programs that are easy to understand, reason about, and maintain [Hud96]. Different DSL approaches can be found, which focus on either problem domains (from the real-world) such as for example the financial domain [vD97], or application domains such as the domain of software architectures (e.g. [MR97]), distributed applications [Fuc97] and transaction management [Fab05; FC05].

### 5.1.2 Transformations

Models are specified at different levels of abstraction and sometimes also in different languages. Transformation from one or multiple source models to one or multiple target models, called model transformation is an important issue within MDE. In Kleppe et al. [KWB03] the following definition of model transformation is provided:

> *"A transformation is an automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language"*

In [Sch06], Schmidt remarks the essential role played by transformations in MDE, as *they are able to take models as input and synthesize various types of artifacts as output — such as source code, simulation inputs, XML deployment descriptions, or alternative model representations. The ability to synthesize artifacts from models helps ensure the consistency between application implementations and analysis information associated with functional and QoS requirements captured by models. This automated transformation process is often referred to as "correct-by-construction", as opposed to conventional handcrafted "construct-by-correction" software development processes that are tedious and error prone.*

#### 5.1.2.1 Classifying Transformations

Transformations can be classified according to different characteristics, as presented in [MCG05]. In this taxonomy, two orthogonal dimensions are defined:

- Horizontal versus vertical:

  – Horizontal transformation indicates transformation between different models at the same level of abstraction. Model refactoring is an example of such a transformation because the source model is restructured and the target models are at the same level of abstraction. In the next section, we go into more detail on model refactoring.

  – Vertical transformation indicates a transformation where the source and target models reside at different levels of abstraction. Refinement is an example of such a transformation. The original model and its refined version are at different levels of abstraction.

- Rephrasing versus translation:

  – Rephrasing indicates a transformation where the models are expressed in the same modelling language. This kind of transformation is also called an endogenous trans- formation. Examples of rephrasing are optimisation, which aims at improving certain operational properties while preserving the semantics of the software, and refactoring which aims at improving certain software quality characteristics while preserving the software's behaviour.

– Translation indicates a transformation where the source and target models are ex- pressed in different languages. This kind of transformation is also called an exogenous transformation. Examples of translation are reverse engineering which extracts a higher-level specification from a lower-level one, and migration which translates a program written in one language to another, while keeping the same level of abstrac- tion.

In this dissertation we propose transformations that take as input high-level specifications — expressed in the proposed high-level rule and rule connection languages — and produce, as output, concrete Java and JAsCo code respectively. Therefore, as we move from a higher level of abstraction to a lower one, they fit into the category of *vertical* transformations. Moreover, our transformations operate on source specifications that are expressed in a different language than the target one, and therefore also classify as *translations*.

## 5.2 Domain Entities

In our approach, the domain vocabulary of interest can be explicitly captured as *domain entities*. Domain entities are the building-blocks used in the definition of the high-level rules and their connection with the core application. We propose the definition of domain entities that are based on the modeling elements typically found in data modeling approaches: *domain class*, *domain property* and *domain operation*. Note that as our domain model builds on top of an existing object-oriented application, the chosen domain entities have an object-oriented flavour. In section 5.1.1.2 we have mentioned two approaches for the explicit representation of domain knowledge: the slot-based approach (from frame-based systems) and the relation-based approach (from ontologies). Our approach is based on the former and therefore uses attributes to represent relations. The only explicit relation between domain classes that is supported by our approach is the inheritance relation.

A *domain class* defines a set of domain properties and a set of domain operations and can have many instances. A *domain property* describes a common property or characteristic of the instances of a domain class whereas a *domain operation* represents a behavior that can be performed by instances of the domain class. They can be either used to extract domain knowledge present in the implementation of an existing application or to express new domain vocabulary that needs to be constructed as a result of domain evolution. These domain entities and their relations are depicted in the upper part of the metamodel depicted in Figure 5.1.

Figure 5.2 shows a schematic view of domain entities typically found in the e-commerce domain. Example domain classes are *Customer*, *Product*, *ShoppingBasket*, *ShopAccount* and *Shop*. A *Customer* typically defines domain properties such as the *name*, *age* and *account* and defines domain operations to *login*, *logout*, *add a product to his/her shopping basket* and *become frequent*. The *ShopAccount* domain class defines the *amountSpent* and *boughtProducts* domain properties, whereas *ShoppingBasket* defines the *applyDiscount(discount)* domain operation and defines a domain property holding a relation to the *customer* that owns it. The *Product* domain class defines the *price* domain property and finally the *Shop* domain class defines operations for the *checking out of a shopping basket* and another one

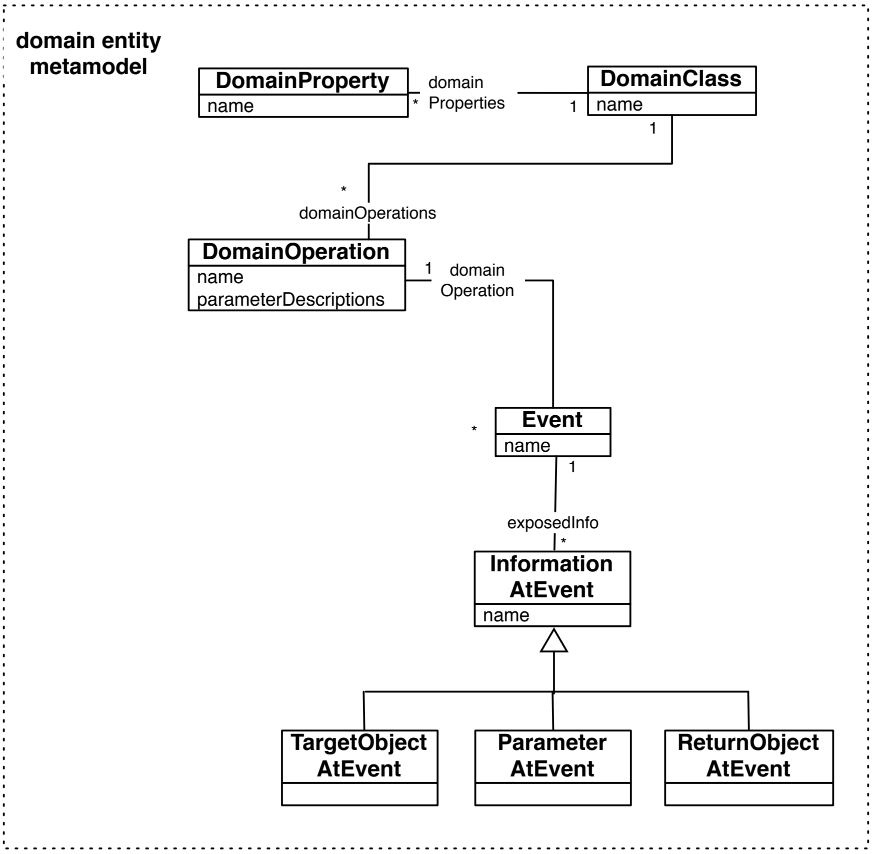


*Figure 5.1:* Domain entity metamodel

for a special kind of checkout called *checkout express* that allows a customer to proceed with the checkout without having to enter all the payment details. Note that the graphical notation used in Figure 5.2 aims at illustrating the slot-based nature of the domain entities. Note that in our approach, the relations between domain classes are implicit, since they are determined by the way domain entities are mapped to implementation (described in chapter 6).

Domain entities in our approach are mapped to implementation entities. This mapping can be a simple one-to-one link to an existing implementation entity but it can also be more complex in the case of domain entities that do not nicely map to one well-identified implementation entity or that are even unanticipated in the existing implementation. For these more complex mappings, support for specifying them completely at the domain level is provided, as it will be explained in chapter 6.

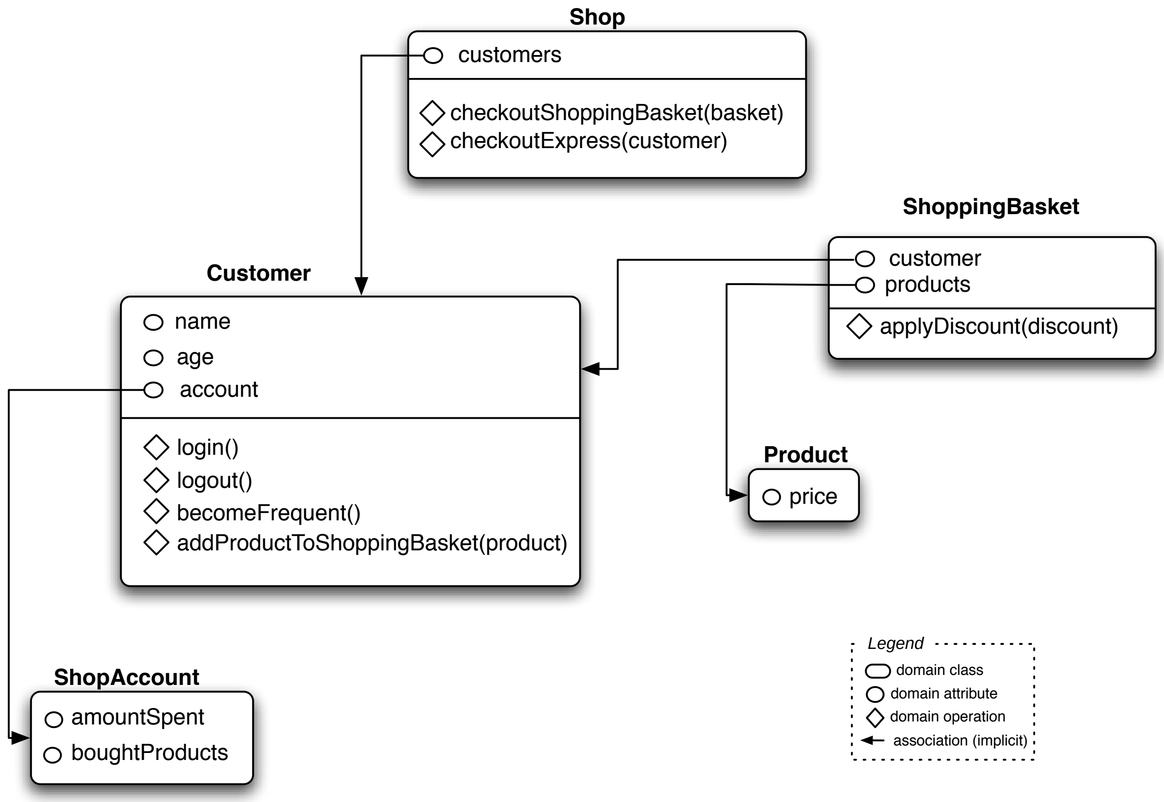


*Figure 5.2:* Graphical representation of some typical domain entities in the e-commerce domain

Because rules are connected at well-defined points in the execution of the core application, there is the need for modelling at the high level the concept of *event.* In our approach this is modelled by a special kind of domain entity that denotes the execution of a domain operation. This is illustrated in Figure 5.1. Events are defined in terms of existing domain operations defined in the domain model. Also, they can expose information available in the execution context of that operation, namely target object, parameters and return value. An example event is the *Checkout* event (defined in the fragment below) which captures the execution of the *checkOutShoppingBasket(shoppingBasket)* domain operation defined in the *Shop* domain class and which exposes the target under the name *shop*, the first parameter as *basket* and the return value as *total*.

```
EVENT Checkout AT Shop.checkOutShoppingBasket(shoppingBasket)
EXPOSING TARGET AS shop
      PARAMETER 0 AS basket
      RETURN VALUE AS total
```

## 5.3 High-Level Business Rules

The idea of defining rule-based knowledge in terms of a high-level rule language is not new and therefore present in some existing approaches, such us JRules, QuickRules, VisualRules, JBoss rules and HaleyRules (as explained in detail in chapter 9). Although these approaches allow specifying rules in terms of domain concepts described in a business model, the high-level terms are simple aliases for implementation entities and thus a one-to-one mapping between them is required. These one-to-one mappings are not enough to: i) represent domain concepts that have a more complex realization at the level of the implementation, and ii) represent unanticipated domain concepts required in the specification of high-level rules.

We propose a high-level rule language that is able to talk about domain concepts — represented as domain entities — that have a more complex mapping to implementation. A prototype of this language has been implemented (described in chapter 7). The main contribution of this section is the presentation of the features of our high-level rule language and the argumentation of their need. The concrete syntax of the implemented prototype of this language is not a contribution, as it is based on existing languages (e.g. OCL [OMG03]). Examples expressed in this language are given.

The different elements of a high-level business rule are depicted in the business rule metamodel shown in Figure 5.3. This figure also shows the relations that exist between the metamodels: a rule is defined in terms of domain entities and therefore, relations exist between the business rule metamodel and the domain entity metamodel.

### 5.3.1 Rule

As proposed by other current high-level rule languages [ILO; YAS; Inn; JBob; Halb], we define a high-level rule as a statement of the form:

$$IF < condition > THEN < action >$$

The action of the rule is only triggered when the condition is met. The following simplified grammar defines the expressions allowed in the condition and action parts of the rule (the complete concrete grammar of the prototype implementation of the high-level business rule language can be found in Appendix A):

$$\begin{aligned}
< condition > \quad &:= \quad < singleCondExp > [(AND|OR|XOR) < singleCondExp >]^* \\
< singleCondExp > \quad &:= \quad [NOT] < compExp > \\
< action > \quad &:= \quad < singleAction > [AND < action >]^*
\end{aligned}$$

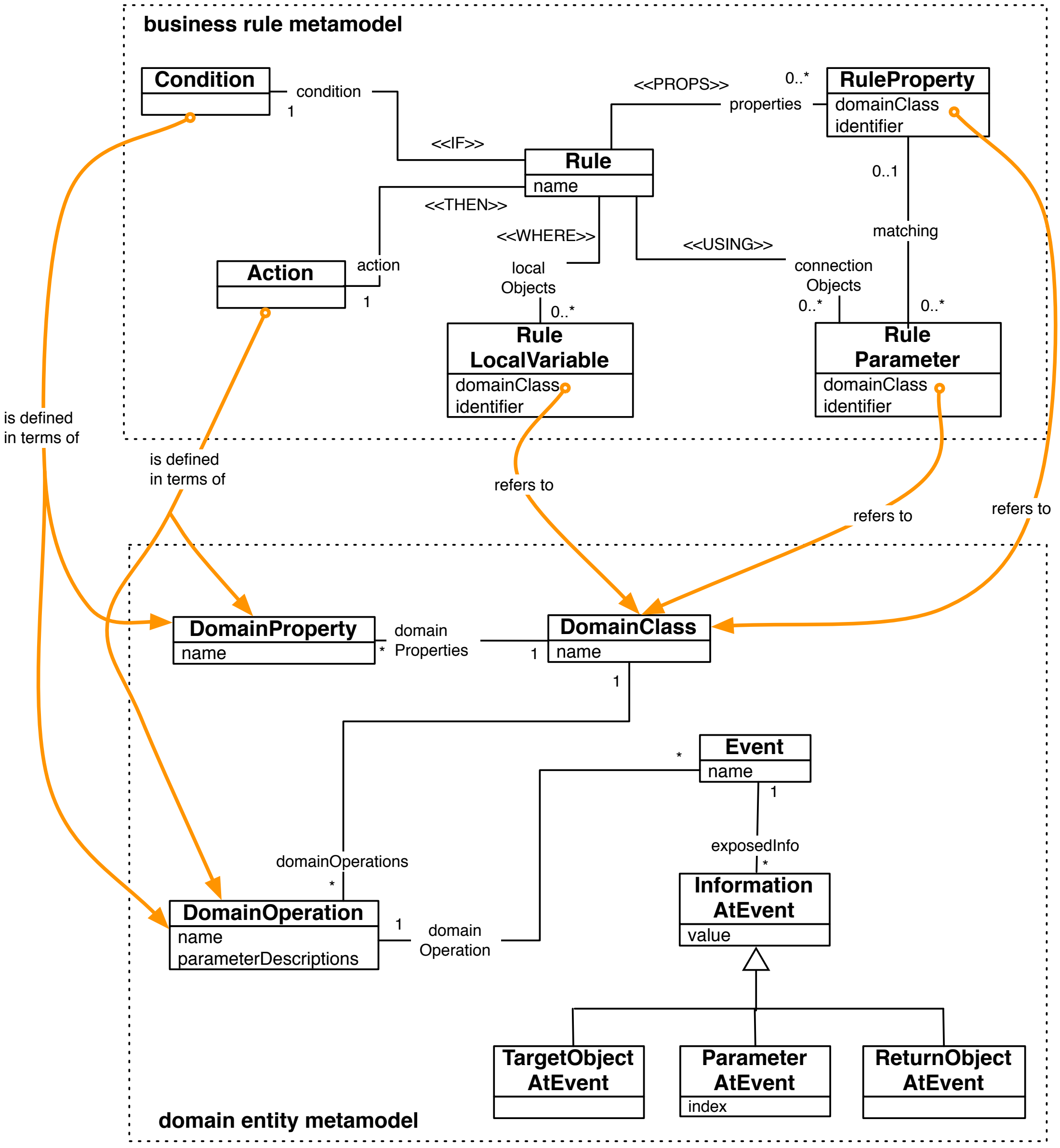


*Figure 5.3:* Business rule and domain entity metamodels and their relations

The condition denotes a boolean expression that can involve the invocation of domain operations, the retrieval of domain properties and the reference to domain objects (instances of domain classes) specified in the rule. Also these elements can be combined in logical or comparison expressions as well as in nested combinations. An example condition is:

```
customer.account.amountSpent() >= 100 OR customer.account.boughtProducts >= 10
```

where the result of two comparisons are related in an *or* expression. Each comparison involves the result of navigations over the domain model. Similarly to OCL [OMG03], the dot notation is used in our language to express navigations. In this condition, *customer.account* refers to the domain property *account* in *customer*, which is an instance of the *Customer* domain class; *customer.account.amountSpent* refers to the result of retrieving the domain property *amountSpent*, defined in the domain class *ShopAccount*, on the *account* property of the *customer* domain object.

The action part denotes the invocation of domain operations that can involve accessors, references to domain objects and domain operation invocations. An example action is:

```
basket.applyDiscount(discount) AND customer.becomeFrequent()
```

where *basket* and *customer* are instances of the domain classes *ShoppingBasket* and *Customer* respectively and where *applyDiscount(discount)* and *becomeFrequent()* are domain operations defined in those domain classes, respectively.

The *localAssignment* action is a special kind of rule action that allows assigning a new value to a *localRuleObject*. A *localRuleObject* denotes any domain object available in the context of the rule — either a rule property, parameter or local variable (explained in the coming sections 5.3.2, 5.3.3 and 5.3.4 respectively). As an example, suppose a rule receives as parameter a real value under the name *totalAmount*. Then, its action part can include the expression:

```
totalAmount IS (totalAmount - 10)
```

This action specifies that the total amount must be reduced by 10. Note that this specification does not explicitly indicate how this new value should be passed to the context of the caller which triggers this rule. It is not the responsibility of the rule to specify how this change should be reflected. This is taken care of in the rule connection, as explained in the coming section 5.6.

The relation between the condition and action parts of a rule and the involved domain entities is depicted in Figure 5.3 by the relations labeled with *"is defined in terms of"* which relate the *Condition* and *Action* metaclasses to the *DomainProperty* and *DomainOperation* metaclasses from the domain entity metamodel.

### 5.3.2 Rule Properties

A rule condition typically defines a comparison between some domain entity and a hard-coded value, or similarly, a rule action involves a hard-coded value. In order to avoid the repetition of the same logic in many rules that only vary in these hard-coded values, rules are parameterized with rule properties. In our language rule properties are defined using the following clause:

$$PROPS < domainClassName > AS < rulePropertyName >$$

Defining rule properties enables the definition of rule templates which capture the main business rule logic and leave out the deployment details, which need to be specified at rule instantiation time. The definition of rule properties is depicted in the upper part of Figure 5.3 by the association between the metaclasses *Rule* and *RuleProperty.* Note also the relation labeled *"refers to"* that exist between the *RuleProperty* and the *DomainClass* metaclasses which link the two metamodels.

### 5.3.3 Rule Parameters

Rules are parameterized with values from the context in which they are going to be executed. In our language, this is done by means of the following clause:

$$USING < domainClassName > AS < ruleParameterName >$$

This is depicted in the upper part of Figure 5.3 by the association between the metaclasses *Rule* and *RuleParameter.* The details on how these parameters are provided to the rule at rule connection time are presented in section 5.4. Note also the relation labeled *"refers to"* that exist between the *RuleParameter* and the *DomainClass* metaclasses which link the two metamodels.

### 5.3.4 Rule Variables

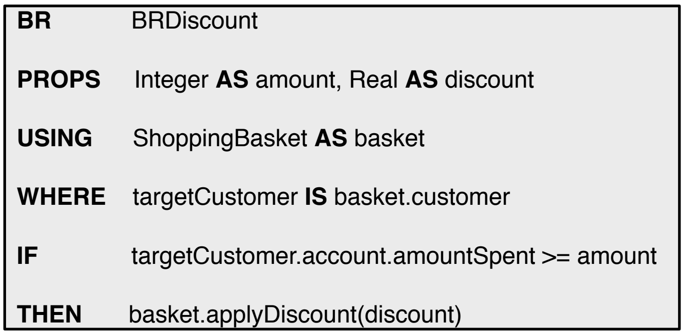
Optionally, in order to ease the manipulation of domain objects, a rule can assign a local name to objects available in its context. For example, a local name can be assigned to the result of a navigation along the domain model. This is done by means of the following clause:

$$WHERE < variableName > IS < navigationInDomainModel >$$

where a *navigationInDomainModel* defines a domain model navigation path which has as a starting point either a *ruleParameterName* or a *rulePropertyName.* This enables referring to the local variable name everywhere in the rule where that navigation is needed. The definition of rule variables is depicted in Figure 5.3 by the association between the metaclasses *Rule* and *RuleLocalVariable.* Note also the relation labeled *"refers to"* that exist between the *RuleLocalVariable* and the *DomainClass* metaclasses which link the two metamodels.

An example high-level rule, *BRDiscount*, is shown in Figure 5.4. It applies a discount on a customer's shopping basket if the customer has already spent more than a certain amount of money. This rule involves the identified e-commerce domain entities shown in Figure 5.2.

In this section we have presented the features of the proposed high-level business rule language. As this language is inspired on existing languages [ILO; YAS; JBob; Halb], the presented features are similar to the ones provided by those languages. The main difference is the ability to refer to domain entities that are not just simple aliases of implementation entities but can have a more complex mapping to implementation. Among the presented



```
BR        BRDiscount

PROPS     Integer AS amount, Real AS discount

USING     ShoppingBasket AS basket

WHERE     targetCustomer IS basket.customer

IF        targetCustomer.account.amountSpent >= amount

THEN      basket.applyDiscount(discount)
```

*Figure 5.4:* BRDiscount rule expressed in the high-level rule language

features, the main contribution is the possibility of proving the rule with values taken from the connection context at which the rule application is triggered.

## 5.4 High-Level Business Rule Connections

So far we have shown how AOP can successfully achieve the decoupling of the crosscutting rule connection code and presented aspect patterns as guidelines for the implementation of rule connection aspects (chapter 4). Even though these proposed aspects are a good solution to the problem of crosscutting connection code, they are entirely expressed at the programming level, and thus also exclude the domain expert. Moreover, as many different connection elements need to be taken into account as part of the connection aspects, it is hard also for application engineers to write these aspects by hand. In order to overcome these limitations, in this section we propose abstracting the recurrent rule connection elements as features of a high-level rule connection language. This language allows expressing rule connections as separate and explicit entities at the domain level. Separating rules from their connections (also at the domain level) allows reusing both parts independently.

The *high-level business rule connections* specify the details of the rules' integration with the core application and typically denote an event at which the rules need to be applied, the exact moment when the rule needs to be applied at that event, and the specification of how the required rule information is made available to the rule.

In chapter 4, six rule connection elements — part of a connection aspect — were identified:

A) determining the rule application time

B) restricting rule application time to a context designated by the rule activation time

C) making required information available to the rule

D) triggering rule

E) retrieving rule results from the rule

F) proceeding with the core application's execution

The first three connection elements are the variable parts of any rule connection and also determine the way the other elements need to be tackled. Thus, the high-level rule connection language only provides features for those three variable elements and not for the rest, since the latter are dependent on the former.

The high-level rule connection features are depicted in the rule connection metamodel shown in Figure 5.5. This figure also shows the relations that exist between the different metamodels: that rule connection refers to events defined as domain entities and therefore, relations exist between the two metamodels, i.e. the business rule connection metamodel and the domain entity metamodel. Similarly, a rule connection needs to comply with what it is defined in the rule that is being connected, e.g. the number and types of properties passed to the rule needs to coincide with those defined in the rule, the objects in the connection context are mapped to the expected objects by name. Thus, relations exists also between the rule connection metamodel and the rule metamodel.

A prototype of this language has been implemented (described in chapter 7). The main contribution of this section is the presentation of the features of our high-level rule connection language and the analysis of how they vary.

### 5.4.1 Rule Connection

The first feature simply specifies the actual rule that needs to be connected. This is specified as follows:

$$CONNECT < brname >$$

where *brname* is the name of the rule to be connected. If *brname* corresponds to a rule template, then concrete values need to be defined for the rule, as follows:

$$PROPS < value1 >, ..., < valueN >$$

Thus, this clause is used to instantiate the rule template to an actual rule using those values.

The relations between the metaclass *RuleConnection* and the metaclasses *Rule* and *RuleProperty* depict these two presented features (Figure 5.5).

### 5.4.2 Connection Event

A rule is applied at a well-defined point in the execution of the core application. In our domain model this well-defined point corresponds to the execution of a domain operation and is expressed by an *event*. From the point of view of the business analyst, events represent points in time where it is likely to have business logic applied. In what follows we refer to the execution of the domain operation captured by the event as *event execution*. We identify three ways in which a rule can be connected at an event:

a) *before* an event, meaning that the rule's condition is checked just before the execution of the event, which is then immediately followed by (in case the condition is met) the execution of the rule's action. For example, a rule can be connected *before a customer is checking out*, meaning the point in time just before the domain operation *checkout(shoppingBasket)* defined in the domain class *Shop* is executed

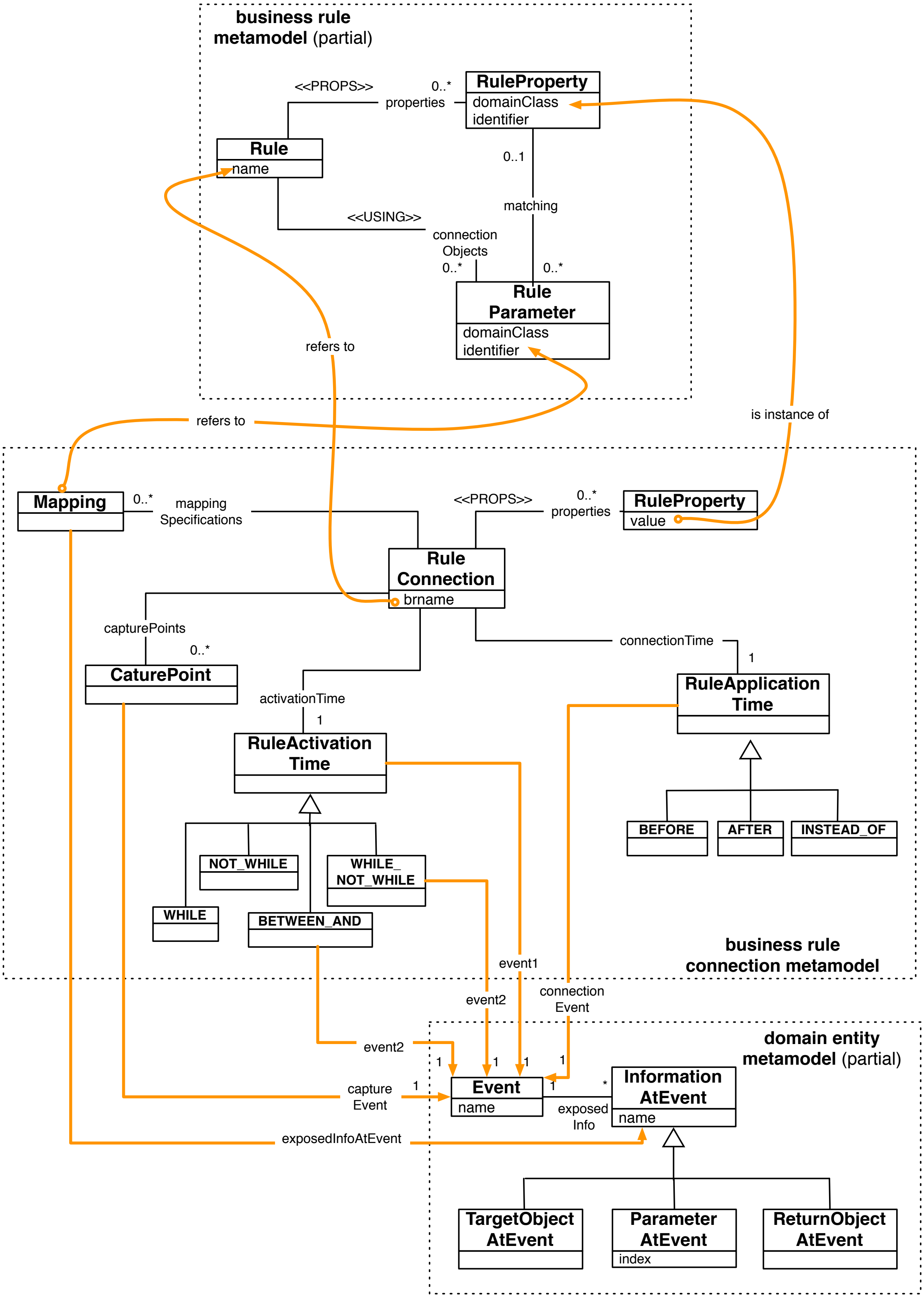


*Figure 5.5:* Business rule connection metamodel and its relations to the domain entity metamodel and the business rule metamodel

b) *after* an event, meaning that the rule's condition is checked just after the execution of the event, which is then immediately followed by (in case the condition is met) the execution of the rule's action. For example, a rule can be connected *after a customer logs in*, which maps to the point in time just after the domain operation *logIn()* is executed on a customer.

c) *instead of* an event, meaning that the core application is interrupted just before the execution of the event, the rule's condition is checked and if met, its action is triggered, completely replacing the original behavior captured by the event. For instance, a payment rule encapsulating a new payment policy can be connected instead of *the payment process*, which means in replacement of the execution of the domain operation *proceedPayment()* in *Shop*.

We provide constructs for these three variations of the rule application time feature, as follows:

$$[BEFORE|AFTER|\text{INSTEAD OF}] < \textit{eventname} >$$

where *eventname* is the name of the event at which to connect the rule. The event is a domain entity explicitly captured in the domain model. The rule application time feature and its variations are depicted by the hierarchy of *RuleApplicationTime* metaclasses shown in the rule connection metamodel (Figure 5.5). The relation from the *RuleApplicationTime* metaclass to the *Event* metaclass depicts the fact that a rule application time is defined in terms of an event.

Note that because in this dissertation a bottom-up approach is taken, these features are inspired on the different ways a rule can be connected at the implementation level, namely from a *before*, *after* or *around* advice (as shown in chapter 4). However, it is important to stress that these high-level connection features are higher-level abstractions for those kinds of advice and thus a one-to-one mapping does not necessarily exist between them (as it is explained in section 5.6.2).

### 5.4.3 Rule Activation Time

The application of a given rule can be restricted to certain contexts. For instance, a discount rule — which would typically be applied when the product price is retrieved — can be restricted only to those price retrievals that occur *while the customer is checking out*, or within the period of time *between the moment the customer logs in and the moment he/she adds a product to the shopping cart*, or *not while the customer is browsing the products*. In our connection language, the applicability context of a rule is referred to as *activation time*. The specification of the activation time is optional and when not specified it is assumed that the rule is always active. The activation time is defined in terms of one or more events in one of the four following ways:

1.

$$\text{ACTIVATE WHILE} < \textit{event} >$$

meaning that the rule is active during the period of time denoted by the execution of *event*.

2. $$\text{ACTIVATE NOT WHILE} < event >$$

   meaning that the rule is active not while *event* is executing.

3. $$\text{ACTIVATE WHILE} < event1 > \text{AND NOT WHILE} < event2 >$$

   meaning that the rule is considered active while *event1* is executing but not while *event2* is executing.

4. $$\text{ACTIVATE BETWEEN} < event1 > AND < event2 >$$

   meaning that the rule is active during the period of time initiated by the execution of *event1* and terminated by the execution of *event2*.

The hierarchy of *RuleActivationTime* metaclasses and their relation to the metaclass *Event* depict this feature in the rule connection metamodel (Figure 5.5).

### 5.4.4 Connection-Specific Information

A rule expects to receive the information declared in the USING clause at rule connection time. At the moment the rule is connected at an event, two situations can occur:

i the required information is available in the context of the connection event and thus it can be directly passed to the rule. The kind of information that can be passed to the rule depends on whether the rule is connected before, after or instead of an event: in case of a connection *before* or *instead of* an event, the parameters and the receiver of the domain operation are exposed by the event and thus can be passed to the rule, whereas if the rule is connected *after* an event, the parameters, the receiver *and* the result of invoking the domain operation are available.

ii the rule requires information that is not available in the context of the connection event: in order to capture this unavailable information, *capture points* are defined as an extra component of the connection specification, as follows:

$$\text{CAPTURE AT} < event1 >, ..., < eventN >$$

where *event1, ..., eventN* are names of events that capture the moment when the required information is reachable, and expose it.

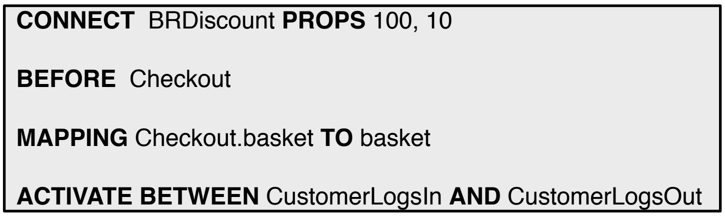
Independently of whether the information is contextual or captured, it needs to be mapped to the information required by the rule. This is done by linking the available/required information in *mapping* specifications of the form:

$$MAPPING < event > . < infoExposed > TO < infoRequired >$$

These features are depicted in the rule connection metamodel (Figure 5.5) by the metaclasses *CapturePoint* and *Mapping* and their relations.

The example shown in Figure 5.6 specifies the connection of the *BRDiscount* rule at the *Checkout* event (defined in section 5.2). This discount should only be considered in the period of time starting from the moment the customer becomes frequent — which is denoted by the event *CustomerBecomesFrequent* — and the moment the customer logs out — denoted by the event *CustomerLogsOut*. These two events are defined as follows:

```
EVENT CustomerLogsIn AT Customer.login()

EVENT CustomerLogsOut AT Customer.logout()
```



```
CONNECT  BRDiscount PROPS 100, 10

BEFORE  Checkout

MAPPING Checkout.basket TO basket

ACTIVATE BETWEEN CustomerLogsIn AND CustomerLogsOut
```

*Figure 5.6:* High-level connection of BRDiscount rule at `Checkout` event

In this section we have presented the features of our high-level business rule connection language. To our knowledge, the idea of expressing the connection of the rules in terms of domain concepts has not been proposed before. The main contribution of this section is the analysis of which features need to be provided by this language. These features are abstractions that build on top of the recurrent connection elements identified in the aspect patterns presented in chapter 4. Although a prototype implementation of this language is provided, its concrete syntax is not the main contribution.

## 5.5 Transforming the High-Level Domain Model

The need for achieving a clear separation of concerns does not only apply to the code artefacts of a model-driven system but also to the models themselves [KR03]. This is a crucial requirement towards facilitating traceability, reuse, and evolution. We adhere to this goal and propose the automatic translation from high-level rules and their connections to executable implementations in OOP and AOP respectively, as explained in sections 5.6.1 and 5.6.2. SoC is achieved at both levels: at the domain level the rules are separated from their connections whereas at the implementation level rules are encapsulated in rule objects and rule connections are cleanly encapsulated in aspects.

When crosscutting domain abstractions are expressed at a higher level of abstraction, their translation to implementation is not straightforward. We identify and tackle challenges in the translation process from high-level rules and connections to implementation.

In our approach, MDE is partially applied since automatic code generation is pursued only for the integration of high-level rules and their connections and not for the generation of the entire application. The existence of a core implementation which is developed and maintained using standard software engineering techniques is assumed.

### 5.5.1 Introduction to Transformation Systems

Transformation systems typically divide the transformation process into a set of transformation modules or *transformations*. In its most general form, each transformation takes an input and produces an output. Depending on the concrete technique used for implementing the transformation, the input and output can correspond to one or more fragments written in the corresponding languages. This distinction gives raise to two important characteristics of the transformation process which have a direct impact on the division into transformation modules: *granularity* and *scope*.

#### 5.5.1.1 Transformation Granularity

One of the ultimate goals of language engineering is the design of well-modularized languages. Well-modularized means that the actual implementation of the language is well-modularized. In the ideal case, each specific language feature is implemented by a different module. Well-modularized language implementations contribute to increasing the expressiveness of the languages as it enables the addition of new features or the replacement of existing ones without having to manually modify the implementation of other existing features [Cle07].

A possible way of modularizing the implementation of a language is as separate *transformations*. Transformations can be categorized into fine-grained and course-grained. Fine-grained transformations transform only a small part of the source model and produce only a small part of the target model. On the contrary, course-grained transformations transform a big part of the source model and produce a big part of the target model. Fine-grained transformations are well-modularized pieces of transformation logic and therefore are more reusable than course-grained transformations. Another advantage of fine-grained transformations with respect to course-grained ones is that the former ones can deal with variability in a better way: when a new feature is added to the source language, a new transformation that encapsulates the change can simply be added, requiring a minimum amount of implementation effort and avoiding having to refactor existing transformations. Because of these advantages, we can conclude that — in general — fine-grained transformations are preferred versus course-grained ones.

However, fine-grained transformation also present disadvantages: firstly, the smaller the input, the less information is available for performing the transformation. When this information is not enough, additional steps might be needed for compensating this loss. Secondly, the smaller the input fragments, the more the transformations will depend on the results of other transformations. This is because transformations that take small input fragments, produce small output fragments which often have to be combined with the outputs produced by other transformations in order to yield a complete fragment at the target side. These additional dependencies do not appear in a coarse grained transformation modularization. The larger the input, the larger the output and the bigger the chance that the output is a complete and independent model.

#### 5.5.1.2 Transformation Scope

The area covered by a single transformation step is called scope [vWV03]. First, scope should be considered in the *source* and the *target* of the transformation. The input scope denotes the area of the source model which is covered by the transformation. The output scope denotes the area of the target model affected or produced by the transformation. Second, a transformation step can have a *local* or *global* scope in both, *source* and *target*. To better understand these notions, the term pivot is introduced [BHW97]: *the pivot of a transformation is the main input element around which the transformation step revolves.* In [vWV03], van Wijngaarden et al. consider a transformation step to have a:

- *local source scope*: when it only requires the pivot (or extra information available in the subtree[1] of the pivot) as input.
- *global source scope*: when input information is located outside the subtree of the pivot.
- *local target scope*: when the output is localized in a single output node.
- *global target scope*: when the output is scattered in multiple output nodes.

This classification gives raise to all possible combinations: *local-to-local, local-to-global, global-to-local, global-to-global.* For more information about these categories, the interested reader is referred to [vWV03].

When using contemporary language development techniques, implementing the more complex *local-to-global*, *global-to-local* and *global-to-global* transformations is not an easy task. The transformation process is decomposed into a set of implicitly co-operating transformation modules [Cle07]. These implicit dependencies between the transformations complicates the implementation of the transformation process as a whole. Support for better separation of concerns in the implementation of the transformations is required, as identified and tackled in [Cle07].

## 5.6 Transforming High-Level Business Rules and their Connections

In this section we present transformations that take as input specifications in the proposed high-level rule and connection languages and produce as output rule objects — in Java — and aspects — in JAsCo. In our approach, we pursue our transformations to be fine-grained, which allows realizing the advantages discussed earlier (section 5.5.1.1). Moreover, we pursue a level of granularity that is aligned with the features of our high-level languages: each transformation is in charge of transforming one high-level feature. Moreover, we observe that most of the transformations are complex as they correspond to the categories of local-to-global and global-to-global transformations. We describe and depict each transformation and its dependences with other transformations. These transformations have been implemented in a prototype and can be performed completely automatically, as described in chapter 7.

### 5.6.1 Transforming High-Level Business Rules

In this section we present transformations from high-level rules to rule objects in Java. A different fine-grained transformation is proposed per high-level rule feature. Table 5.1 provides an overview of these transformations and shows to which category they correspond[2].

[1] The term subtree refers to a branch of the abstract syntax tree

[2] Note that information about the mapping for each of the domain entities involved in the rule is required as well as an extra input, but it is omitted in the table for simplicity.

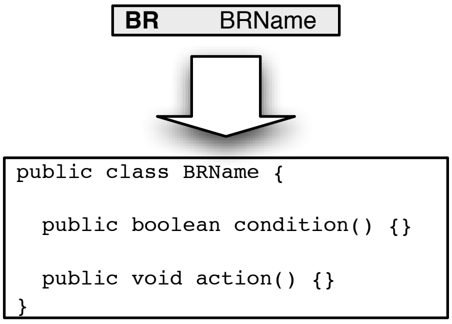
These translations are carried out fully automatically. Every time a domain property, a domain operation or an instance of a domain class is referred to in a rule, a value, a behavior or an object needs to be provided respectively. This is a mechanism which is based on traversing the mappings from the domain entities — involved in the definition of the rules — to implementation entities. These mappings are described exhaustively in the coming chapter 6. For now we make abstraction of their concrete details and assume the existence of a function *CodeRepresentation* in charge of taking a high-level expression, i.e. an expression only involving domain entities from the domain model, and returning an implementation expression in Java. This function uses the information specified in the mapping of the domain entities involved in the high-level expression in order to produce a code representation for it.

| ID | PIVOT | EXTRA INPUT | MAIN OUTPUT | EXTRA OUTPUT | KIND |
|---|---|---|---|---|---|
| 1 | *BR <br>* | none | - rule class defining method signatures for the condition and action | none | L-2-L |
| 2 | *PROPS*<br>*<DClass_1> AS <prop_1>,*<br>*...,*<br>*<DClass_n> AS <prop_n>* | name of business rule | - one variable per rule property<br>- rule constructor for setting the values received as parameters to those generated variables | none | G-2-G |
| 3 | *USING*<br>*<DClass_1> AS <conObject_1>,*<br>*...,*<br>*<DClass_k> AS <conObject_k>* | none | - one variable per connection object<br>- setter and getter per variable | none | L-2-G |
| 4 | *WHERE*<br>*<localVar_1> IS <path_1>,*<br>*...,*<br>*<localVar_j> IS <path_j>* | none | - one local variable per definition<br>- initialization method for initializing these local variables with values received as parameters | none | L-2-G |
| 5 | *IF <condition>* | none | - body of condition method | none | L-2-L |
| 6 | *THEN <action>* | none | - body of action method | none | L-2-L |

*Table 5.1:* Transformations from high-level rule constructs to OOP implementations

#### 5.6.1.1 Transforming BR

Following the rule object pattern [Ars01], a high-level rule is transformed into a class which defines methods implementing its condition and action, with return types boolean and void respectively. This is illustrated in Figure 5.7: the arrow depicts the transformation from the high-level construct to a Java implementation.

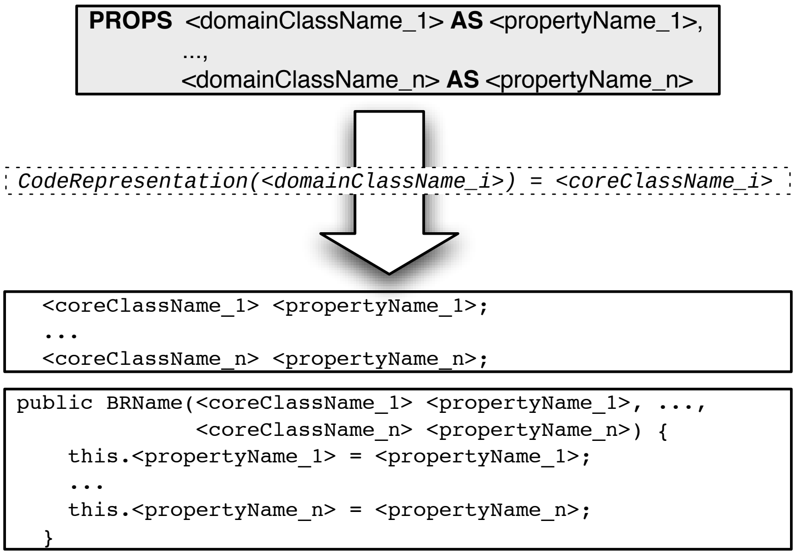


```
public class BRName {

  public boolean condition() {}

  public void action() {}
}
```

*Figure 5.7:* Transformation of 'BR' clause

#### 5.6.1.2 Transforming PROPS

For each property — defined in the *PROPS* clause — a local variable is created which is assigned to a concrete object in the constructor of the class. This is illustrated in Figure 5.8: two outputs are produced represented as independent boxes; the dotted box on top of the arrow shows the extra information that needs to be obtained during the execution of the transformation, in this case the mapping of the domain classes referred to in the *PROPS* clause.


**PROPS** <domainClassName_1> **AS** <propertyName_1>,
...,
<domainClassName_n> **AS** <propertyName_n>
*CodeRepresentation(<domainClassName_i>) = <coreClassName_i>*

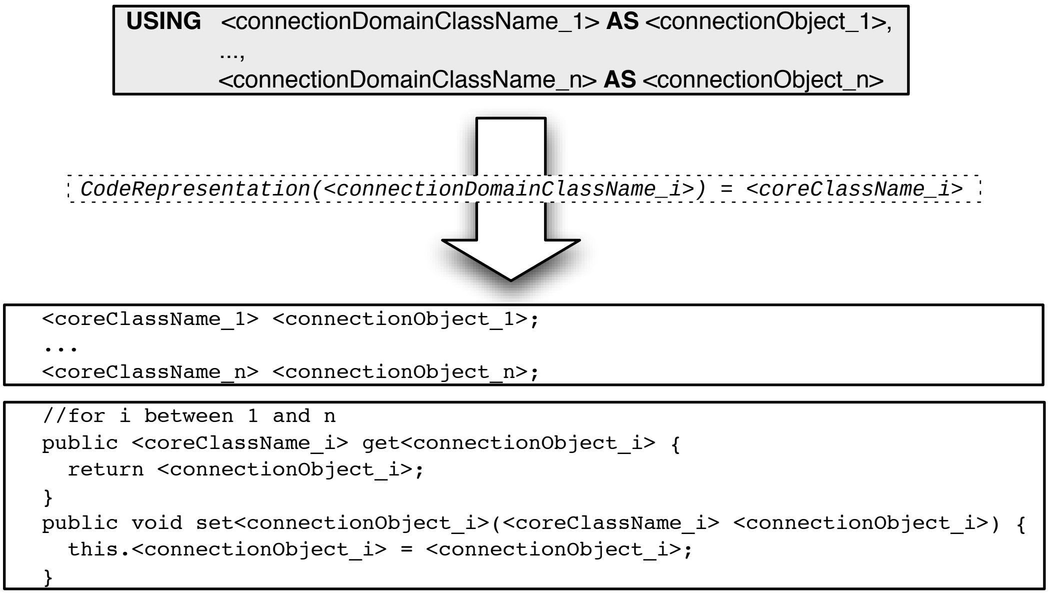
```
  <coreClassName_1> <propertyName_1>;
  ...
  <coreClassName_n> <propertyName_n>;
```

```
public BRName(<coreClassName_1> <propertyName_1>, ...,
              <coreClassName_n> <propertyName_n>) {
    this.<propertyName_1> = <propertyName_1>;
    ...
    this.<propertyName_n> = <propertyName_n>;
  }
```

*Figure 5.8:* Transformation of 'PROPS' clause

#### 5.6.1.3 Transforming USING

For each object expected at connection time — defined in the *USING* clause — a local variable and a getter and setter are generated (Figure 5.9). As before, the transformation needs extra information about the mapping of each of the domain classes referred to in the *USING* clause.



```
USING  <connectionDomainClassName_1> AS <connectionObject_1>,
       ...,
       <connectionDomainClassName_n> AS <connectionObject_n>
```

```
CodeRepresentation(<connectionDomainClassName_i>) = <coreClassName_i>
```

```
<coreClassName_1> <connectionObject_1>;
...
<coreClassName_n> <connectionObject_n>;
```

```
//for i between 1 and n
public <coreClassName_i> get<connectionObject_i> {
  return <connectionObject_i>;
}
public void set<connectionObject_i>(<coreClassName_i> <connectionObject_i>) {
  this.<connectionObject_i> = <connectionObject_i>;
}
```

*Figure 5.9:* Transformation of 'USING' clause

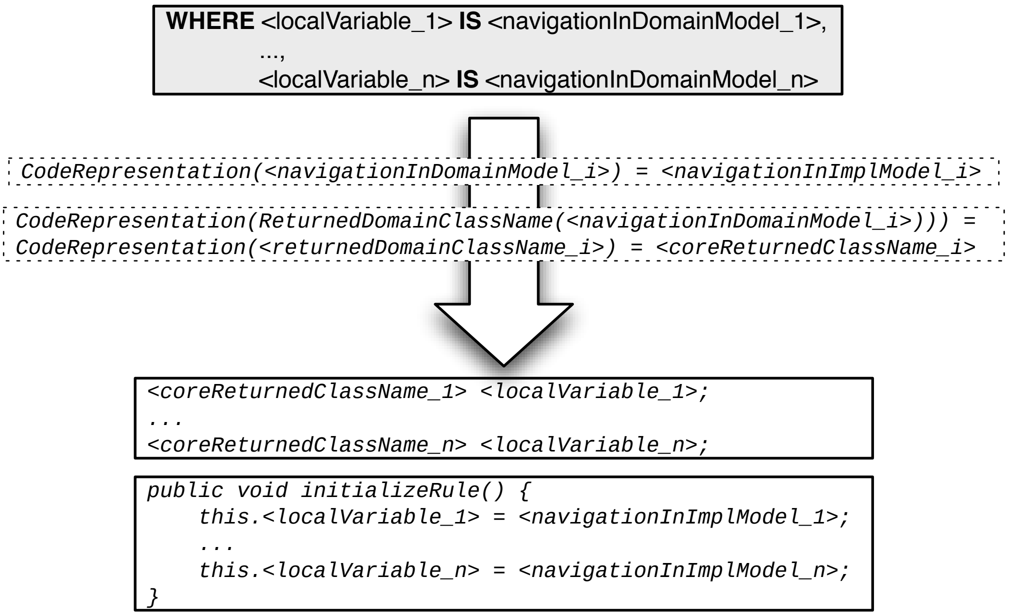
#### 5.6.1.4 Transforming WHERE

For every local variable defined in the *WHERE* clause, an attribute is added and an `initializeRule` method is included which initializes these attributes (Figure 5.10). This transformation needs to obtain a representation of the implementation of each of the domain model navigations involved in the *WHERE* clause. This is obtained by applying the *CodeRepresentation* function. Also, for each domain model navigation, the domain class that represents the type of the return value needs to be obtained which in turn is translated to its implementation, by applying the *CodeRepresentation* function again.

#### 5.6.1.5 Transforming IF and THEN

The bodies of the condition and action methods include the concrete implementations that result from obtaining the mappings of the domain entities referred to in the *IF* and *THEN* clauses respectively (Figure 5.11). In order these transformations to execute, each of the conditions and actions needs to be translated to their representation in terms of implementation entities. This is indicated by applying the *CodeRepresentation* function onto these conditions and actions.

A concrete example of the translation from the the *BRDiscount* high-level rule to a rule object in Java is shown in Figure 5.12. The individual transformations for each of the parts



```
WHERE <localVariable_1> IS <navigationInDomainModel_1>,
      ...,
      <localVariable_n> IS <navigationInDomainModel_n>
```

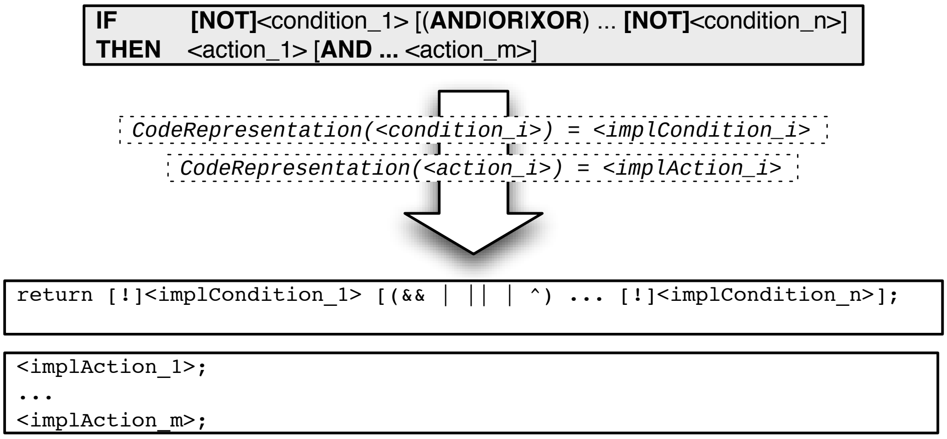
```
CodeRepresentation(<navigationInDomainModel_i>) = <navigationInImplModel_i>
```

```
CodeRepresentation(ReturnedDomainClassName(<navigationInDomainModel_i>))) =
CodeRepresentation(<returnedDomainClassName_i>) = <coreReturnedClassName_i>
```

```
<coreReturnedClassName_1> <localVariable_1>;
...
<coreReturnedClassName_n> <localVariable_n>;
```

```
public void initializeRule() {
    this.<localVariable_1> = <navigationInImplModel_1>;
    ...
    this.<localVariable_n> = <navigationInImplModel_n>;
}
```

*Figure 5.10:* Transformation of ‘WHERE’ clause



```
IF      [NOT]<condition_1> [(AND|OR|XOR) ... [NOT]<condition_n>]
THEN    <action_1> [AND ... <action_m>]
```

```
CodeRepresentation(<condition_i>) = <implCondition_i>
```

```
CodeRepresentation(<action_i>) = <implAction_i>
```

```
return [!]<implCondition_1> [(&& | || | ^) ... [!]<implCondition_n>];
```

```
<implAction_1>;
...
<implAction_m>;
```

*Figure 5.11:* Transformation of ‘CONDITION’ and ‘ACTION’ clauses

described in the high-level specification are followed and their outputs are put together in order to obtain the resulting rule object. This diagram clearly shows that these translations are examples of *local-to-global* transformations, as for a single input construct, many non-localized outputs are generated. The package *ecommerce* contains the core implementation classes, to which domain entities map.

### 5.6.2 Transforming High-Level Business Rule Connections

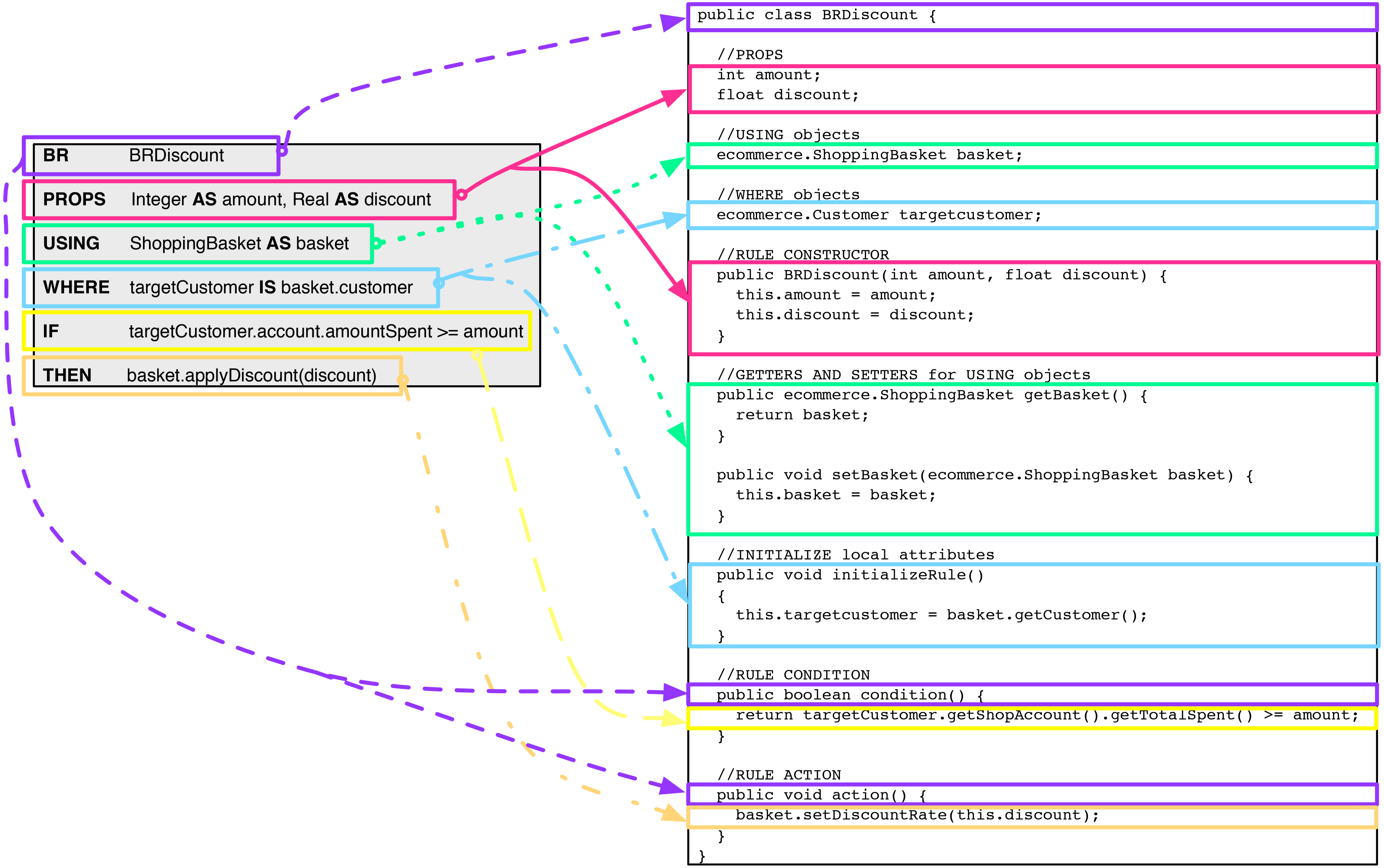
In this section we present the automatic transformation from high-level rule connections to aspects. The main contribution of this section is the transformations themselves and the analysis of the dependencies that exists between them. The actual output of the transformations corresponds to different instantiations of the aspect patterns presented before (chapter 4). The use of AOP as a target paradigm in this transformation is completely transparent for the domain expert, as the AOP peculiarities are not exposed in the high-level rule connection language (as described in Section 5.4). We illustrate these transformations using JAsCo.

As already hinted in chapter 4, the connection elements are not completely independent from each other. This is because the way a specific connection element is implemented influences the way other connection elements need to be implemented. We pursue the definition of fine-grained transformations where each of them takes as input a high-level rule connection construct and produces as output an AOP-based implementation. However, we observe that the finer the transformations, the bigger their scope. This impedes analyzing the transformations completely in isolation from each other. These dependencies complicate the transformation process as a whole.

We propose five transformations, one per high-level feature. For three of the high-level features, a number of variations are identified and therefore more-specific transformations are proposed to tackle each specific case. Table 5.2 gives an overview of these transformations from high-level rule connection constructs to AOP-based implementations[3].

We observe that some transformations require as input — in addition to the pivot — extra information about the configuration of other features in order to be able to produce a concrete output. This is the case with transformation (4), which needs to analyze the actual rule that is being connected — that is not part of the rule connection specification — and use the result of that analysis as an extra input for the transformation. Moreover, with respect to their output, we observe two complex scenarios: i) the outputs of two different transformations need to be combined to produce one final result (e.g. transformations (3) and (4)); ii) a transformation produces an incomplete output with 'gaps' that need to be filled in by outputs produced by other transformations (e.g. transformation (1)). In the rest of this section we explain these situations in more detail.

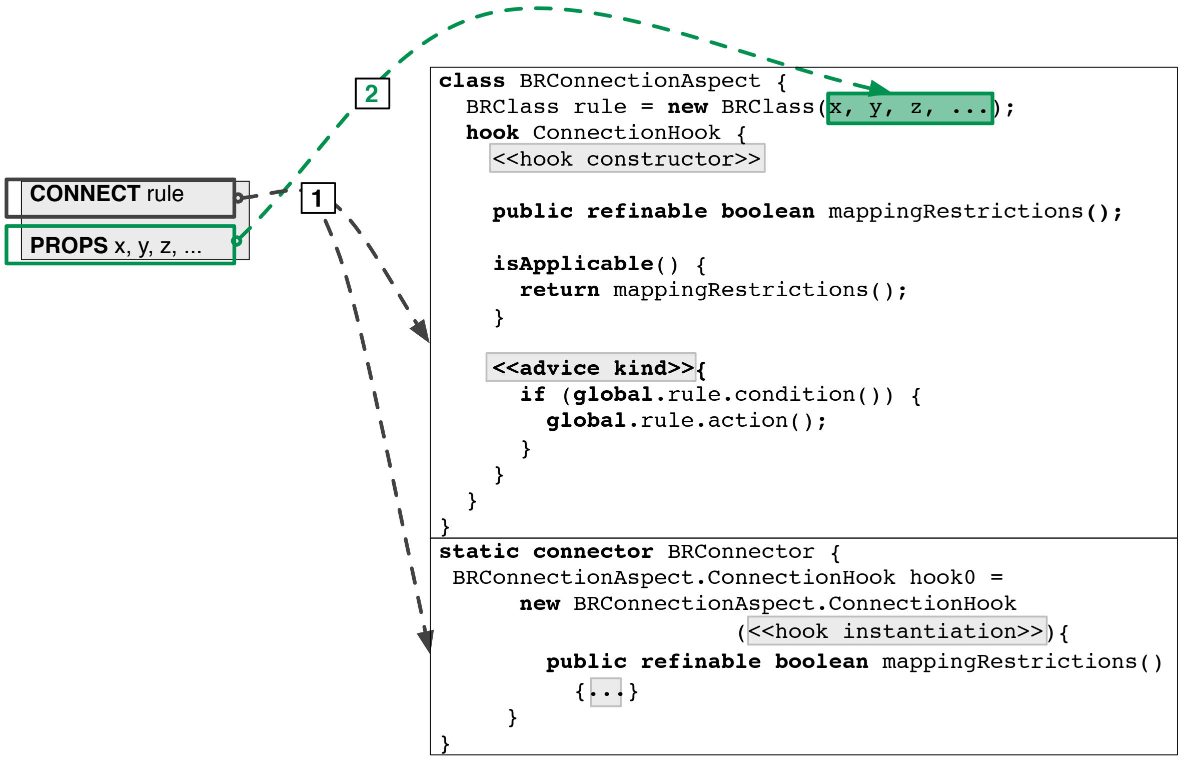
[3] Note that information about the mapping for each of the domain entities involved in the rule connection is required as well as an extra input, but it is omitted in the table for clarification reasons.



```
BR        BRDiscount
PROPS     Integer AS amount, Real AS discount
USING     ShoppingBasket AS basket
WHERE     targetCustomer IS basket.customer
IF        targetCustomer.account.amountSpent >= amount
THEN      basket.applyDiscount(discount)
```

```
public class BRDiscount {

  //PROPS
  int amount;
  float discount;

  //USING objects
  ecommerce.ShoppingBasket basket;

  //WHERE objects
  ecommerce.Customer targetcustomer;

  //RULE CONSTRUCTOR
  public BRDiscount(int amount, float discount) {
    this.amount = amount;
    this.discount = discount;
  }

  //GETTERS AND SETTERS for USING objects
  public ecommerce.ShoppingBasket getBasket() {
    return basket;
  }

  public void setBasket(ecommerce.ShoppingBasket basket) {
    this.basket = basket;
  }

  //INITIALIZE local attributes
  public void initializeRule()
  {
    this.targetcustomer = basket.getCustomer();
  }

  //RULE CONDITION
  public boolean condition() {
    return targetCustomer.getShopAccount().getTotalSpent() >= amount;
  }

  //RULE ACTION
  public void action() {
    basket.setDiscountRate(this.discount);
  }
}
```

*Figure 5.12:* Java class generated from the high-level `BRDiscount` rule



```
class BRConnectionAspect {
  BRClass rule = new BRClass(x, y, z, ...);
  hook ConnectionHook {
    <<hook constructor>>

    public refinable boolean mappingRestrictions();

    isApplicable() {
      return mappingRestrictions();
    }

    <<advice kind>>{
      if (global.rule.condition()) {
        global.rule.action();
      }
    }
  }
}
static connector BRConnector {
 BRConnectionAspect.ConnectionHook hook0 =
      new BRConnectionAspect.ConnectionHook
                (<<hook instantiation>>){
        public refinable boolean mappingRestrictions()
          {...}
      }
}
```

*Figure 5.13:* Transformations (1) and (2)

| ID | PIVOT | EXTRA INPUT | MAIN OUTPUT | EXTRA OUTPUT | KIND |
|---|---|---|---|---|---|
| 1 | *CONNECT <br>* | none | - aspect bean, connection hook and connector templates<br>- invocation of rule constructor<br>- advice body in charge of triggering rule application | none | L-2-G |
| 2 | *PROPS <value1>...<valueN>* | none | - rule constructor parameters | none | L-2-L |
| 3 | *BEFORE/AFTER/INSTEAD OF <event>* | none | - connection hook constructor<br>- connector deploying connection hook on concrete core application event<br>- restriction on the kind of advice that is needed in connection hook | none | L-2-G |
| 4 | *MAPPING* | information on whether the contextual data passed to the rule gets assigned | - invocation of getters and setters of rule attributes (declared in USING clause)<br>- restriction on the kind of advice that is needed in the connection hook | - position in the code of the connection hook where those getters and setters invocations need to be injected<br>- code for proceeding with core application taking into account rule results | G-2-G |
| 5 | *CAPTURE AT* | none | - hook capturing the required information<br>- aspect bean global variables keeping the captured information<br>- deployment of extra hook in connector | none | L-2-G |
| 6 | *a) ACTIVATE while <event>/ ACTIVATE not while <event>/ ACTIVATE while <event1> and not while <event2>/*<br>*b) ACTIVATE between <event1> and <event2>* | none | a) - pointcut involving cflow and/or !cflow pointcut designators<br>- extra deployment logic in connector<br>b) - stateful hook in the same aspect bean in charge of intercepting the application at two core events that mark the start and end of the activation period<br>- extra deployment logic in connector | none | L-2-G |

*Table 5.2:* Transformations from high-level rule connection constructs to AOP implementations

#### 5.6.2.1 Transforming CONNECT

Transformation (1) takes as input a *CONNECT* specification and produces (see Figure 5.13):

1. An aspect bean in charge of creating a new instance of the class implementing the rule that is being connected. This aspect bean defines a *hook* for the actual rule connection which defines an advice in charge of first checking the rule's condition by invoking the `condition()` method on the rule and second — in the case it is satisfied — triggering the rule's action if the rule's condition is satisfied. In addition, the signature of a refinable method *mappingRestrictions* is also generated. A refinable method in JAsCo is a method that has a dynamic body, i.e. the body is not fixed by the aspect bean but provided by the connector, and therefore fixed at aspect bean deployment time. Thus, a different body for the same method can be provided by different connectors. The aim of the *mappingRestrictions* method is to verify whether the restrictions imposed by the mapping of the connection event are satisfied or not. In our transformation process, the actual body of this refinable method is going to be produced by transformation (3) since it is only then when the information about the event and its restrictions are available (section 5.6.2.3). This method is invoked in the *isApplicable()* method of the generated aspect bean. As a result, only when the mapping restrictions are satisfied, the aspect bean proceeds with its execution, and therefore with the actual rule application.

2. The general schema of the connector in charge of deploying the connection hook is generated. However, note that the concrete details about how the hook constructor must be defined, which advice kind is needed and on which concrete method the hook needs to be deployed as well as the body of the *mappingRestrictions* method cannot be determined by transformation (1), as information about the rule application time is needed. As this information is embodied in other high-level features, different transformations will produce the corresponding outputs that need to be used to fill in the gaps.

#### 5.6.2.2 Transforming PROPS

In the case the instantiated rule is a rule template, transformation (2) is triggered which takes the values specified in the *PROPS* clause and produces rule constructor parameters as output. This result of this transformation and its combination with the output produced by transformation (1) is depicted in Figure 5.13. The keyword *global* is used in JAsCo to refer to members (i.e. variables or methods) defined globally in the aspect bean.

#### 5.6.2.3 Transforming BEFORE | AFTER | INSTEAD OF

Transformation (3) takes as input a rule application time specification — in the form of either *BEFORE* ⟨event⟩, *AFTER* ⟨event⟩ or *INSTEAD OF* ⟨event⟩ — and produces:

1. The constructor for the connection hook, which defines a pointcut capturing the execution of an abstract method parameter.

2. The concrete deployment code that needs to be written in the JAsCo connector in charge of deploying the connection hook on the corresponding concrete core application method that results from obtaining the mapping of the event. This is shown in Figure 5.14. Note that the output of this transformation does not depend on the concrete case of *BEFORE*, *AFTER* or *INSTEAD OF* connection.

An event captures the execution of a domain operation. In order to obtain the code representation for a given event, we need to analyze the mapping for the domain operation that defines that event. As it will be explained in detail in the chapter 6, a domain operation maps to an expression that involves navigations, arithmetical and logical operators. The code representation for a domain operation is then the signature of the most external method involved in its mapping expression[4]. For example, suppose *do(x, y)* is the domain operation involved in the connection event. Suppose *do(x, y)* maps to an expression of the kind: $targetObject.im_1(10, x.im2(), y)$. Then the code representation for this mapping is the signature of the OO method $im_1$. This signature is then used to instantiate the connection hook.

In addition, the body of the *mappingRestrictions* method is generated. This method checks whether the contextual information available in the context of the joinpoint coincides with the information specified in the mapping of the connection event. For instance, in the previous mapping example, the first parameter of $im1$ is fixed to the value 10 in the mapping of *do(x, y)*. Thus, the generated body of the *mappingRestrictions* method will look like this:

```
return thisJoinPoint.getArgumentsArray()[0].equals(10);
```

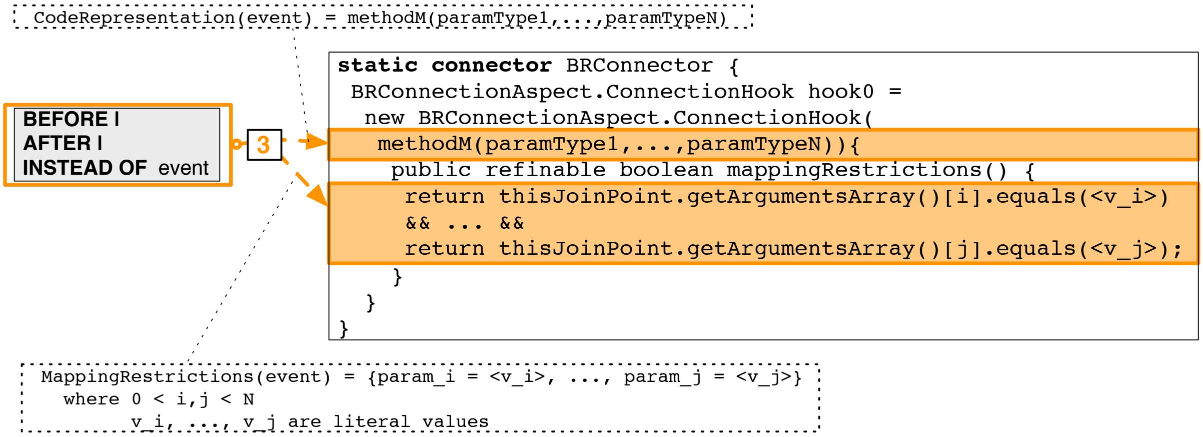
As a result, the generated connector ensures that the rule is only triggered when the method involved in the connection event is invoked with the parameters specified at mapping time. Note that the details of the mapping from domain entities to implementation are the subject of the coming chapter 6 and thus omitted here.

3. A filter on which are the possible kinds of advice needed in the connection hook: depending on the case of a *BEFORE*, *AFTER* or *INSTEAD OF* connection, a different kind of advice has to be generated. Moreover, determining the kind of advice also depends on whether the contextual information — passed to the rule using the *MAPPING* clause — is assigned to a new value in the rule. As explained in section 5.3, besides the invocation of domain operations, a rule action can also assign values to domain objects available in the local context of the rule by using the *IS* operator. When the assigned domain object corresponds to an object from the connection context (i.e. received as a parameter of the rule), the new assigned value has to be considered back in the connection context where the rule is triggered. A different kind of advice is needed depending on which kind of information gets assigned in the rule (i.e. target object, parameter or return value). This implies the existence of dependencies between transformations (3) and (4) which makes impossible the independent analyzes of their outputs. We analyze these dependencies per cases in the following section.

#### 5.6.2.4 Transforming MAPPING

This is a complex transformation because it requires not only the pivot but also extra information previously generated by other transformations. Also, besides generating multiple outputs, the places where these outputs need to be inserted as part of the hook connection

[4] A domain operation can also map to an attribute (as it will be explained in chapter 6), but for the definition of events only domain operations that map to methods can be used.



```
CodeRepresentation(event) = methodM(paramType1,...,paramTypeN)
```

```
static connector BRConnector {
 BRConnectionAspect.ConnectionHook hook0 =
  new BRConnectionAspect.ConnectionHook(
   methodM(paramType1,...,paramTypeN)){
    public refinable boolean mappingRestrictions() {
     return thisJoinPoint.getArgumentsArray()[i].equals(<v_i>)
     && ... &&
     return thisJoinPoint.getArgumentsArray()[j].equals(<v_j>);
    }
  }
}
```

```
MappingRestrictions(event) = {param_i = <v_i>, ..., param_j = <v_j>}
  where 0 < i,j < N
        v_i, ..., v_j are literal values
```

*Figure 5.14:* Partial output of transformation (3)

implementation vary depending on the case. Also, code for proceeding with the core application's execution needs to be generated as well depending on the case. These challenges are explained in the rest of the section.

- ***If the rule is connected* before *the connection event.*** Two cases are possible which are illustrated in Figure 5.15 (note that in the coming figures the *mappingRestrictions* and the *isApplicable* methods are omitted in the aspect bean implementation for simplicity reasons):

  - the information available in the context of the connection event is passed to the rule where it is assigned to a new value: we need to be able to access the contextual domain objects and make them available for the rule, trigger the rule's action where the domain objects are assigned to new values, and retrieve the modified information from the rule to be taken into account in the invocation of the original behavior captured by the connection event. Thus, an *around advice* is created for this purpose, since it allows intercepting the application at a certain point, adding some extra business logic and proceeding with the original execution, eventually considering a different target object and parameters. Figure 5.15 illustrates this situation (case 1.a depicts it for target object whereas case 2.a for parameters).
  - the contextual information is passed to the rule and *not* assigned by the rule: in this case, a *before advice* suffices to trigger the rule's action, as the original event execution does not need to be modified. Figure 5.15 illustrates this situation (case 1.b depicts it for target object whereas case 2.b for parameters).

  In the case of an event parameter that is passed to the rule, a corresponding parameter of the core method that results from obtaining the mapping of that event needs to be obtained. The following formula is applied (assume *Mapping* is a function that returns the implementation counterpart of a given domain element):

$$Mapping(param(i, event)) = Mapping(param(i, domainOperation(event))) =$$

$$= param(Mapping(i), Mapping(domainOperation(event)) = param(j, coreMethod)$$

where:

$$0 < i < \# \text{ parameters domain operation}$$

$$0 < j < \# \text{ parameters core method}$$

Thus, for a given event parameter $i$, a corresponding core parameter $j$ is obtained, where $i$ can be different to $j$. This parameter $j$ is then passed to the rule in the aspect code.

- ***If the rule is connected* after *an event*.** Two cases are possible (illustrated in Figure 5.16):

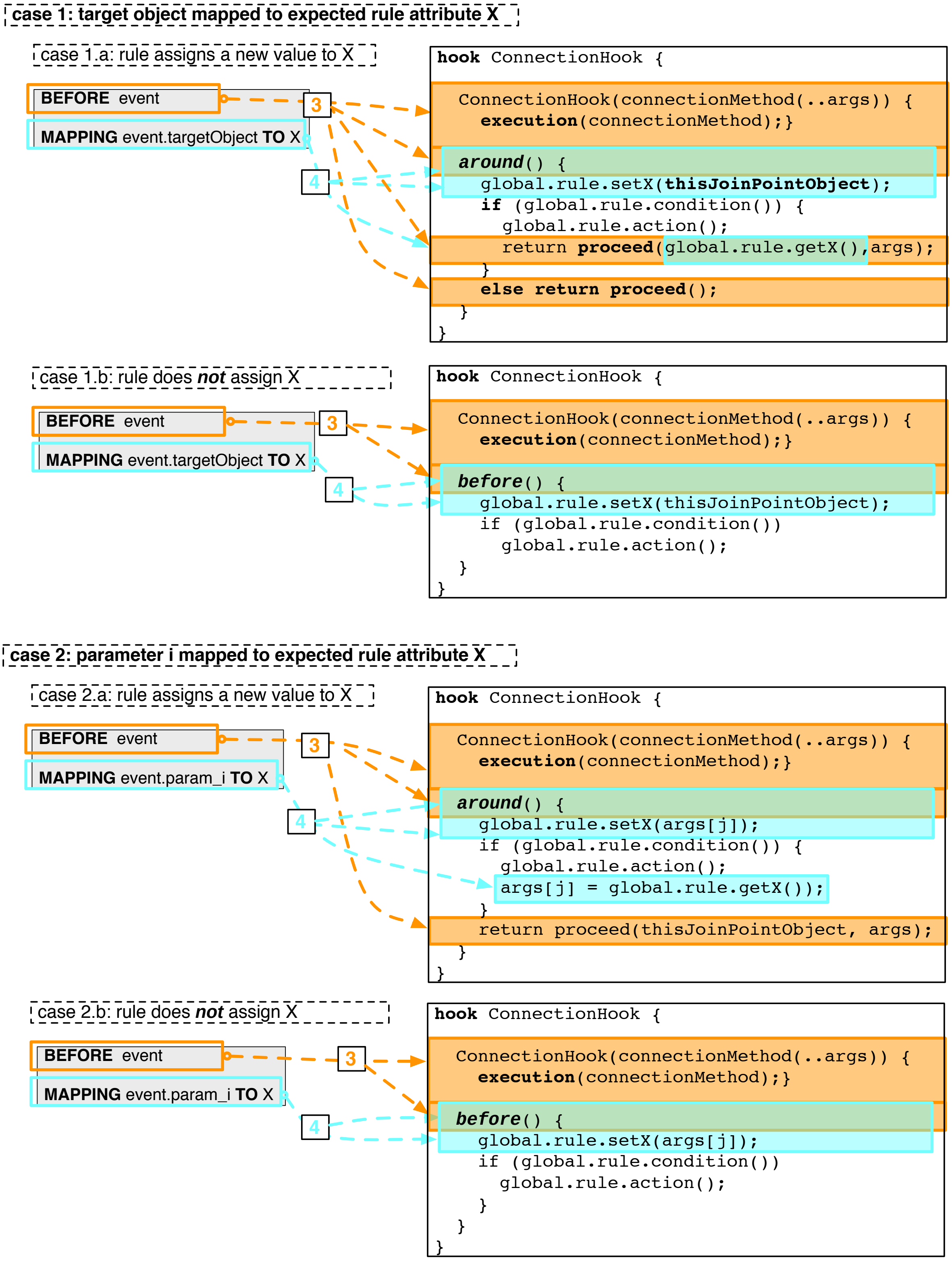
  1. the result of invoking the event is passed to the rule: in this case, an *around advice* is created which first invokes the original behavior captured by the event and passes the result of that execution to the rule. Two cases are possible regarding the return value of the around advice: (1.a) if in the rule the passed value is assigned a new value, then the around advice returns that new value; (1.b) otherwise, the original result is returned.
  2. no result is passed to the rule: an *after advice* suffices to trigger the rule's action, after the execution of the connection event.

- ***If a rule is connected* instead of *the execution of the connection event*.** Only one case is possible (illustrated in Figure 5.17):

  The original execution has to be replaced by the rule's action. This is achieved in an *around* advice which invokes the rule's action and does not proceed with the original execution.

#### 5.6.2.5 Transforming CAPTURE

Every capture point is translated into an additional hook, as illustrated in Figure 5.18. Depending on the information that needs to be captured, a different advice on that hook is generated: *before* for arguments and target object and *after returning* if return value is required. This implies dependencies between this transformation and the transformation of the *MAPPING* clause (transformation (4)) as the latter has the knowledge of which kind of information is needed from the capturing event and therefore which kind of advice is required in the capturing hook. The captured information is stored as variables in the aspect bean (shared among all the hooks of that aspect bean). This is a local-to-global transformation as several non-localized results are generated.


case 1: target object mapped to expected rule attribute X
case 1.a: rule assigns a new value to X
BEFORE event
MAPPING event.targetObject TO X
3
4
case 1.b: rule does *not* assign X
BEFORE event
MAPPING event.targetObject TO X
3
4
case 2: parameter i mapped to expected rule attribute X
case 2.a: rule assigns a new value to X
BEFORE event
MAPPING event.param_i TO X
3
4
case 2.b: rule does *not* assign X
BEFORE event
MAPPING event.param_i TO X
3
4

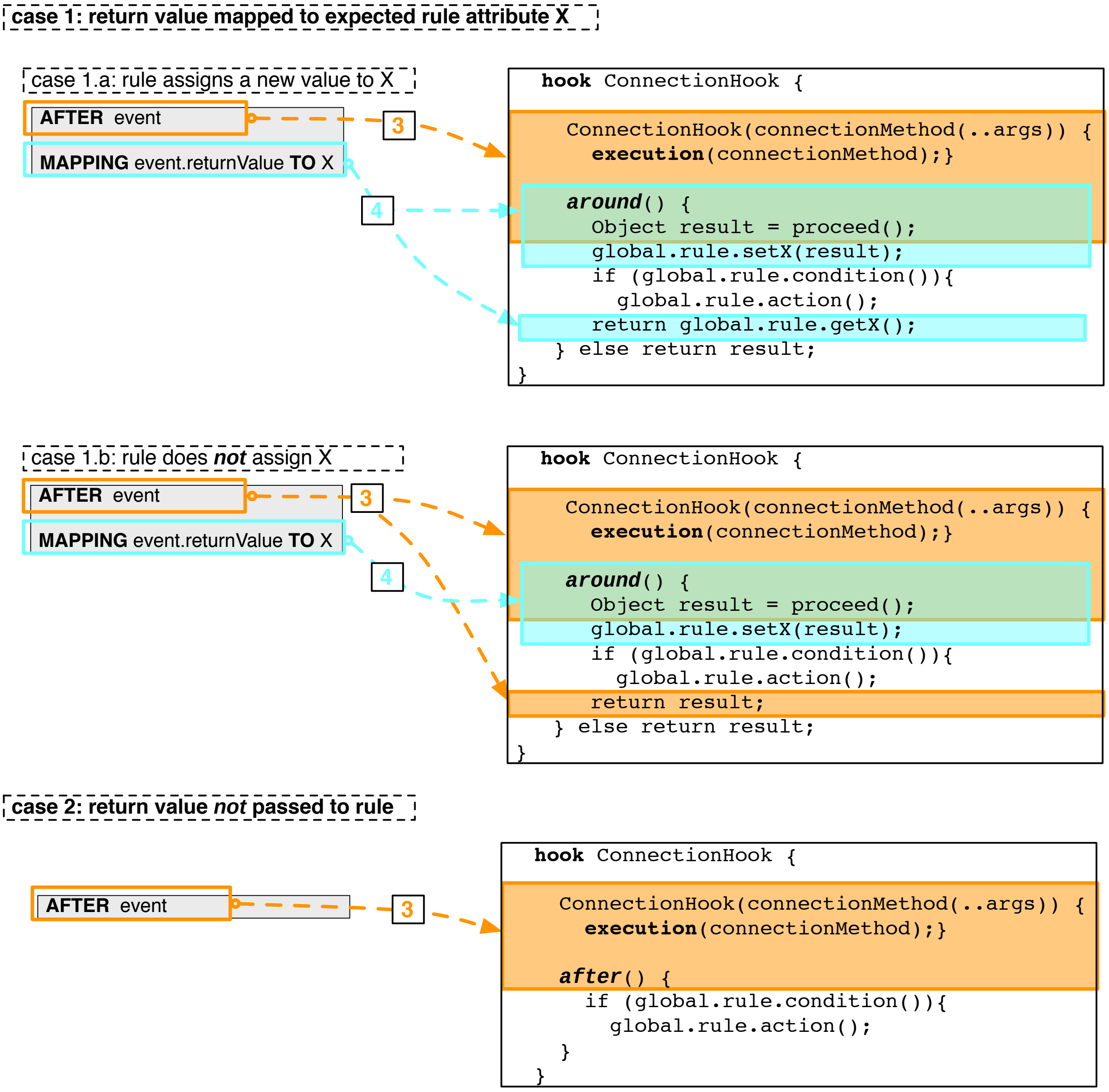
```
hook ConnectionHook {
  ConnectionHook(connectionMethod(..args)) {
    execution(connectionMethod);}
  around() {
    global.rule.setX(thisJoinPointObject);
    if (global.rule.condition()) {
      global.rule.action();
      return proceed(global.rule.getX(),args);
    }
    else return proceed();
  }
}
```

```
hook ConnectionHook {
  ConnectionHook(connectionMethod(..args)) {
    execution(connectionMethod);}
  before() {
    global.rule.setX(thisJoinPointObject);
    if (global.rule.condition())
      global.rule.action();
  }
}
```

```
hook ConnectionHook {
  ConnectionHook(connectionMethod(..args)) {
    execution(connectionMethod);}
  around() {
    global.rule.setX(args[j]);
    if (global.rule.condition()) {
      global.rule.action();
      args[j] = global.rule.getX());
    }
    return proceed(thisJoinPointObject, args);
  }
}
```

```
hook ConnectionHook {
  ConnectionHook(connectionMethod(..args)) {
    execution(connectionMethod);}
  before() {
    global.rule.setX(args[j]);
    if (global.rule.condition())
      global.rule.action();
    }
  }
}
```

*Figure 5.15:* Transformations (3) and (4): case of a ‘before’ connection

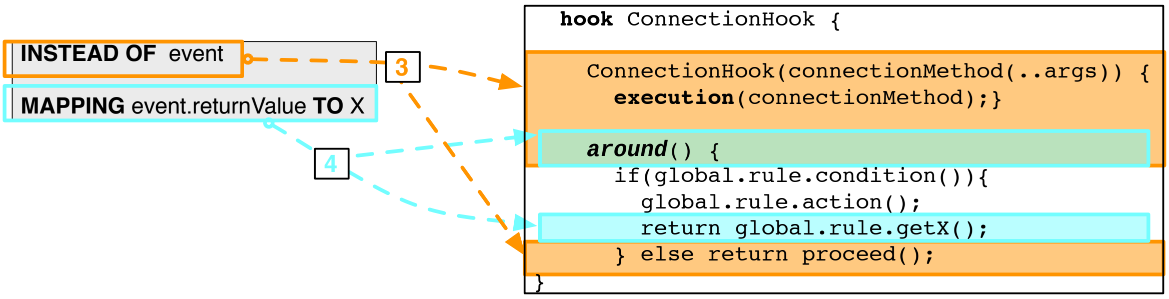

case 1: return value mapped to expected rule attribute X
case 1.a: rule assigns a new value to X
AFTER event
MAPPING event.returnValue TO X
3
4
case 1.b: rule does *not* assign X
AFTER event
MAPPING event.returnValue TO X
3
4
case 2: return value *not* passed to rule
AFTER event
3

```
hook ConnectionHook {
   ConnectionHook(connectionMethod(..args)) {
     execution(connectionMethod);}

   around() {
     Object result = proceed();
     global.rule.setX(result);
     if (global.rule.condition()){
       global.rule.action();
     return global.rule.getX();
  } else return result;
}
```

```
hook ConnectionHook {
   ConnectionHook(connectionMethod(..args)) {
     execution(connectionMethod);}

   around() {
     Object result = proceed();
     global.rule.setX(result);
     if (global.rule.condition()){
       global.rule.action();
     return result;
  } else return result;
}
```

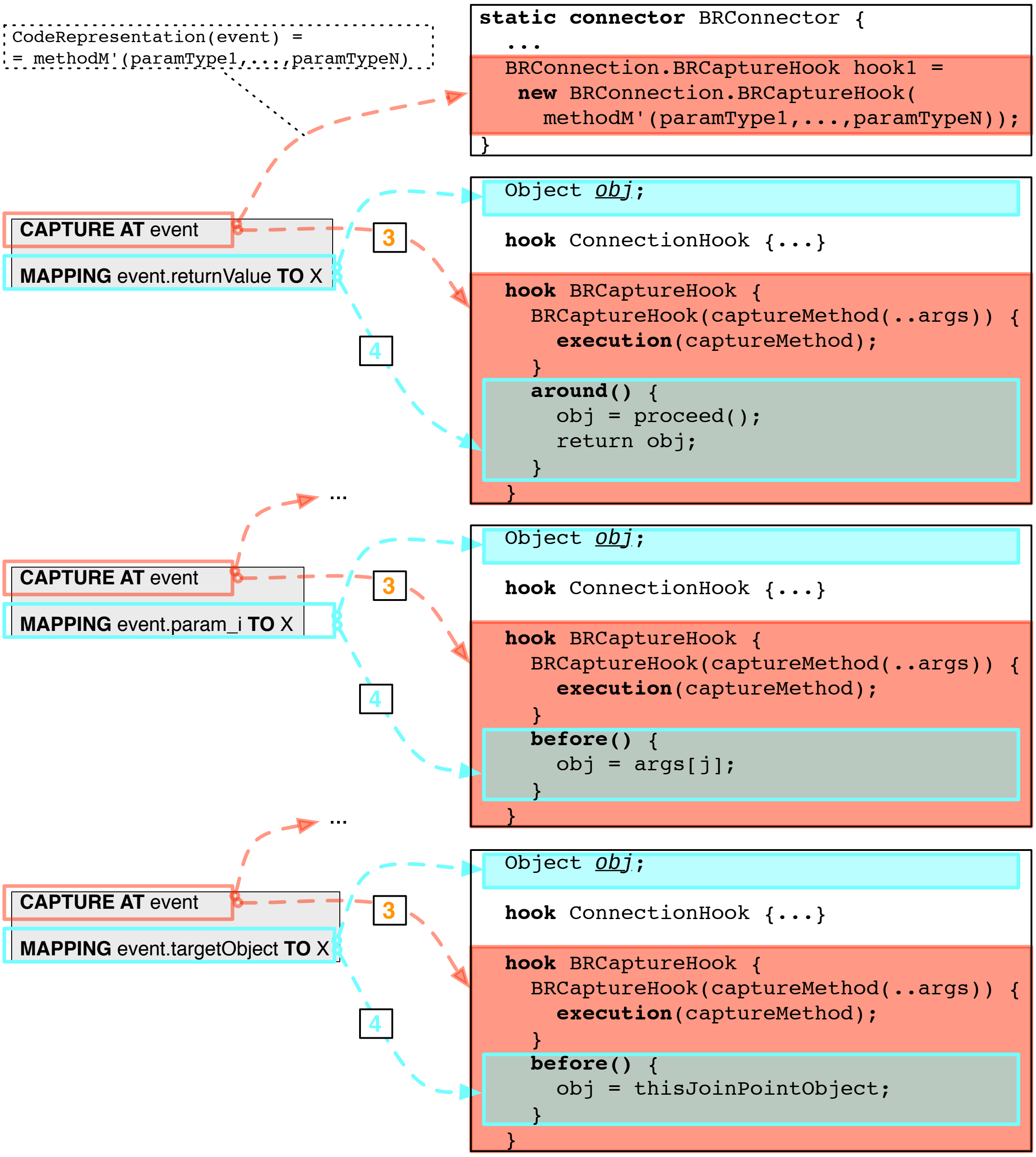
```
hook ConnectionHook {
   ConnectionHook(connectionMethod(..args)) {
     execution(connectionMethod);}

   after() {
     if (global.rule.condition()){
       global.rule.action();
   }
 }
```

*Figure 5.16:* Transformations (3) and (4): case of an ‘after’ connection



```
hook ConnectionHook {
   ConnectionHook(connectionMethod(..args)) {
     execution(connectionMethod);}

   around() {
     if(global.rule.condition()){
       global.rule.action();
       return global.rule.getX();
     } else return proceed();
}
```

*Figure 5.17:* Transformation (3) and (4): case of an ‘instead of’ connection



CodeRepresentation(event) =
= methodM'(paramType1,...,paramTypeN)
CAPTURE AT event
MAPPING event.returnValue TO X
3
4
...
CAPTURE AT event
MAPPING event.param_i TO X
3
4
...
CAPTURE AT event
MAPPING event.targetObject TO X
3
4

```
static connector BRConnector {
  ...
  BRConnection.BRCaptureHook hook1 =
   new BRConnection.BRCaptureHook(
     methodM'(paramType1,...,paramTypeN));
}
```

```
  Object obj;

  hook ConnectionHook {...}

  hook BRCaptureHook {
    BRCaptureHook(captureMethod(..args)) {
      execution(captureMethod);
    }
    around() {
      obj = proceed();
      return obj;
    }
  }
```

```
  Object obj;

  hook ConnectionHook {...}

  hook BRCaptureHook {
    BRCaptureHook(captureMethod(..args)) {
      execution(captureMethod);
    }
    before() {
      obj = args[j];
    }
  }
```

```
  Object obj;

  hook ConnectionHook {...}

  hook BRCaptureHook {
    BRCaptureHook(captureMethod(..args)) {
      execution(captureMethod);
    }
    before() {
      obj = thisJoinPointObject;
    }
  }
```

*Figure 5.18:* Transformations (4) and (5)

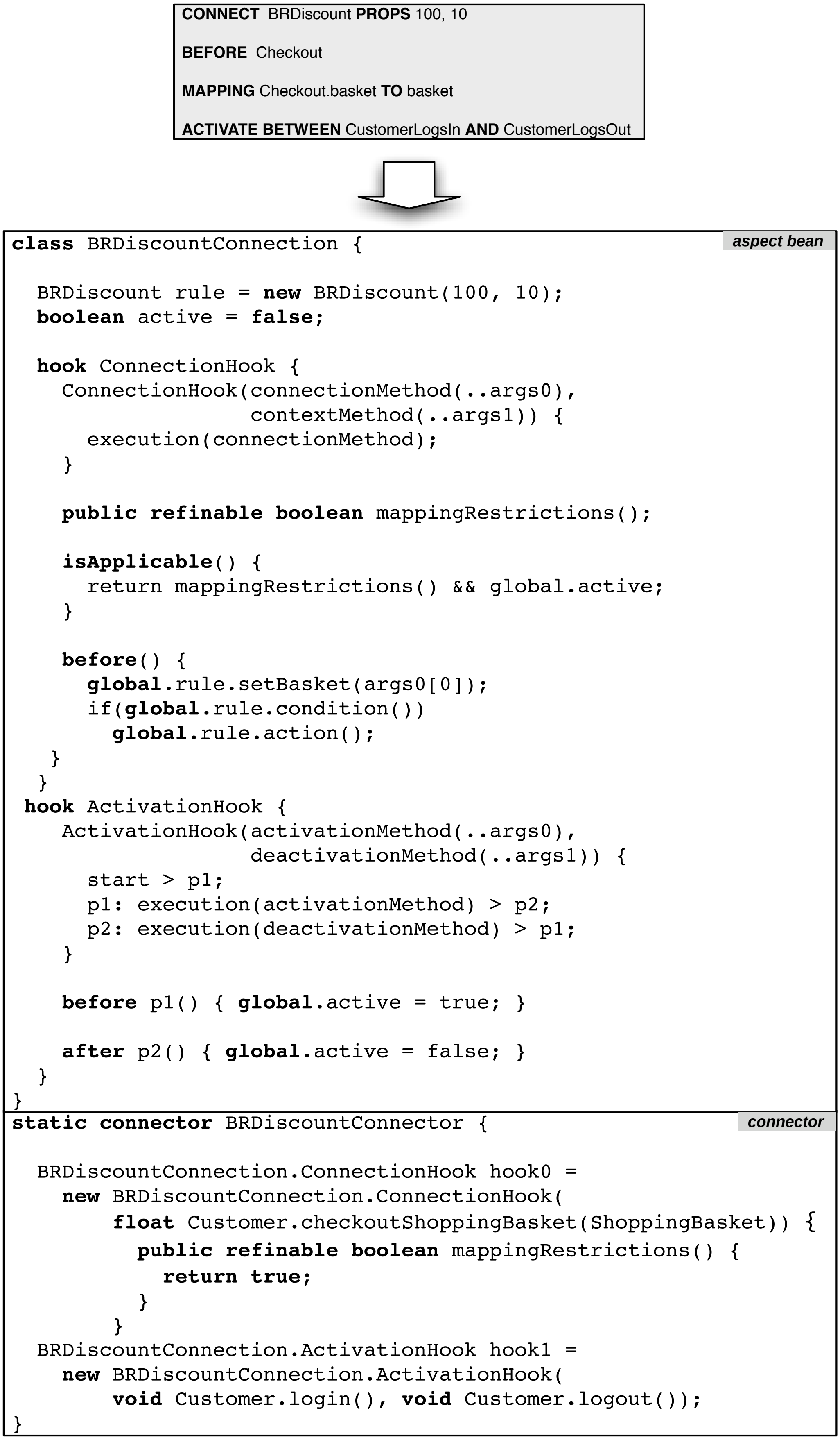
#### 5.6.2.6 Transforming ACTIVATION

The activation time is translated to the exact patterns shown in chapter 4: the constructs *ACTIVATE WHILE* ⟨event1⟩, *ACTIVATE NOT WHILE* ⟨event1⟩, *ACTIVATE WHILE* ⟨event1⟩ *NOT WHILE* ⟨event2⟩ and *ACTIVATE BETWEEN* ⟨event1⟩ *AND* ⟨event2⟩ translate to the code shown in Figures 4.5, 4.6, 4.7 and 4.8 respectively. These translations are examples of local-to-global transformations.

Figure 5.19 shows the result of translating the connection of the *BRDiscount* rule (introduced in section 5.4).

## 5.7 Summary

In this chapter we presented a domain model for expressing domain concepts explicitly and defining business rules and their connections with the core application in terms of those explicit domain concepts. The proposed business rule language is simpler than a fully-fledged programming language and thus facilitates the task of writing the rules. Note however that rules can be very powerful because they do not simply involve aliases to implementation entities but also entities that can have a very complex realization at the level of the implementation. Contrary to current approaches where the rules themselves can refer to complex OO constructs, in our approach the rules are simple since all the complexity is taken away from their specification and encapsulated in the mapping of the involved domain entities (explained in detail in chapter 6). The high-level rule connection language provides features that are abstracted from the recurrent connection elements distilled in the aspect patterns presented in chapter 4. We also presented transformations from both high-level rules and high-level connections to rule objects and connection aspects respectively. These transformations pose challenges in the transformation process as they correspond to the categories local-to-global and global-to-global for which extra non-localized inputs need to be gathered and/or outputs of different transformations need to be combined in a non-trivial way. The proposed transformations were discussed and illustrated taking into account the dependencies that exist among them.



```
CONNECT  BRDiscount PROPS 100, 10

BEFORE  Checkout

MAPPING Checkout.basket TO basket

ACTIVATE BETWEEN CustomerLogsIn AND CustomerLogsOut
```

```
class BRDiscountConnection {

  BRDiscount rule = new BRDiscount(100, 10);
  boolean active = false;

  hook ConnectionHook {
    ConnectionHook(connectionMethod(..args0),
                   contextMethod(..args1)) {
      execution(connectionMethod);
    }

    public refinable boolean mappingRestrictions();

    isApplicable() {
      return mappingRestrictions() && global.active;
    }

    before() {
      global.rule.setBasket(args0[0]);
      if(global.rule.condition())
        global.rule.action();
   }
  }
 hook ActivationHook {
    ActivationHook(activationMethod(..args0),
                   deactivationMethod(..args1)) {
      start > p1;
      p1: execution(activationMethod) > p2;
      p2: execution(deactivationMethod) > p1;
    }

    before p1() { global.active = true; }

    after p2() { global.active = false; }
  }
}
```

```
static connector BRDiscountConnector {

  BRDiscountConnection.ConnectionHook hook0 =
    new BRDiscountConnection.ConnectionHook(
        float Customer.checkoutShoppingBasket(ShoppingBasket)) {
          public refinable boolean mappingRestrictions() {
            return true;
          }
        }
  BRDiscountConnection.ActivationHook hook1 =
    new BRDiscountConnection.ActivationHook(
        void Customer.login(), void Customer.logout());
}
```

*Figure 5.19:* Transformation from the high-level connection of `BRDiscount` to JAsCo

# Chapter 6

# Mapping Domain Knowledge To Implementation

In chapter 5 we described how high-level rules are defined in terms of high-level domain entities of a domain model. In this chapter, the mapping from high-level domain entities to implementation is made explicit. Although other approaches exist today which advocate this idea, they only support simple one-to-one mappings. We build on this current support and enhance it in many directions. In this chapter we first set up the context and motivate the need for having more complex mappings (section 6.1). We then present the main features of the proposed and implemented mapping language (section 6.2). Finally we show how the proposed mapping language can be used to realize five different mapping use cases (sections 6.3 to 6.7).

## 6.1 Context: Advanced Domain Mappings

An essential step for achieving the automatic generation of executable rules from high-level specifications is the definition of how the domain entities involved in those specifications are mapped to the implementation. In this chapter we present an approach for making the *mapping* between the domain and the implementation models explicit.

Empirical studies have shown that the explicit description of domain knowledge is the most essential information needed by software maintainers. Moreover, they also identify the importance of linking this explicit domain knowledge to the corresponding implementation entities. This introduces the idea of *domain concept location*, which refers to the place in the code where a certain domain concept is implemented. Koskinen et al. [KSP04] observed that domain concept descriptions and their link to implementation knowledge are among the three most frequent types of information needed by software maintainers. This link is most useful for corrective and adaptive maintenance. Bennet et al. [BR00] confirm that it is crucial for developers to understand how domain concepts relate to the code.

Other business rule approaches exist today which advocate the idea of expressing high-level rules in terms of domain concepts that map to an implementation. Examples of such approaches are JRules, QuickRules, VisualRules, JBoss Rules and HaleyRules (explained in detail in chapter 9). However, in these approaches the domain concepts are simple aliases for implementation entities and thus a one-to-one mapping between them is assumed. When domain concepts have more complex realizations at the implementation level, the need for supporting more complex mappings than the one-to-one mappings arises. We can imagine, for instance, the need for expressing mappings to expressions that can involve many implementation entities which need to be obtained through navigations in the implementation. Theoretically, any OO expression could be written in order to describe the mapping. Thus, when pursuing the endeavor of defining a more powerful mapping language, the issues to be faced are: how to create a systematic approach to the definition of complex domain mappings? Which expressions should be supported by the mapping language? Where to draw the line of expressiveness? We have analyzed these issues and distinguished relevant cases for which the need for certain mapping functionalities is observed. As a result of this analysis, we designed a simple but powerful mapping language, which is explained in the next section. The subsequent sections present the relevant cases we identified and analyzed, illustrating how the mapping language is used to support them. In some cases, we show extensions in the form of syntactic sugar and extra verification mechanisms that make our basic mapping language features more usable.

The presented mapping language allows expressing mappings in terms of more than one entity in the implementation. These entities can be combined in complex expressions which can in turn involve nested navigations and literal values. Moreover, the proposed mapping language enhances the current domain mapping support found in existing approaches in three innovative directions:

- domain entities can map to more than one implementation entity: typically this mapping links one domain class to many OO classes in the existing implementation. Although this mapping is expressed at the structural level in terms of domain classes, operations and properties on the one hand and OO classes, methods and attributes on the other hand, in order for the mapping to be usable, the actual instances of those classes that are involved in the mapping need to be obtained. This is a real challenge in the case where the correspondence between the many instances of the OO classes involved in the mapping cannot be determined statically. For establishing the instance correspondence in this case, we propose the use of AOP.
- domain entities can explicitly represent derived information: the derivation can be expressed at two levels, low and high levels. In the former case, the derivation expression is defined in terms of implementation entities whereas in the latter case, only domain entities of a domain model are involved. Moreover, at the implementation level, calculating or obtaining derived information can result in crosscutting code, in which case the use of AOP is proposed. Defining new domain entities in terms of other existing domain entities allows for domain evolution.
- domain entities can be completely unanticipated in the existing implementation. The realization of these domain entities at the implementation level requires the addition of new implementation entities. We pursue this addition to be done in a non-invasive way. Thus again, the use of AOP is proposed.

In all these cases, AOP is used in a transparent way for the user of the mapping language.

## 6.2 The Mapping Language

In this section, the basic capabilities of the proposed mapping language are presented. A prototype of this language has been implemented, the details of which are described in chapter 7. The complete grammar of the implemented prototype is in Appendix C.

We consider the case where the target application is developed in Java. We restrict the possible Java entities to the following ones: classes (concrete or abstract), methods (static or not), attributes (static or not) and interfaces. We consider that OO classes can be related in hierarchies and that single inheritance is supported. Figure 6.1 depicts these decisions with respect to what elements and relations are possible in both models, the domain and the implementation models.

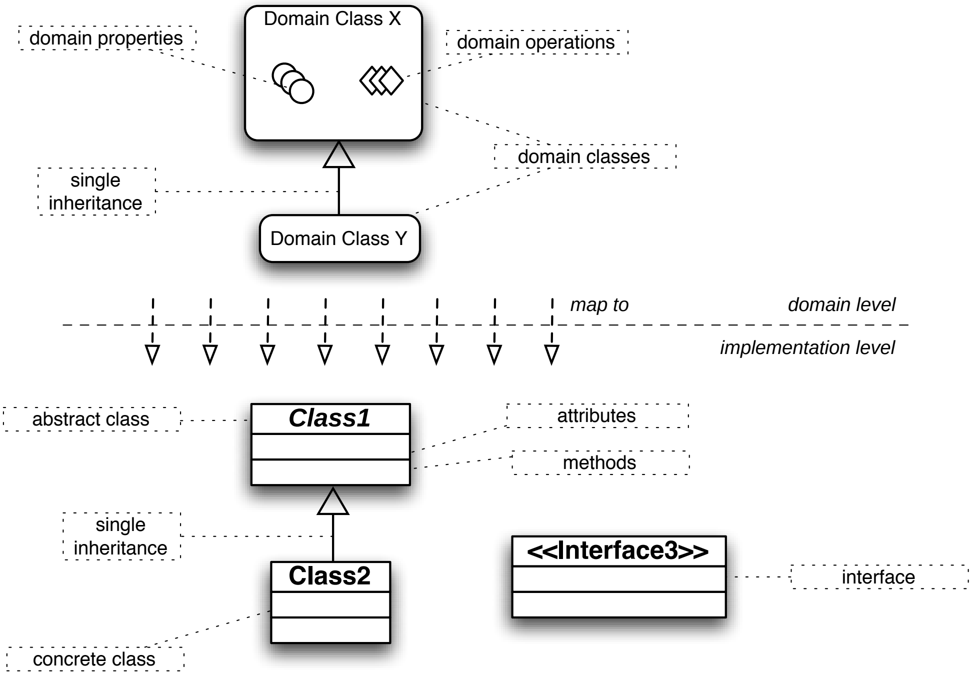


*Figure 6.1:* Entities and relations considered in both the domain and implementation levels

### 6.2.1 Basic Mappings

The basic construct in our mapping language has the form:

$$< LHS >< map - to >< RHS >$$

The $< LHS >$ defines any of the domain entities supported in the model, i.e. a domain class, domain property or domain operation. The $< map - to >$ has different flavors. The basic mapping operator considered here is the operator to map to implementation, called *MAP-TO-IMPL*. When this concrete operator is used, the $< RHS >$ is completely defined in terms of implementation entities. We consider several cases which are described below. For all these cases, we decided to use the same mapping operator, i.e. *MAP-TO-IMPL*, because of the uniformity of this approach. However, depending on how this operator is used, different semantics are associated to it. This can be regarded as some sort of operator overloading.

First, when the $< LHS >$ is the name of a domain class, the $< RHS >$ has to be the name of a Java class or interface. An example definition of a mapping for a domain class is shown below. In this example and the coming ones the following notation is used: $DomainClass_1$, ..., $DomainClass_n$ are names of domain classes whereas $Class_1$, ..., $Class_n$ are names of Java classes or interfaces.

```
DomainClass1 MAP-TO-IMPL Class1
```

In order to define domain operations and properties — referred to as *contained domain entities* — for a given domain class, the name of the domain class has to be followed by braces that enclose the mapping definition for those contained domain entities. Note that this definition does not necessarily need to come after the mapping definition for the domain class itself.

```
DomainClass1 {
  ...//mapping definition for domain operations and/or domain attributes
}
```

Second, in the case when the $< LHS >$ is a domain operation or property, the $< RHS >$ is any expression in the target OO language, Java in our case[1] [2]. The value for the domain entity in the $< LHS >$ corresponds to the value that results from evaluating the expression. A relationship exists between the $< LHS >$ and the $< RHS >$ expression, which is that all the variable names occurring in the $< RHS >$ should also occur in the $< LHS >$. Variables on the $< LHS >$ are defined for referring to the target entity, in the case of a domain operation and property, and parameters, in the case of domain operation. The simplest case for the mapping of a contained domain entity, is when there is a one-to-one match between the contained domain entity that is being mapped and the implementation entity defined at the $< RHS >$. An example is shown below. For the coming examples assume the following notation: $do_1$, ..., $do_n$ and $dp_1$, ..., $dp_m$ are names of domain operations and domain properties respectively whereas $im_1$, ..., $im_n$ and $ia_1$, ..., $ia_m$ are names of implementation methods and attributes respectively. The target variable name, $e$ in the example, is interpreted as follows: when included in the $< LHS >$ it is an instance of the domain class where the mapping is being defined whereas when included in the $< RHS >$ it is an instance of the type to which the domain class maps (the instance correspondence relation is explained in detail in section 6.4). Note that in the case where a domain class maps to a Java interface, the mappings for its contained domain entities define expressions at the $< RHS >$ that can only use $e$ as a target of a method invocation and not of an attribute retrieval.

```
DomainClass1{
 e.dp1 MAP-TO-IMPL e.ia1
 e.do1(x, y) MAP-TO-IMPL e.im1(int:x, string:y)
}
```

[1] In Java the distinction is made between statement and expression. Conditional and iteration statements are not supported in the $< RHS >$ of a mapping definition.

[2] Note that in this case, the mapping has a more behavioral flavor. On the contrary, the case of a mapping for a domain class is more structural, as a structural relation exists between an element of the domain and an element of the implementation. Even in the implementation of the operator, this observation is apparent: the name of a class is stored for a mapping of a domain class, and not an expression that needs to be evaluated.

Note that in this example mapping definition, besides a difference in names that might exist between the domain entities in the $< LHS >$ and the corresponding methods and attributes in the $< RHS >$ of the mappings in lines 2 and 3, a *one-to-one* match exists between them: one implementation entity exists which exactly implements that one domain concept; in addition, in the case of a domain operation (line 3), the parameters perfectly correspond in terms of number and order. It is clear that in a similar manner more indirect mappings can be specified, for example:

```
1 DomainClass1{
2   e.do1(x,y) MAP-TO-IMPL e.im1(string:y, int:x)
3   e.do2(y) MAP-TO-IMPL e.im2(int:10, string:y)
4 }
```

The definition in line 2 exemplifies that a different order can be specified for the parameters of the domain operation on the $< LHS >$ and the corresponding method on the $< RHS >$, whereas line 3 shows how literals can be used to fix the value of method parameters in the $< RHS >$. Moreover, as mentioned earlier, the variables in the $< LHS >$ can be used in any place at the $< RHS >$. For example, the target entity in the $< LHS >$ can be included either as a target or as a parameter of any method invocation in the $< RHS >$, as shown below. In any case, the navigations are defined in accordance to the types of the involved implementation entities, as specified in the existing application.

```
1 DomainClass1{
2   e.do1(x) MAP-TO-IMPL e.ia1.im1(string:x)
3   e.do2(x,y) MAP-TO-IMPL e.ia1.im2(y.im3(float:5*x, y.im4() < 10))
4   e.dp1 MAP-TO-IMPL e.im5(int:10, string:"hello")
5 }
```

This listing illustrates that it is possible to map a domain property to an implementation method, as shown in line 4. Also, more complex combinations of the basic cases are possible, such as nested navigations involved in comparisons, arithmetical and logical operations, etc. as illustrated in line 3 of the same mapping specification.

The invocation to static methods or the retrieval of static attributes is also supported in the expressions at the $< RHS >$, as shown below. Note that the class to which the domain class *DomainClass1* maps can be referred to in the $< RHS >$ (as shown in line 2) as well as any other OO class or interface available in the core application (as shown in line 3).

```
1 DomainClass1{
2   e.do1(x) MAP-TO-IMPL Class1.im1(string:x)
3   e.do2() MAP-TO-IMPL Class2.ia1
4 }
```

Note that the $< LHS >$ is not explicitly typed. The mappings determine the types of the domain model. When the $< RHS >$ is an expression, information about types of method parameters need to be explicitly specified. An exception to this is the case when an expression is used as a method parameter, in which case the type for that parameter is determined at the time the mapping specification is translated into an object representation for that mapping. This translation process is explained in detail the next chapter 7 (section 7.6).

### 6.2.2 High-Level Mappings

Our approach supports mappings at different levels of abstraction. The previous mapping operator *MAP-TO-IMPL* only targeted implementation elements in the $< RHS >$ and therefore is used to define low-level mappings. However, once there are some domain entities defined, the specification of new domain entities in terms of the existing ones is possible. In this section we introduce another flavor for the $< map - to >$ operator: the “high-level” mapping operator *MAP-TO-DOMAIN* which allows the definition of high-level mappings that are completely specified at the domain level. These high-level mappings require pointing to existing domain entities of a domain model. Thus, the initial low-level mappings can be seen as a bootstrap of the domain model on top of which higher-level abstractions can be defined.

In a high-level mapping, the $< RHS >$ is defined analogously to the $< RHS >$ of the *MAP-TO-IMPL* case, with the difference that the expression now only involves domain entities. These domain-level expressions are defined in terms of several predefined domain operators: navigation, arithmetical and logical operators. Furthermore, literals and variables are allowed in these domain expressions. The use of the operator is illustrated in the examples shown below. In the case of mappings to domain, the target variable name ($e$ in the examples) is interpreted at both sides as an instance of the domain class where the mapping is being defined.

```
1  DomainClass1 {
2    e.dp1 MAP-TO-IMPL ...
3    e.do1(x, y) MAP-TO-IMPL ...
4
5    e.do2() MAP-TO-DOMAIN e.do1(10, "hello")
6    e.do3(y, x) MAP-TO-DOMAIN e.do1(x, y)
7    e.do4(z) MAP-TO-DOMAIN e.do3(z, e.do2()) + e.dp1
8
9    e.dp2 MAP-TO-DOMAIN e.do1(20, "bye")
10   e.dp3 MAP-TO-DOMAIN e.dp1 < 10
11 }
```

High-level mappings are added to the domain class definition, between the braces, potentially mixed with definitions in terms of the other operator *MAP-TO-IMPL*, as illustrated in lines 2 and 3 of the previous mapping specification.

The domain model also supports single inheritance between domain classes. This is defined as shown below. As a result, all domain operations and properties defined in the super domain class are inherited by the sub-domain class.

```
DomainClass1 INHERITS-FROM DomainClass2
```

### 6.2.3 Special Mapping Operators

We can imagine the situation where defining a mapping for a domain entity implies grabbing dynamic information that is only available at well-defined points in the execution of the core application. Moreover, this information cannot be obtained by simply invoking a method, retrieving an attribute, navigating the implementation model or evaluating a simple arithmetical or logical expression at the $< RHS >$ of a mapping specification. This is because this information crosscuts the core application, and therefore grabbing it would result in tangled and scattered code. In order to grab this dynamic information in a non-invasive way, we could imagine extending the mapping language to allow writing AOP expressions in the $< RHS >$ of a mapping specification. However, unleashing the full power of AOP introduces two fundamental issues. The first issue is again how to constrain the power that comes with AOP so that it is usable? Secondly, how to express a mapping that relies on AOP in a mapping language? So far, in the case of for instance a domain entity mapping to a method, the value for the domain entity is the value returned by the invocation of that method. However, in the case of mappings based on AOP, the idea of "calling" or "invoking" some functionality defined in an aspect does not apply. Because dependency inversion is ensured, aspects are not "called" but triggered at the occurrence of joinpoints. Thus, it is clear that mappings to AOP need to be treated differently than the mappings presented so far.

In our approach we consider the case where the well-defined points of interest where to grab dynamic information correspond to method executions. In this context, two kinds of information is available. First of all, we can capture the time when the method is executed. In this context different variations are possible, we can talk about the exact time at which a method is executed, or a relative time, such as whether a method is executed before of after another method, the time that elapsed between two method executions, etc. Secondly, we can capture the values that are involved in that method execution, e.g. values that are passed as parameters, or the value that is returned by that method execution, etc. Inspired in this kind of examples, our approach proposes general-purpose mapping operators that can deal with this kind of dynamic information and which are implemented using AOP technology. As a result of using these special operators in mapping specifications, aspects are automatically installed in the system which keep track of the required dynamic information and make it available when needed. Again, the actual rationale behind these AOP operators becomes clearer in the subsequent sections where we illustrate their uses with examples (sections 6.6 and 6.7).

Two kinds of AOP mappings are supported, which correspond to the two well-known AOP flavors, *dynamic* and *static*. The dynamic kind depends on dynamic AOP: aspects, join points and advices are used for the implementation of these operators. These aspects are predefined in the domain model infrastructure. Connectors for the deployment of these aspects are automatically generated by our prototype implementation which use the information specified at mapping time. Two example operators in this category are supported, the *timeBetween* and *previousResult* operators, which can be used at the $< RHS >$ of a mapping definition, as shown below in lines 5 and 6. These operators are generic and therefore not tight to the particularities of a specific domain.

```
1 DomainClass1 MAP-TO-IMPL Class1
2 DomainClass1 {
3  e.do1(p1, p2, ..., pn) MAP-TO-...
4  e.do2(t1, t2, ..., tn) MAP-TO-...
5  e.dp MAP-TO-DOMAIN timeBetween(e.do1(p1, p2, ..., pn), e.do2(t1, t2, ..., tn))
6  e.dp1 MAP-TO-DOMAIN previousResult(e.do1(p1, p2, ..., pn))
7 }
```

The *timeBetween* operator allows measuring the time elapsed between the invocations of the two domain operations received as parameters whereas the *previousResult* operator allows for keeping track of the value that results from the invocation to a domain operation to be used at a later point in time. Note that writing `e.do1(p1, p2, ..., pn)` in the $< RHS >$ of line 6 is not equivalent, since doing so would imply actually invoking `do1` every time the value of `dp1` has to be obtained. Instead, the use of the *previousResult* operator indicates that the execution of `do1` has to be monitored in order for its last result to be kept, but no invocation is actually carried out as a result of this mapping.

The main difference between the ordinary mappings presented earlier (low or high-level) and the mappings in terms of AOP-based domain operators is that in the latter, an extra step is carried out for setting up the aspects in charge of calculating the values to be associated to the domain entities for which the mapping is being defined. This implies that the value for the domain entities for which the mapping is defined in terms of these special operators is only made available when the domain operations involved in those mappings (more specifically in the deployment of those operators) have been invoked at least once.

Special operators can be involved in more complex expressions (i.e. arithmetical, logical and navigational). The details of the AOP implementation for these operators can be found in chapter 7. It is important to note that it is not the aim of this dissertation to provide an exhaustive categorization of operators but to show how such operators can be implemented using AOP and how they can be used in mapping expressions. Following the same idea, other domain operators that were not foreseen in the proposed library could be added. Moreover, we can envision using the same mechanism presented here for the definition of domain-specific operators, i.e. operators which are specifically designed for a particular domain.

Finally, a last mapping operator is defined: *MAP-TO-VALUE*. This operator can be used for the definition of domain properties for which their mapping cannot be defined in terms of an expression, neither at the implementation nor at the domain level. It states that a domain property holds a value defined by a certain domain class. An initial value can be assigned to it which might be changed later on by business rules. This becomes clearer in section 6.7 where we demonstrate the use of this operator in the e-commerce domain. This operator is used as follows:

```
DomainClass1 MAP-TO-IMPL Class1
DomainClass1 {
 e.dp1 MAP-TO-VALUE DomainClass2:<<initialValue>>
}
```

The implementation of this operator is based on static AOP. At mapping translation time, a new aspect is automatically generated and installed in the system which introduces a new attribute for the realization of the *dp1* domain property and methods for its manipulation. The attribute and methods are added, in a non-invasive way, to the class *Class1*. The type of the introduced attribute is defined by the mapping of the domain class `DomainClass2` (referred to in line 3). The initial value `<<initialValue>>` (specified in line 3) is set to the introduced attribute. Again, the details of the aspect generation and deployment process for this operator is included in chapter 7.

### 6.2.4 Mapping Events

The mapping language also allows defining domain events in terms of domain operations in the domain model. The syntax for this is:

```
1  EVENT event1 AT DomainClass1.do1()
2  EXPOSING  TARGET AS target
3          RETURN VALUE AS return
4
5  EVENT event2 AT DomainClass1.do2(p0, ..., pn)
6  EXPOSING  TARGET AS target
7          RETURN VALUE AS return
8          PARAMETER 0 AS par0
9          ...
10         PARAMETER N AS parN
```

In line 1 of this mapping specification, an event with name *event1* is defined which captures the execution of the domain operation *do1* defined in the domain class *DomainClass1*. A second event *event2* is defined in an analogous way in line 5. Also, these examples show that events can expose the available contextual information including target object, parameters and return value of the involved domain operations. In the case of the event *event1*, the contextual target object and the return value are exposed with names *target* and *return* respectively.

## 6.3 Use Case 1: Pulling Up a Class

This case represents a bottom-up scenario where a complete OO class is to be pulled up at the domain level[3]. This can be done by explicitly mapping all the attributes and methods defined in that OO class one by one. However, this is a tedious task. Thus, a new mapping operator *ALIAS-FOR* is provided in our mapping language to be used as a shortcut for the definition of these mappings, as shown below. Note that additional contained domain entities can be added to the domain class definition by means of including mapping definitions for them in between the braces that follow the domain class name.

```
DomainClass1 ALIAS-FOR Class1
DomainClass1 {
  ...
}
```

[3] This mechanism is also supported in [Der06] where it is referred to as "deification".

This operator indicates a recursive mapping definition for the class *Class1*. As a consequence, a domain class as well as contained domain entities for all the implementation entities contained in the definition of the OO are defined. Moreover, this mechanism recursively propagates to all the OO classes involved in the methods and attributes defined in the OO class, and so on until domain entities for all OO classes and their attributes and methods have been created at the domain level.

## 6.4 Use Case 2: Mapping One-to-Many Classes

We have seen how our mapping language allows mapping a domain class to an existing OO class. We have also seen that in the definition of the domain entities contained in that domain class, it is possible to refer to attributes or methods that are not directly defined in the core class — to which the domain class maps — but that can be reached through navigations in the implementation. Thus, in the case the domain class defines entities that map to several classes, the domain class itself is conceptually considered to be mapped to all those classes as well. However, this mapping to many classes is not made explicit in the mapping language.

We can imagine the need for explicitly mapping a domain class to many OO classes. This can be achieved by introducing new grouping operators in our mapping language. Two cases are identified: 1) a domain class maps to the result of the union of two or more OO classes; 2) a domain class maps to the result of the intersection of two or more OO classes. Two operators *MAP-TO-UNION* and *MAP-TO-INTERSECTION* are respectively introduced for the realization of these cases. These operators enhance the mapping language in two directions:

- *syntactic sugar*: the union and intersection operators can be seen as shortcuts for the definition of the contained domain entities in the domain class that is being mapped. In the case of the union for instance, a `DomainClass1 MAPS-TO-UNION Class1, Class2` definition would imply carrying out the actual union of *Class1* and *Class2* and pulling up the result of that union as domain entities of *DomainClass1*.
- *mapping verification*: an explicit definition of a mapping to a union can also enable compatibility checks between the mappings of the domain class on the one hand and the mappings of its contained domain entities on the other hand. More concretely, it becomes possible to verify that the contained domain entities map to OO entities defined in the classes involved in the union.

An important issue is that although these explicit one-to-many mappings are defined in terms of classes, the correspondence between the actual instances of those classes needs to be analyzed, which is not always trivial. As shown earlier in this dissertation (section 5.3), instances of domain classes are manipulated in the high-level specifications of rules and connections. For example, a high-level rule can define — in its *USING* clause — that it expects an instance of the domain class *DomainClass*. This means that, when this rule gets translated to code, in place of the expected domain instance, an object needs to be obtained. Thus, the correspondence between domain instances and core instances needs to be established. In the case of a one-to-one mapping, establishing the correspondence between an instance of a domain class and an instance of the core class is straightforward: a domain class instance is mapped to an instance of the core class to which the domain class maps,

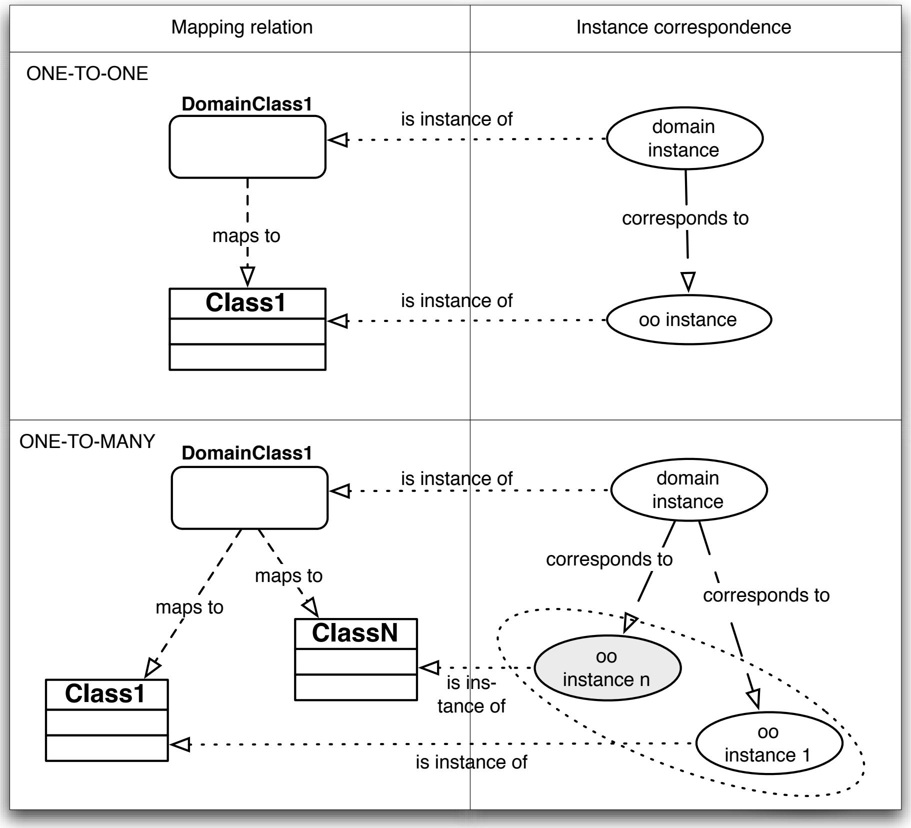


*Table 6.1:* Instance correspondence relation between a domain class and the core classes to which the domain class maps

and thus in this case a one-to-one relation also exists at the instance level. However, establishing this correspondence is more complex in the case of one-to-many mappings, since more than one core instance might need to be obtained for a given domain instance. Which and how many objects are needed depends on how the OO classes, to which the domain class maps, are related to each other in the implementation. The instance correspondence is depicted in Table 6.1. The grey oval in the case of the one-to-many mappings represents the instance that needs to be obtained. The oval around the two instances represents that somehow those instances need to be combined. This is analyzed in more detail in the coming sections.

### 6.4.1 Mapping to Union

The domain class is the union of many core classes. In other words, the domain class can be seen as the result of merging and pulling up the core classes. The contained domain entities — defined in the domain class — map to core entities defined in any of the core classes. For the coming explanations, consider the example of a domain class *DomainClass1* that maps to the union of two core classes, *Class1* and *Class2*.

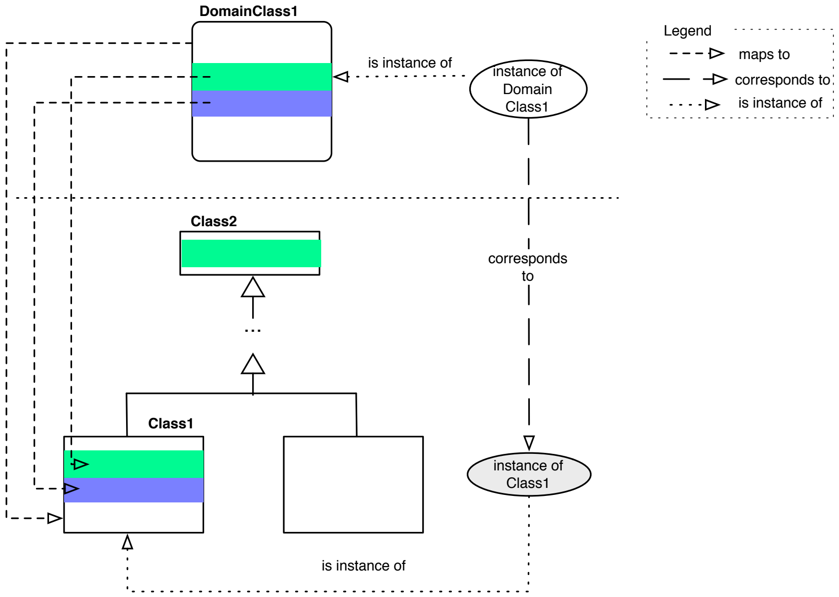


*Figure 6.2:* Mapping to a union of core classes: Class1 is a (direct or indirect) subclass of Class2

**Instance correspondence:** in order to determine the instance correspondence, it is needed to analyze how the attributes and methods defined in Class1 and Class2, which are realizations of the contained domain entities defined in DomainClass1, are related. We distinguish between the following cases:

1. *The relevant information defined in Class2 is inherited by its subclass Class1*;
2. *The relevant information to be drawn from Class2 is navigable from Class1*; and
3. *The relevant information to be drawn from Class2 is neither navigable from Class1 nor inherited by it.*

*1. The relevant information defined in Class2 is inherited by its subclass Class1*

Some domain entities defined in *DomainClass1* are realized by the class *Class1* whereas others are realized by its superclass *Class2* (Figure 6.2). Consider the case in which both classes are in the same inheritance chain of a class hierarchy and that *Class1* is a subclass (direct or indirect) of *Class2*. Due to inheritance, *Class1* inherits the core entities defined in *Class2*. Therefore, this mapping to many classes is analogous to a mapping to one class, namely *Class1*, where all the contained domain entities are realized by core entities defined in *Class1*. This implies that only one instance of *Class1* is needed for a given instance of *DomainClass1*.

*2. The relevant information to be drawn from Class2 is navigable from Class1*

*Class2* is accessible from *Class1* through navigation (involving attributes and method invocations). This implies that in order to obtain the implementation entity $ie_2$, defined in Class2, which realizes the domain entity $de_2$, a navigational expression starting from an instance of Class1 can be defined, which follows the form: $obj_{C1}.ie_3.....ie_2$, where $obj_{C1}$ is an instance of Class1. This is depicted in Figure 6.3. Note that instances of each of the two core classes are needed to realize a given instance of *DomainClass1*. However, only obtaining the instance of *Class1* is sufficient since the corresponding instance of *Class2* can be obtained from the former through navigation.

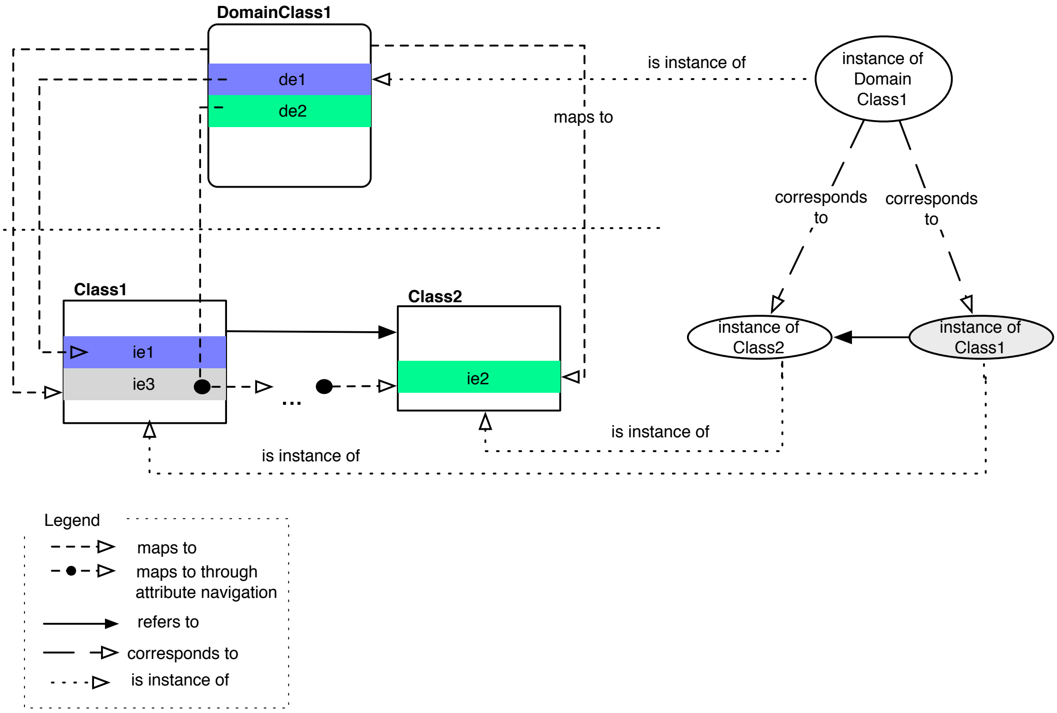


*Figure 6.3:* Mapping to a union of core classes: Class2 is navigable from Class1

*3. The relevant information to be drawn from Class2 is neither navigable from Class1 nor inherited by it*: This situation is depicted in Figure 6.4. In this case, classes are not related with respect to the required information. Note that associations might still exist between the classes, but those associations do not relate the classes in the desired way. Therefore, because the relation between the required instances of *Class1* and *Class2* is not explicit, navigations to refer to the required entities *ie1* and *ie2* cannot be expressed starting from one instance to the other or vice versa. Thus, the mappings for those domain entities *de1* and *de2* cannot be expressed as navigational expressions starting from either class.

In this case, a single instance of either class is not enough to draw all the required information. Instead, an instance of each of those classes needs to be obtained. The specification shown below illustrates how to refer to these different instances. The specification of the class name followed by the brackets enclosing the target variable indicates that the corresponding instance of that class needs to be obtained. The same definition could be expressed at the domain level in an analogous way.

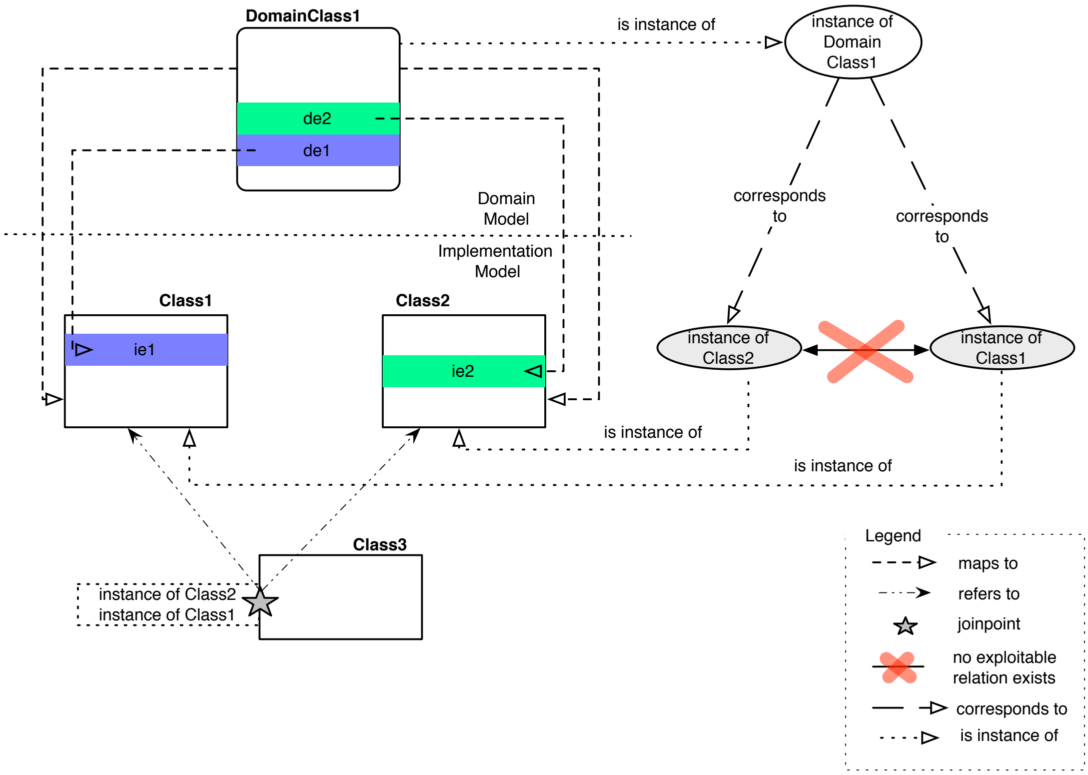


*Figure 6.4:* Mapping to a union of core classes: the relevant instances of Class1 and Class2 involved in the mapping are not related with respect to the required information.

```
DomainClass1 MAP-TO-UNION Class1, Class2
DomainClass1 {
 e.do1(x, y) MAP-TO-IMPL Class1(e).m(x, y)
 e.dp1 MAP-TO-IMPL Class2:e.ia1
}
```

We consider the case where the instance correspondence can be determined at run time, at well-defined points in the execution of the core application. We propose using AOP to grab and explicitly maintain the relation between instances that are implicitly related. In our approach, the well-defined points correspond to method executions. Pointcuts are used to intercept the application's execution and aspects are used to keep the link between the relevant instances. For this approach to be usable, AOP needs to be used again in a transparent way, i.e. avoiding cluttering the mapping language with extra technical complexity. We propose extending the previous definition with a specification that indicates how instances are grabbed from the context of a method execution, as shown below. In line 1, the $R$ between the square brackets after the class names in the $< RHS >$ specifies that the instance of that class is the return value, whereas $P$ is used for denoting that what needs to be grabbed is a parameter of that method invocation. Optionally (though not illustrated in the listing below) the $T$ option can be specified to indicate that the instance of interest is the target value.

```
1  DomainClass1 MAP-TO-UNION Class1[P(0)],Class2[R]: ClassX.method(ParamClass1,...,ParamClassN)
2  DomainClass1 {
3    ...
4  }
```

**Example:** consider the scenario in Figure 6.5.

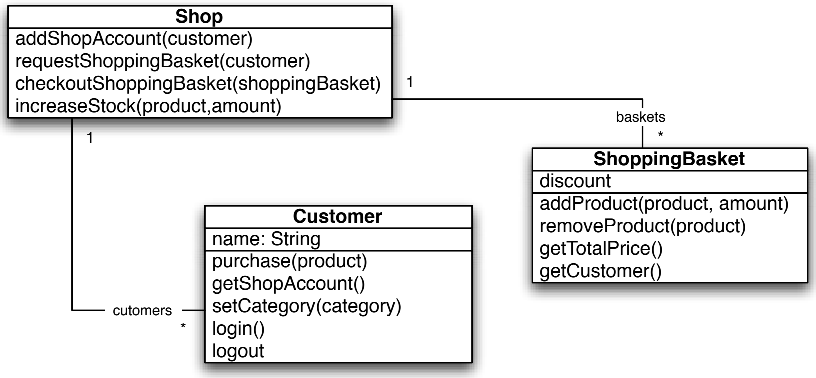


*Figure 6.5:* Example design scenario in the e-commerce application

In this design it is not possible to obtain the shopping basket for a given customer with simple navigations. This relation is dynamically established at the moment a customer requests a new shopping basket to an instance of the `Shop` class. This is achieved by invoking the `requestShoppingBasket(customer)` method on a shop object. It is at that point in time that the relation between the shopping basket returned by this method and the customer received as parameter of the method can be established. The specification of the mapping for the *Customer* domain class as the union of the `Customer` and the `ShoppingBasket` classes is shown in the listing below.

```
Customer MAP-TO-UNION ecommerce.Customer[P(0)], ecommerce.ShoppingBasket[R]:
                        ecommerce.Shop.requestShoppingBasket(ecommerce.Customer)
Customer {
  ...
}
```

### 6.4.2 Mapping to Intersection

A domain class that maps to the intersection of many core classes abstracts the commonalities of those classes. In our mapping language, an intersection is defined as follows:

```
1 DomainClass1 MAP-TO-INTERSECTION Class1,..., ClassN
2 DomainClass1 {
3  e.de1 MAP-TO-IMPL e.ie1
4 }
```

This declaration implies that the contained domain entities defined in the *DomainClass1* are realized by implementation entities that are common to all the classes *Class1*, ..., *ClassN*, in this example *ie1* (used in line 3). This situation is illustrated in Figure 6.6.

**Instance correspondence:** for a given instance of the domain class, only one instance of any of the core classes is needed.

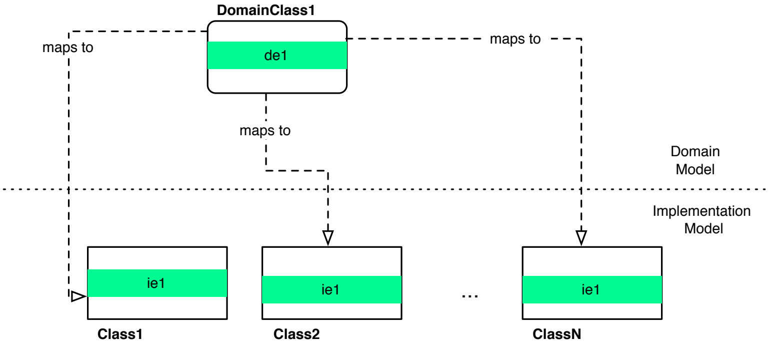


*Figure 6.6:* Mapping from a domain class to the intersection of many OO classes

**Example:** Consider the following mapping specification.

```
Product MAP-TO-INTERSECTION ecommerce.CD, ecommerce.DVD
Product {
 p.productID MAP-TO-IMPL p.productID
 p.price MAP-TO-IMPL p.price
}
```

In line 1, the domain class *Product* is defined which abstracts the commonalities of the core classes `ecommerce.CD` and `ecommerce.DVD`. In this example, both core classes define common attributes such as `productID` and `price`. The domain class can then define domain properties *productID* and *price* as domain abstractions of their counterparts at the implementation level (lines 2 to 5). Note that in the case the classes `CD` and `DVD` inherit from a common superclass, e.g. `Product`, which abstracts their commonalities, mapping the domain class *Product* to the core class `Product` would not be equivalent, as it would imply that the domain class *Product* is also an abstraction for other kinds of products which also inherit from the `Product` class (e.g. imagine for instance the existence of a class `Book` which also inherits from `Product`).

## 6.5 Use Case 3: Anticipated Mappings

We define the term *anticipated* with respect to an existing implementation. A domain entity is *anticipated* when its mapping can be defined in terms of one or many existing implementation entities. In our mapping language, a mapping for an anticipated domain entity can be expressed by means of directly or indirectly (via navigations) pointing to the existing implementation entities involved in its realization. In the following listing, examples of anticipated mappings in the e-commerce domain are shown.

```
Customer {
  c.name MAP-TO-IMPL c.nameCustomer

  c.account MAP-TO-IMPL c.getAccount()

  c.amountSpent MAP-TO-IMPL c.getShopAccount().amountSpent

  c.frequent MAP-TO-IMPL c.account.isFrequentCustomer()

  shop.resetCustomerAccount(name) MAP-TO-IMPL
          shop.setAmountToCustomerAccount(java.lang.Float: 0, java.lang.String: name)

  c.age MAP-TO-IMPL c.getAge()

  c.boughtProducts MAP-TO-IMPL c.getShopAccount().purchasedProducts

  c.addTenBoughtProducts() MAP-TO-IMPL c.getShopAccount().addBoughtProducts(10)
}
```

These example mappings are low-level since they are expressed in terms of implementation entities. In addition, our approach allows expressing mappings for anticipated domain entities entirely at the domain level, by using the *MAP-TO-DOMAIN* operator and referring to already defined domain entities in the $< RHS >$ of the mapping definition. Consider the following examples:

```
Customer {
  ...
  c.discount MAP-TO-DOMAIN percentage(10, c.amountSpent)
  c.youngerThan(c1) MAP-TO-DOMAIN c.age < c1.age
}
```

In line 3 a new discount domain property is defined as a percentage of the amount spent by that customer. The $< RHS >$ of this mapping definition calculates a derived value from another anticipated domain property of the customer, the *amountSpent*, and it involves the use of the *percentage* domain operator. Note that although the discount domain property is derived, no explicit means exists for calculating its value in the current implementation. In line 4 a domain operation *youngerThan(c1)* is added for which the mapping is defined as a comparison between the target and the parameter customers' ages. Again, the $< RHS >$ involves a calculation in terms of the *age*, an anticipated domain property of a customer, and $<$, a logical domain operator.

When the mapping of anticipated domain entities is defined completely at the domain level, the main challenge lies in translating the $< RHS >$ of the mapping specification into an expression that only involves implementation entities. This is because complex navigations, literal values and nested mappings might need to be taken into account in this translation.

## 6.6 Use Case 4: Calculating Values at Execution Points

In the previous use case we have explained how information that is derived from the existing implementation can be explicitly represented at the domain level. In this section we analyze the case where the derivation process is not as simple as evaluating a navigational, arithmetical or logical expression but it implies calculating a value based on information that is available at multiple points in the execution of the core application. Obtaining this information results in crosscutting code and thus the use of AOP is proposed. Special operators are provided as part of the mapping language to deal with this case: *timeBetween* and *previousResult* operators (presented in section 6.2.3). In the rest of this section we present concrete examples in the e-commerce domain of the use of these operators.

### 6.6.1 Dealing with Timing Information

Assume it is of interest to measure the time that elapses between the moment a customer orders a product until the customer checks out and confirms that order. This concept can be represented as a new domain property *timeBetweenOrderAndCheckout* which is added to the specification of the contained domain entities for the domain class *Customer*. The mapping for this new domain property is defined completely at the domain level using the *timeBetween* domain operator, as shown below.

```
Customer {
  ...
  c.checkout(sb) MAP-TO-IMPL c.checkoutShoppingBasket(ecommerce.ShoppingBasket:sb)
  c.addProductToShoppingBasket(p) MAP-TO-IMPL c.addProduct(ecommerce.Product:p)
  c.timeBetweenOrderAndCheckout MAP-TO-DOMAIN
            timeBetween(c.addProductToShoppingBasket(p), c.checkout(sb))
}
```

The connector that gets generated out of this mapping specification deploys the `TimeBetween` aspect on the OO methods `c.addProduct(ecommerce.Product)` and `c.checkoutShoppingBasket(ecommerce.ShoppingBasket)`.

### 6.6.2 Dealing with Cached Information

Assume the last amount spent by a customer during checkout wants to be made explicit at the domain level. This concept can be represented as a new domain property *lastAmountSpent* which is added to the specification of the contained domain entities for the domain class *Customer*. Assuming that the domain operation *checkout(sb)* returns the total amount spent by the customer in the checking out of the shopping basket received as parameter, the mapping for this new domain property is defined at the domain level using the *previousResult* domain operator, as shown below.

```
Customer {
  ...
  c.lastAmountSpent MAP-TO-DOMAIN previousResult(c.checkout(sb))
}
```

The connector that gets generated out of this mapping specification deploys the `PreviousResult` aspect (predefined in the domain model framework) on the `c.checkoutShoppingBasket(ecommerce.ShoppingBasket)` method.

## 6.7 Use Case 5: Unanticipated Mappings

The software applications that are considered in this dissertation tackle concerns of a particular problem domain which is strongly connected to the real-world (e-commerce, finance, etc.). As these domains are in constant change, the corresponding software systems need to constantly adapt to those changes. These systems — so-called E-type systems [Leh96] — are characterized by their impossibility to completely specify the problem of interest. On the one hand, our approach proposes building a model of domain entities on top of these volatile systems in order to extract the domain knowledge they encode in their implementation. On the other hand, our approach enables expanding the initial set of domain entities by means of incrementally extending the domain model with new domain entities. This allows for domain evolution. Note that the domain entities that result from extracting domain knowledge from the existing application only partially represent the corresponding domains, as only anticipated domain concepts (i.e. foreseen in the existing implementation) are made explicit. Exposing domain knowledge that is anticipated in the existing application is enough for expressing business rules that only require anticipated domain knowledge. However, the high-level — and executable — specification of business rules can be discrepant from the implementation of the core application functionality and therefore it might be required to talk about unanticipated domain knowledge (i.e. knowledge that is not present at all and cannot be derived from the existing implementation).

Thus, when it is not possible to identify explicit implementation entities in the existing application for the realization of a given domain concept, we say that the domain concept is *unanticipated*. Our mapping language deals with the realization of unanticipated domain concepts by means of the *MAP-TO-VALUE* operator. Because the implementation of this operator is based on static AOP, it becomes possible to add new implementation entities which realize the unanticipated domain knowledge.

Consider for example the situation where we want to introduce the concept of loyalty categories in the domain model for the e-commerce domain. The possible values for the loyalty categories can be for example *gold*, *silver* and *bronze*. Consider that there is no implementation entity realizing the loyalty category concept in the existing core application. We propose representing the customer category as a domain property of the *Customer* domain class, as follows:

```
1 Customer {
2   ...
3   c.loyaltyCategory MAP-TO-VALUE String:"silver"
4 }
```

In this example, the *loyaltyCategory* domain property defined in line 3 holds a *String* value which is initially set to *"silver"*. As a result of this definition, code is generated: a Java interface, a JAsCo mixin aspect implementing that interface and a JAsCo connector deploying the mixin aspect on the `Customer` class. This code is shown in code fragment 6.1.

```
public interface ILoyaltyCategory
   extends jasco.runtime.mixin.IMixin {

     public java.lang.String getLoyaltyCategory();

     public void setLoyaltyCategory(java.lang.String t);
}

class IntroduceLoyaltyCategory{

   hook IntroduceLoyaltyCategoryHook implements ILoyaltyCategory{

     private java.lang.String loyaltyCategory = "silver";

     IntroduceLoyaltyCategoryHook(method(..args)) {
      execution(method);
     }
      public void setLoyaltyCategory(java.lang.String t){
       loyaltyCategory = t;
      }
      public java.lang.String getLoyaltyCategory(){
       return loyaltyCategory;
      }
   }
}

static connector IntroduceCategoryToCustomer {
  perobject
    IntroduceLoyaltyCategory.IntroduceLoyaltyCategoryHook hook1 =
      new IntroduceLoyaltyCategory.IntroduceLoyaltyCategoryHook
                          (* ecommerce.Customer.*(*));
}
```

*Code Fragment 6.1:* Code generated for the realization of the loyaltyCategory domain property added to the Customer domain class

The `ILoyaltyCategory` interface defines getters and setters for the introduced attribute. The aspect bean `IntroduceLoyaltyCategory` defines a hook `IntroduceLoyaltyCategoryHook`. This hook implements the interface `ILoyaltyCategory` and defines the introduced attribute `loyaltyCategory` which is set to the initial value *"silver"*. This hook also provides implementations for the methods defined in the `ILoyaltyCategory` interface. The connector `IntroduceCategoryToCustomer` instantiates the hook `IntroduceLoyaltyCategoryHook` per customer, as a result creating a new instance of the `loyaltyCategory` attribute per customer object.

Because these domain properties are unanticipated in the core application, the latter is unaware of the existence of the attributes introduced for those domain properties. The only place where these domain properties are used is in business rules. Business rules can use the values of domain properties in their conditions and actions and can also calculate and set (using the *IS* operator) new values for those domain properties in their actions.

In the same way domain properties are realized by introduced attributes, we can imagine the need for introducing new methods that realize the implementation of unanticipated domain operations. Note that unanticipated domain operations are partly supported by mappings to derived expressions (explained in section 6.2). Full support for unanticipated domain operations would however imply writing full Java method bodies in mapping specifications, which is of course not the aim. Making unanticipated domain operations persistent in the implementation (by means of static AOP) does not make sense, as state does not need to be kept for the realization of these operations. Note that, on the contrary, the case of unanticipated domain properties does require persistency. This is because business rules can set values to domain properties that can in turn be used in other business rules. Aspects are used then to realize this persistency in a non-invasive way.

## 6.8 Obtaining a *Code Representation* for the Mapping

For every domain entity referred to in a high-level specification — either of a rule or rule connection — an expression in terms of implementation entities needs to be obtained in order to be included in the generated code for that rule or connection (as described in section 5.6). This process involves inspecting and translating mapping specifications to their code representations. This is simple in the case of a perfect mapping. However, it becomes more complex in the case of indirect mappings, mappings to many entities or mappings based on AOP, as more complex issues such as establishing the instance correspondence or retrieving concrete aspect instances need to be tackled. In chapter 5 we made abstraction of how this expression is obtained and assumed the existence of a function *CodeRepresentation* in charge of calculating these code representations. More details on this process are included in chapter 7.

## 6.9 Summary

In this chapter we made explicit the mapping between high-level domain entities and implementation. We built on current support and enhanced it in many directions, in order to enable the definition of more complex mappings than the simple one-to-one mappings provided today. We presented and implemented a mapping language and showed how this language can be used to realize relevant mapping use cases.

# Chapter 7

# Implementation

In this chapter we present the prototype implementation developed as a proof of concept for the ideas presented in this dissertation. This prototype supports the entire domain model explained in chapters 5 and 6. The core of this implementation is a framework of OO classes for representing business rules, business rule connections, domain entities and their mappings to implementation. Moreover, on top of this core framework, three high-level dedicated languages are implemented: one for the definition of high-level business rules (as described in 5.3), a second one supporting the definition of high-level business rule connections (as described in chapter 5.4) and a third one for the definition of domain entities and their mappings (as described in chapter 6). Parsers for these languages have been implemented. Semantical checks in charge of validating high-level specifications against the domain entities defined in the domain model are also supported. High-level specifications expressed in the dedicated rule and connection languages are automatically translated to OOP and AOP programs respectively, following the transformations described in 5.5. For each of the domain entities involved in those specifications, an expression that only involves implementation entities is obtained. This is achieved by inspecting the mapping of those domain entities. In the general case, this implementation expression involves OOP whereas for some more complex mapping cases AOP is used (following the guidelines described in 6). As a result of translating a high-level mapping specification, domain entities and mappings are created and used to populate the domain model. Examples taken from the e-commerce case study application are shown.

## 7.1 Selected Technologies

The following technologies have been used in the implementation of our prototype:

- Java for the implementation of the core domain model framework;
- JavaCC for the implementation of the parsers;
- JAsCo for the implementation of the aspects; and
- Velocity templates for the code generation of Java rule objects and JAsCo connection aspects.

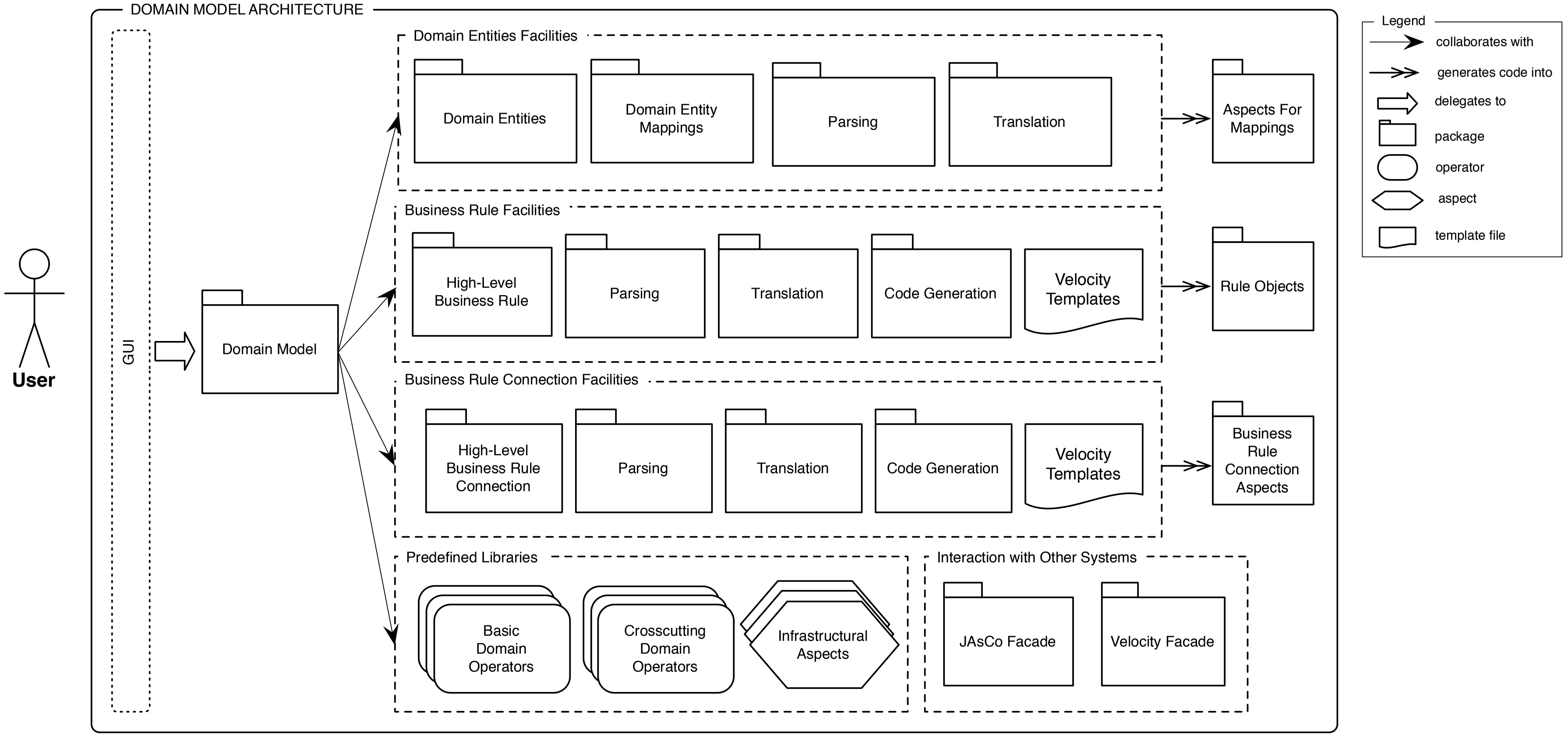


*Figure 7.1:* Overview of the domain model infrastructure

## 7.2 Architecture of the Domain Model Prototype

Figure 7.1 depicts the general overview of our prototype implementation. As it can be observed in the Figure, packages can be grouped according to the core part of the domain model framework they implement. We can mainly identify seven main core parts:

- *Domain Entities Facilities*: it denotes the classes for the representation of domain entities and their mappings as well as classes implementing the parsers and semantical translators for the high-level mapping language. As a result of parsing and translating some mapping specifications, aspects are generated which are gathered in a different package (labeled with *Aspects for Mappings* in Figure 7.1).

- *Business Rule Facilities*: it denotes the classes for the representation of high-level rules. Different rule representations are possible, as it will be explained in section 7.3. Moreover, this part of the framework includes packages which define classes for the implementation of the parsers, semantical checkers and code generators for the high-level business rule language. In addition, velocity templates are predefined as code templates that capture the structure of a rule object implementation in Java. For the actual code generation, a Velocity engine is invoked by classes in the *Code Generation* package. This invocation receives the deployment details to be taken into account in the instantiation of the velocity templates. As the result of this instantiation, a complete Java class is generated — implemented following the *Rule Object* pattern — which is compiled and loaded. The generated Java code is placed in a separate package, as depicted in Figure 7.1 under the name *Rule Objects*.

- *Business Rule Connection Facilities*: analogous to the *Business Rule Facilities*, this group of packages gathers classes for representing a rule connection from different perspectives (more details can be found in section 7.3). Moreover, this part of the framework also includes packages which contain the implementation of parsers, semantical checkers and code generators for the high-level business rule connection language. Three velocity templates are predefined: for the implementation of a hook, an aspect bean and a connector. These templates are deployed with the details of a concrete rule connection specification. The result of the code generation step is a complete JAsCo aspect bean — implementing the rule connection — and a JAsCo connector — deploying the generated aspect bean. The generated aspect bean and connector are compiled and loaded. The generated JAsCo code is placed in a separate package, labeled with *Business Rule Connection Aspect* in Figure 7.1.

- *Predefined Libraries*: as part of the implementation of the domain model, predefined domain operators are provided to be used in the specification of high-level rules. These operators include simple arithmetical (e.g. *addition*, *subtraction*, *multiplication*) and logical operators (e.g. *and*, *or*, *not* as well as comparison operators such as *greater than* and *equals to*, etc.). Moreover, following the ideas described in 6.2.3, examples of crosscutting operators are supported for which aspect templates are predefined in the domain model. An example of these operators is the *timeBetween* operator which takes two domain operations and calculates the time that passed in between the invocations of those domain operations. In addition, AOP is used for the implementation of crosscutting concerns in the framework. An example of such a crosscutting concern is *logging*. A logging aspect is implemented which logs all events that occur during the execution of rules and rule connections and displays relevant information about those events in the domain model's GUI.

- *Interaction with Other Systems*: the domain model interacts with other systems, namely JAsCo and Velocity. In order to support this interaction, facades have been implemented which encapsulate the invocations to those external systems.
- *Domain Model*: this package contains the definition of the main class representing the domain model, namely the `DM` class. Basically, this class is a container class for the business rules, business rule connections, domain entities and mappings defined in the framework (depicted in Figure 7.2). Moreover, a facade class is implemented which defines a simplified API for the communication with the `DM` class. Methods for loading and translating business rules and connections are defined as part of this facade, as well as methods for the definition of domain entities and mappings. This facade is invoked from the domain model's GUI.

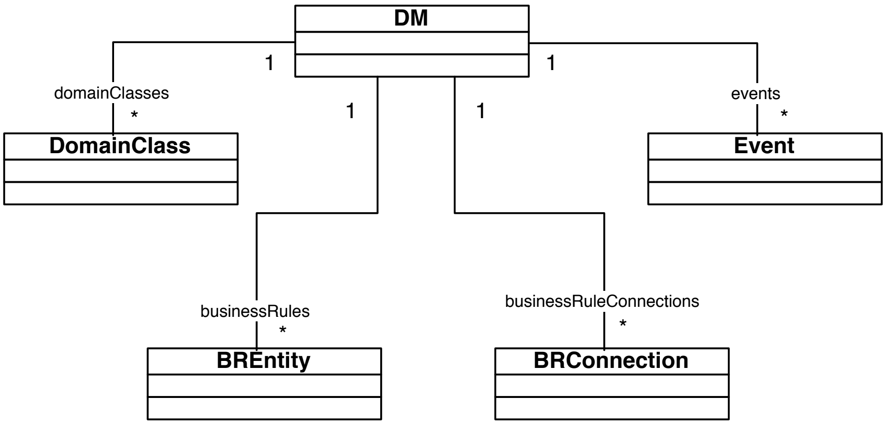


*Figure 7.2:* Overview of the DM class

- *GUI*: a prototype implementation of GUI have been developed which allows communicating with the domain model facade in order invoke the different functionality (shown in Figure 7.3).

## 7.3 Implementation Goals

The prototype has been implemented having in mind the following properties: *modularity*, *extensibility* and *flexibility*.

### 7.3.1 Modularity

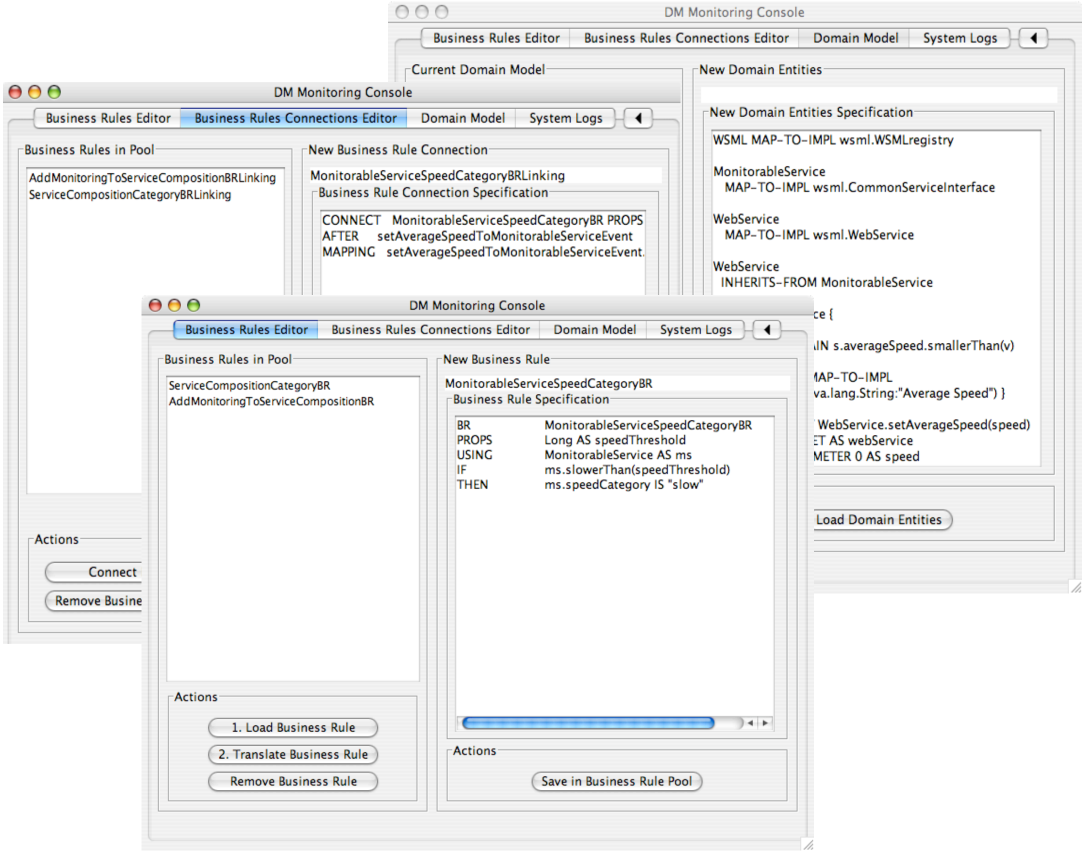
Modularity is achieved by dividing the implementation in separate interacting modules that realize the different parts of the framework:

- core infrastructure classes for the representation of domain entities, mappings, rules and rule connections;
- domain specific languages for:
  1. the definition of domain entities and their mappings to implementation;
  2. the definition of high-level rules;

3. the definition of high-level rule connections;

- classes implementing the translation process from high-level specifications to implementation, in OOP and AOP; and

- code templates encoding the transformation logic from high-level specifications to code.



```
WSML MAP-TO-IMPL wsml.WSMLregistry

MonitorableService
  MAP-TO-IMPL wsml.CommonServiceInterface

WebService
  MAP-TO-IMPL wsml.WebService

WebService
 INHERITS-FROM MonitorableService

ce {

IN s.averageSpeed.smallerThan(v)

AP-TO-IMPL
va.lang.String:"Average Speed") }

 WebService.setAverageSpeed(speed)
ET AS webService
METER 0 AS speed
```

```
AddMonitoringToServiceCompositionBRLinking
ServiceCompositionCategoryBRLinking
```

MonitorableServiceSpeedCategoryBRLinking

```
CONNECT   MonitorableServiceSpeedCategoryBR PROPS
AFTER    setAverageSpeedToMonitorableServiceEvent
MAPPING   setAverageSpeedToMonitorableServiceEvent.
```

```
ServiceCompositionCategoryBR
AddMonitoringToServiceCompositionBR
```

MonitorableServiceSpeedCategoryBR

```
BR          MonitorableServiceSpeedCategoryBR
PROPS       Long AS speedThreshold
USING       MonitorableService AS ms
IF          ms.slowerThan(speedThreshold)
THEN        ms.speedCategory IS "slow"
```

*Figure 7.3:* GUI of the domain model prototype

As hinted in section 7.2, domain model elements — i.e. high-level rules and connections as well as the domain entities and their mappings — have different representations along the translation process from their high-level specification to code. Each representation is realized by a set of classes or class hierarchies. In the case of a high-level rule, three are the possible representations, as described as follows:

- *syntactical*: corresponds to the representation of a high-level rule from the syntactical point of view. This representation is realized by a set of classes related in a class hierarchy (depicted by the classes enclosed by a dotted box in Figure 7.4) that correspond to the elements of the abstract grammar of the high-level rule language (available in Appendix A). The main class representing a high-level rule that is successfully parsed is `ASTBR`.

- *semantical*: corresponds to the representation of a high-level rule from the semantical point of view. This representation is realized by a set of classes related in a class hierarchy (depicted in Figure 7.5) that represent the different elements of a high-level rule which is successfully parsed and validated against the entities defined in the domain model (as presented in 5.3). The main class is this hierarchy is `BREntity`.

- *rule object implementation*: corresponds to the representation of a rule from the code generation point of view. In order to obtain this representation, all the entities that conform a `BREntity` (i.e. instances of type `CodeEntity`) need to be translated to their code representation. In our implementation, the code representation of a given code entity is realized using formatted strings (the process to obtain these strings is explained in 7.4.3).

Analogously, different representations are possible for the high-level rule connections:

- *syntactical*: corresponds to the representation of a high-level rule connection that adheres to the syntax of the high-level rule connection language. Classes are implemented (depicted in Figure 7.6) which correspond to the elements of the abstract grammar for the high-level rule connection language (available in Appendix B). The main class representing a high-level rule connection from a syntactical point of view is `ASTBRC` (BRC stands for business rule connection).

- *semantical*: corresponds to the representation of a high-level rule connection that complies with what is defined in the domain model, i.e. domain entities and their types. Classes are implemented (depicted in Figure 7.7) in correspondence to the different semantical features of a high-level rule connection (as presented in section 5.4). The main class representing a high-level rule connection from a semantical point of view is `BRConnectionEntity`.

- *rule connection implementation*: corresponds to the representation of the variable parts in the implementation of a rule connection aspect and connector. This is tackled in a similar way as for the rules, i.e. using formatted strings that represent the implementation of the different elements that constitute a given `BRConnectionEntity` object. These strings are the input for the code template instantiation.

Finally, with respect to domain entities and their mappings, only two representations are possible: syntactical and semantical. No code representation is needed. This is because, in the case of these elements, code is not generated as a result of the parsing-translation process. Instead, the expected result is a set of objects that are used to populate the domain model. These objects are instances of different classes which represent domain entities and mappings (depicted in Figure 7.9). More concretely, an entry in a mapping specification is of the form: ⟨*domain entity*⟩ ⟨*mapping operator*⟩ ⟨*mapping specification*⟩. The ⟨*domain entity*⟩ part is parsed and translated and as a result an instance of either *DomainClass*, *DomainOperation* or *DomainProperty* is created and stored in the domain model. The ⟨*mapping operator*⟩ part determines the kind of mapping that is needed for that domain entity. Again an instance of the corresponding subclass of `Mapping` is created and associated with the domain entity. The ⟨*mapping specification*⟩ provides the details of how the mapping instance needs to be configured.

- *syntactical*: correspond to the syntactical representation of domain entities and their mappings. The classes that implement this view (depicted in Figure 7.8) correspond to the different elements of the abstract grammar for the high-level business rule mapping language (available in Appendix C).

- *semantical*: correspond to a representation of domain entities and mappings that are successfully parsed and validated against existing entities in the domain model and in the core implementation. Classes for this view are depicted in Figure 7.9 and 7.10. They represent the main kinds of domain entities (as described in chapter 5) and the different kinds of mappings (as defined by the categories identified and presented in chapter 6).

It is important to stress that the use of GoF design patterns [GHJV95] was pursued in the implementation of all the component parts listed before with the objective to achieve a clean design. In particular the *Visitor Pattern* is extensibly used in order to traverse the many class hierarchies implementing the different perspectives of the elements of the domain model.

### 7.3.2 Extensibility

Because the rule and connection languages are defined based on a detailed analysis of the different elements that are distilled in the implementation of a rule and rule connection, their features are quite stable and not expected to change. Therefore, extensibility in the implementation of these languages is not a crucial concern. However, extensibility is a crucial concern for the implementation of the mapping language. This is because the presented mapping categorization (chapter 6) that forms the basis for the definition of the features of our high-level mapping language, is partial. This implies that new mapping categories could be considered which would imply adding new high-level features to our proposed high-level mapping language. This is why extensibility is especially important in the implementation of this language. The prototype implementation supports extensibility by means of two key design decisions:

- a well-modularized class hierarchy of mappings

- the implementation of several visitors which manipulate the mapping hierarchy

### 7.3.3 Flexibility

It is one of the goals of the prototype implementation to enable the dynamic integration of business rules in existing core applications. Therefore, the use of a technology that enables this flexibility is crucial. To this end, we have chosen to base our implementation on dynamic AOP technology, in particular JAsCo, since this technology allows the runtime plug-and-play addition and removal of aspects.

## 7.4 From a High-Level Business Rule to a Java Rule Object

A high-level rule expressed in the dedicated rule language, gets translated to a Java class implementing its corresponding rule object. Many steps conform this translation process, which are described in the rest of this section. An overview of the translation process is depicted in Figure 7.11.

### 7.4.1 Parsing

A javaCC parser for the high-level rule language is generated from the javaCC JJTree grammar of our language. The class implementing this parser is named `br.parser.BRParser`. Also, JavaCC automatically generates a class hierarchy for representing the different parts of a parsed tree. Figure 7.4 depicts the main classes implementing the parsing phase. Given a rule specification, the parser is in charge of checking whether its syntax complies with the syntax of the language specified in the grammar of the language. When this is the case, the parsing process is successful and as a result an object tree is built in memory representing the parsed tree.

### 7.4.2 Translating

A parsed tree is then passed to a second component which implements the translation phase. The main class that implements this translation is `br.translator.TranslationVisitor` (depicted in Figure 7.12). This class is in charge of translating the parsed tree representation of a rule into a well-modularized semantical rule representation. Two steps are identified in this translation phase: *validation* and *type checking* (depicted in Figure 7.12).

- The *validation* phase verifies the existence and correct use of domain entities. It makes sure that the domain entities referred to in the rule definition exists in the domain model with exactly the same names — and number of parameters in the case of a domain operation — as the ones employed in the rule's description. To this end, queries to the domain model need to be performed.

- The *type checking* phase makes sure that those domain entities are used in the expected way with respect to their types. This includes checking that the type of a domain property or the return type of a domain operation correspond to the types expected by the expression in which they are involved (logical or arithmetical). For instance, when the condition part of a rule is defined by an invocation to a domain operation, then the type of that domain operation must be `boolean`. Moreover, when domain expressions are used as parameters of a domain operation, their type compliance also has to be ensured. This applies as well to the case of expressions that involve more complex and nested navigations throughout the domain model as well as assignments. This translation step is implemented in the class `br.translator.TypeCheckingVisitor`.

  As the domain model is not typed, the expected types are determined by means of inspecting the domain entities mapping. For example, consider that *basket* is an instance of the *ShoppingBasket* domain class and that the following domain operation invocation is defined in the rule's action:

```
basket.setDiscountRate(0.5)
```

  then it is needed to check whether the type of the first parameter of this domain operation *setDiscountRate* is compatible with the value *0.5*. The same goes for more complex expressions, such as:

```
customer.account.amountSpent IS (customer.account.amountSpent + 10)
```

where the type that results from the expression *customer.account.amountSpent + 10* must be compatible with the type of *customer.account.amountSpent*.

Once a parsed rule is validated against the domain model, a semantical representation of the high-level rule is created. For this rule representation, a set of classes have been designed (partially shown in Figure 7.5), instances of which are created by the `TranslatorVisitor` class once the high-level rule has been validated. The actual result of this phase is an instance of the class `BREntity` which is stored in the domain model. When the validation process cannot be accomplished, an exception is thrown.

### 7.4.3 Generating Rule Code

This phase takes as input an instance of the `BREntity` class — returned by the translator in the previous step — and generates the Java code implementing a rule object for the corresponding high-level rule. The class implementing this phase is named `br.translator.BRCodeGenerationVisitor`. As an initial step, this code generator class needs to obtain different strings representing the implementation of each of the parts that conform a BREntity object. Each of these parts (e.g. condition and action parts) can involve other code entities (such as arithmetical or logical expressions, domain operation invocations or domain property retrievals and furthermore any combination of the previous ones). Thus, for every of these `CodeEntity` objects involved in a `BREntity`, a string needs to be obtained which represents the implementation of that code entity. This is a complex process because — as mentioned before — one code entity object can be defined in terms of other code entity objects, and this can even be continued recursively. Thus, for a given code entity, all the implementation strings for its contained code entities need to be combined in an appropriate way so that a single string is returned as a result (for example, for a single action involved in the definition of a `BREntity`, a single string needs to be obtained which represents a line of code implementing that action (to be included in the `actions` method of the generated rule class). Ultimately, the base code entities encountered during this translation process correspond to domain operation invocations and domain property retrievals. Translating these base code entities implies obtaining an implementation representation for the involved domain entities, which in turns implies inspecting and translating the mappings for those domain entities. This translation of a domain entity mapping is quite straightforward in the case of one-to-one mappings to implementation. However, it can become quite complex in the case of sophisticated mappings (e.g one-to-many mappings, unanticipated mappings, etc). The more sophisticated the mapping, the more complex the translation becomes.

To realize this translation process, string formatting capabilities provided by *jdk 1.5* are used. The strings that result from this process are passed to and used by the Velocity engine to instantiate the actual rule code template.

As a simple example, consider the translation depicted in Figure 7.13. Every action is translated to a separate line of code which is included in the `action` method of the rule code generated by the translation process described in 7.4. Every domain entity referred to in an action is translated to either a method invocation or an attribute retrieval, according to how the mapping for that domain entity is defined. In this example, the domain operation *customer.addTenBoughtProducts()* for instance maps to the core method *customer.account.addBoughtProducts(java.lang.Integer)*, where the parameter is fixed to the value *10*. Thus, the action *targetcustomer.addTenBoughtProducts()* gets translated to the method invocation `targetcustomer.account.addBoughtProducts(10)`, which is included in the implementation of the `action` method. Note that the fixed value *10*, specified as part of the mapping, is used in the translated invocation.

## 7.5 From a High-Level Business Rule Connection to a JAsCo Aspect

The translation process for a high-level rule connection is analogous to the one for a high-level rule. Now, given a specification of a high-level connection, expressed in the dedicated rule connection language, the implemented prototype performs an automatic translation to a JAsCo aspect bean and a JAsCo connector. In order to obtain these aspect code units, the transformations described in 5.6.2 are followed. Many steps conform this translation process, which are described in the rest of this section. An overview of the translation process is depicted in Figure 7.14.

### 7.5.1 Parsing

A parser for the high-level rule connection language is also implemented in JavaCC. Figure 7.6 depicts the main classes in the implementation of the parsing, including the main class `BRCParser` which is implements the parser itself and the class hierarchy — automatically generated by JavaCC — for representing the different parts of the parsed tree. This parser checks whether the syntax of a given high-level rule connection complies with the grammar of the high-level connection language. When this is the case, the parsing process is successful and as a result a parsed tree is built in memory representing the parsed high-level rule connection.

### 7.5.2 Translating

A parsed rule is then passed to a second component which implements the translation phase. The main classes implementing this translation phase are `brconnection.translator.TranslationVisitor` and `brconnection.translator.TypeCheckingVisitor` (depicted in Figure 7.15). This class is in charge of translating the parsed tree representation of a rule connection into a well-modularized semantical rule connection representation. Two steps are identified as well in this translation phase: *validation* and *type checking* (depicted in Figure 7.12).

- The *validation* phase verifies the existence and correct use of domain entities. In particular, it verifies that the involved events exist in the domain model and that they expose the referred contextual information items with the names that are used in the high-level rule connection specification. To this end, queries to the domain model have to be performed. Also, the information passed to the rule needs to be in accordance to what it is defined as expected in the rule, e.g. the number of values used for instantiating the rule and the number of contextual information items passed to the rule have to respectively coincide with the number of properties and parameters defined in the rule.

- The *type checking* phase makes sure that those domain entities are used in the expected way with respect to their types. This includes checking that the types of the referred contextual information items — exposed by the events — correspond to the types of the rule parameters to which they are mapped. Moreover, the types of the property values used to instantiate a rule need to be compatible with the types of the rules' properties. Again, for these type validations, the domain model needs to be queried. The types are determined by the mappings of the domain entities.

Once a parsed rule connection is validated against the domain model, a semantical representation of the high-level rule connection is created. For this rule connection representation, a set of classes have been designed (partially shown in Figure 7.7), instances of which are created by the `brconnection.translator.TranslationVisitor` class once the high-level rule connection has been validated and typed checked. The actual result of this phase is an instance of the class `BRConnectionEntity` which is stored in the domain model. When the validation process cannot be accomplished, an exception is thrown.

### 7.5.3 Generating Rule Connection Code

This phase takes as input an instance of the `BRConnectionEntity` class — returned by the translator in the previous phase — and generates JAsCo aspect bean and connector code implementing the corresponding high-level rule connection. The class implementing this phase is `brconnection.translator.BRConnectionCodeGenerationVisitor`. This class gathers the necessary deployment information which is used by the Velocity engine at template instantiation time.

Analogous to the rule code generation, as part of this process, every domain entity involved in a rule connection (e.g. events) needs to be translated to an expression in terms of implementation entities. This is achieved by inspecting the domain entity mappings (as explained in 7.4.3).

## 7.6 Translating Mapping Specifications

Two steps conform the translation process, which are described in the rest of this section. An overview of the translation process is depicted in Figure 7.16.

### 7.6.1 Parsing

The parsing process is analogous to the previously described parsing processes and therefore the details are omitted here. The main class implementing the parsing process is `domainmodel.mapping.parser.MappingParser`. When the parsing process succeeds, a parsed tree representation of the mapping is returned. This is illustrated in Figure 7.8.

### 7.6.2 Translating

In this phase, a parsed mapping declaration is validated against existing elements in:

- the core application, in the case of anticipated mappings in terms of existing core entities (defined using the operator MAPS-TO-IMPL); and
- the domain model, in the case of unanticipated mappings to domain (defined using the operator MAPS-TO-DOMAIN).

The class `domainmodel.mapping.translator.TranslationVisitor` implements this translation phase. As a result of this phase, a set of objects that are used to populate the domain model are created, which are instances of the classes representing domain entities and mappings (depicted in Figure 7.9). This step also involves type checking the expressions — both, domain and implementation expressions — which are part of the mapping definitions.

## 7.7 Challenges

During the implementation of the transformations, the following challenges were tackled:

- *resolving dependencies between transformations to AOP:* the transformation of a high-level specification that combines many high-level rule connection features is not as simple as concatenating the outputs of the individual transformations for the involved features. On the contrary, the different outputs have to be combined in a non-trivial way in order to obtain a running aspect. This makes the transformation process more complex as dependences between the individual transformations need to be taken into account.

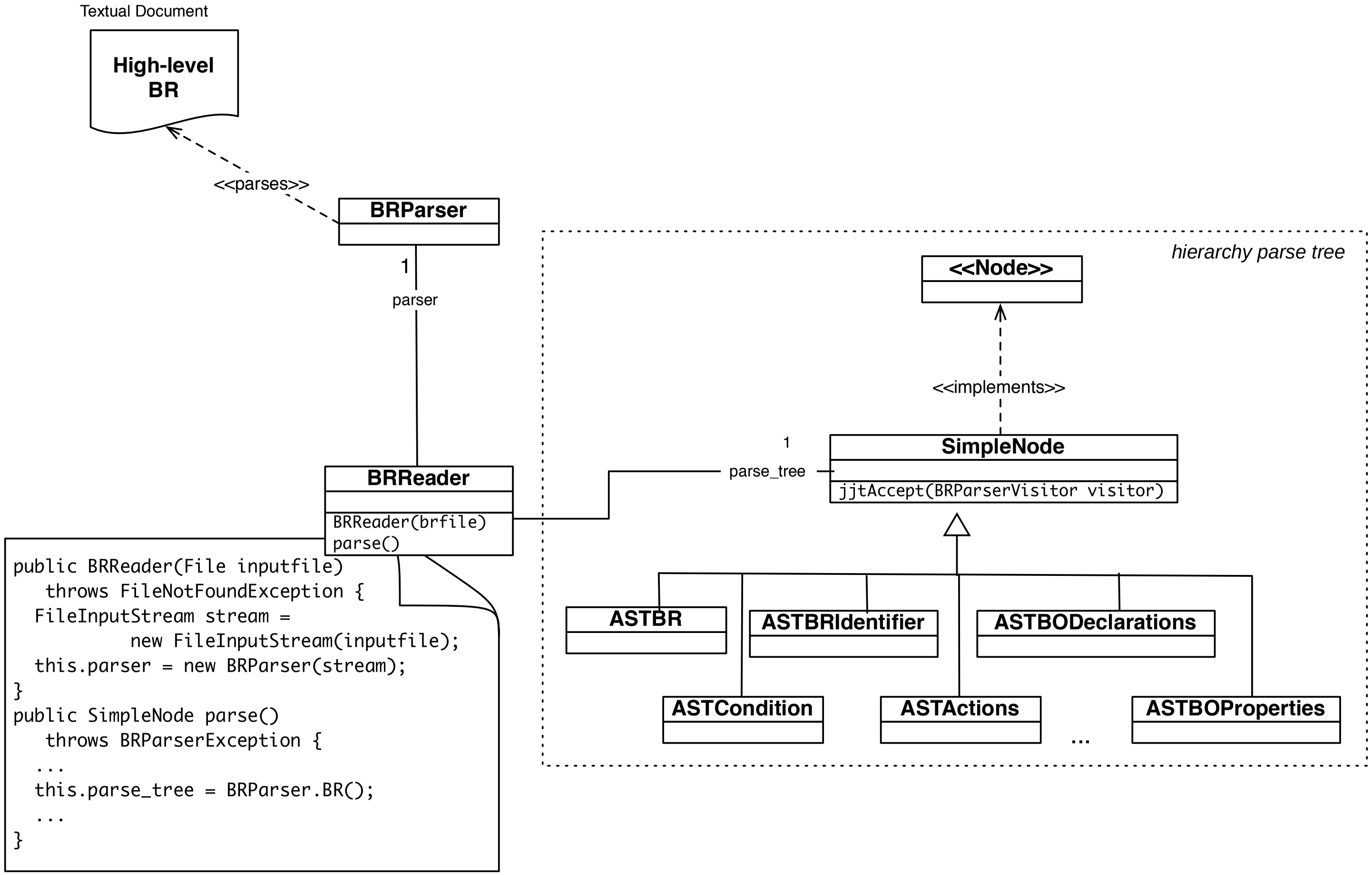
- *ensuring consistency*: at transformation time, the models involved in the high-level definitions need to be consulted in order to ensure consistency. Dependences between the models exist as the rule and connection models refer to elements in the domain entities model. Thus, the domain entities model needs to be consulted to check for the existence of the domain entities referred to in the rules and connections.

- *generating optimized implementations*: the generated OOP and AOP code only involve the constructs that are most adequate for each case. This makes the generated implementations more efficient.

- *translating domain entities to implementation*: during the transformation process of a rule and rule connection, the mappings of the involved domain entities are obtained in order to get an expression only in terms of implementation entities which is included in the generated code for those rule and connection. In case of one-to-one mappings — directly pointing to implementation entities — this step is simple, as only existing implementation entities need to be retrieved. However, this process becomes more complex as nested mappings need to be explored.

## 7.8 Summary

In this chapter we presented the prototype implementation developed as a proof of concept for the ideas presented in this dissertation. This prototype supports the entire domain model explained in chapters 5 and 6. Parsers and semantical checkers were implemented for the three high-level dedicated languages in our approach (high-level business rule, high-level business rule connection and mapping languages). The challenges encountered during the implementation of this prototype were discussed.



```
public BRReader(File inputfile)
   throws FileNotFoundException {
  FileInputStream stream =
           new FileInputStream(inputfile);
  this.parser = new BRParser(stream);
}
public SimpleNode parse()
   throws BRParserException {
  ...
  this.parse_tree = BRParser.BR();
  ...
}
```

*Figure 7.4:* Main classes implementing the parsing phase in the transformation of a high-level business rule

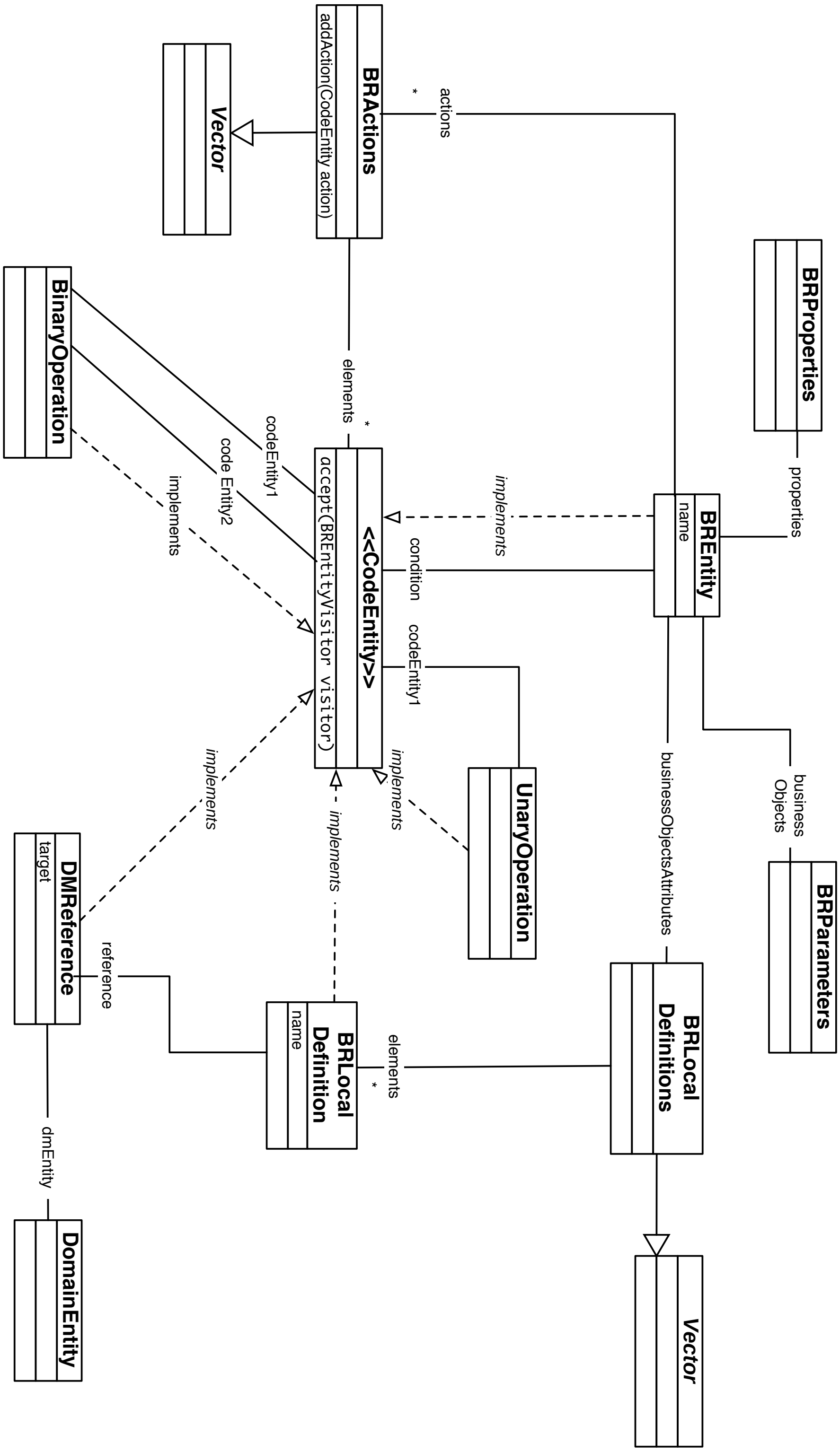
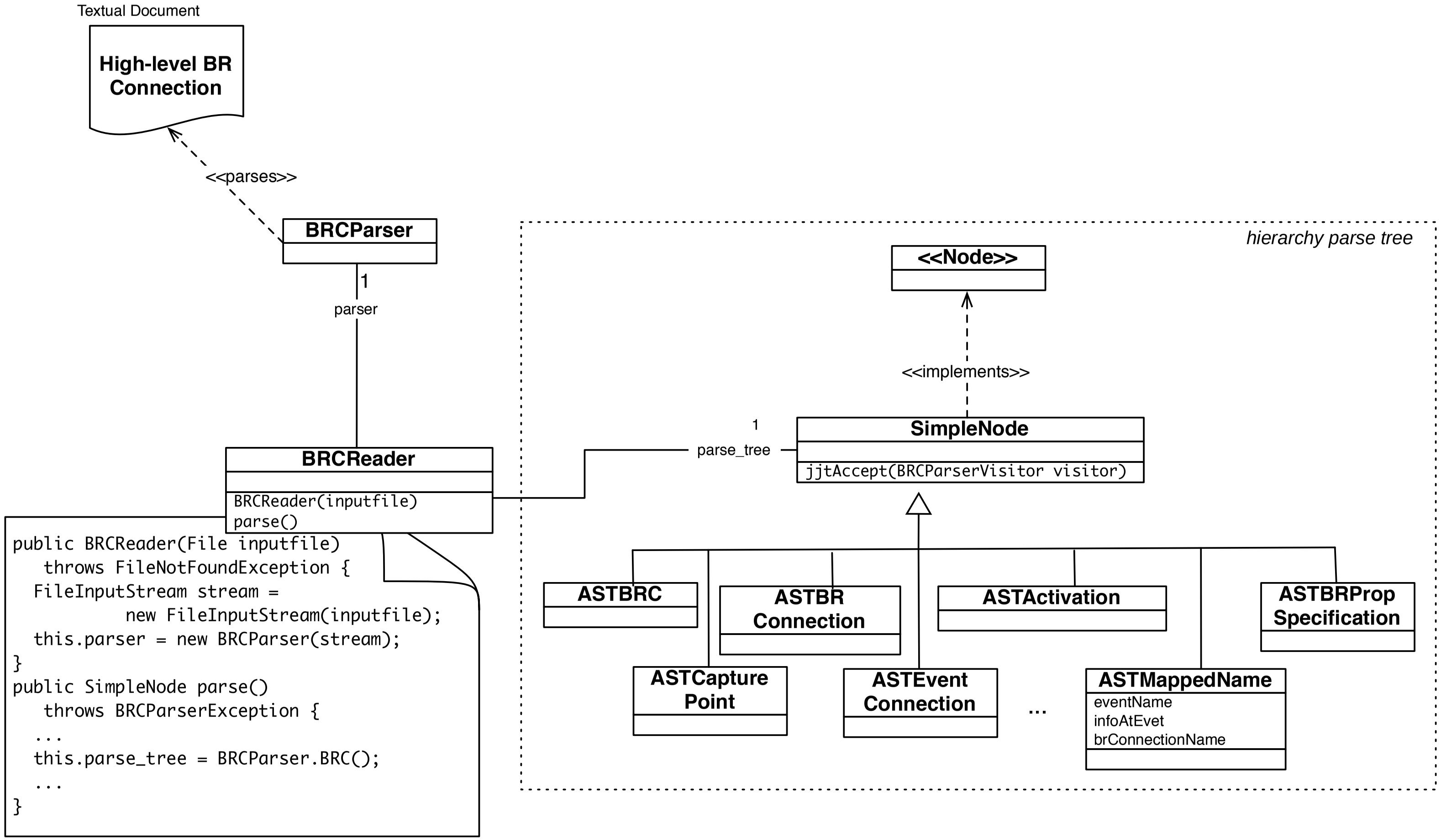


*Figure 7.5:* Main classes representing a parsed and translated high-level business rule



```
public BRCReader(File inputfile)
   throws FileNotFoundException {
  FileInputStream stream =
          new FileInputStream(inputfile);
  this.parser = new BRCParser(stream);
}
public SimpleNode parse()
   throws BRCParserException {
  ...
  this.parse_tree = BRCParser.BRC();
  ...
}
```

*Figure 7.6:* Main classes implementing the parsing phase in the transformation of a high-level business rule connection

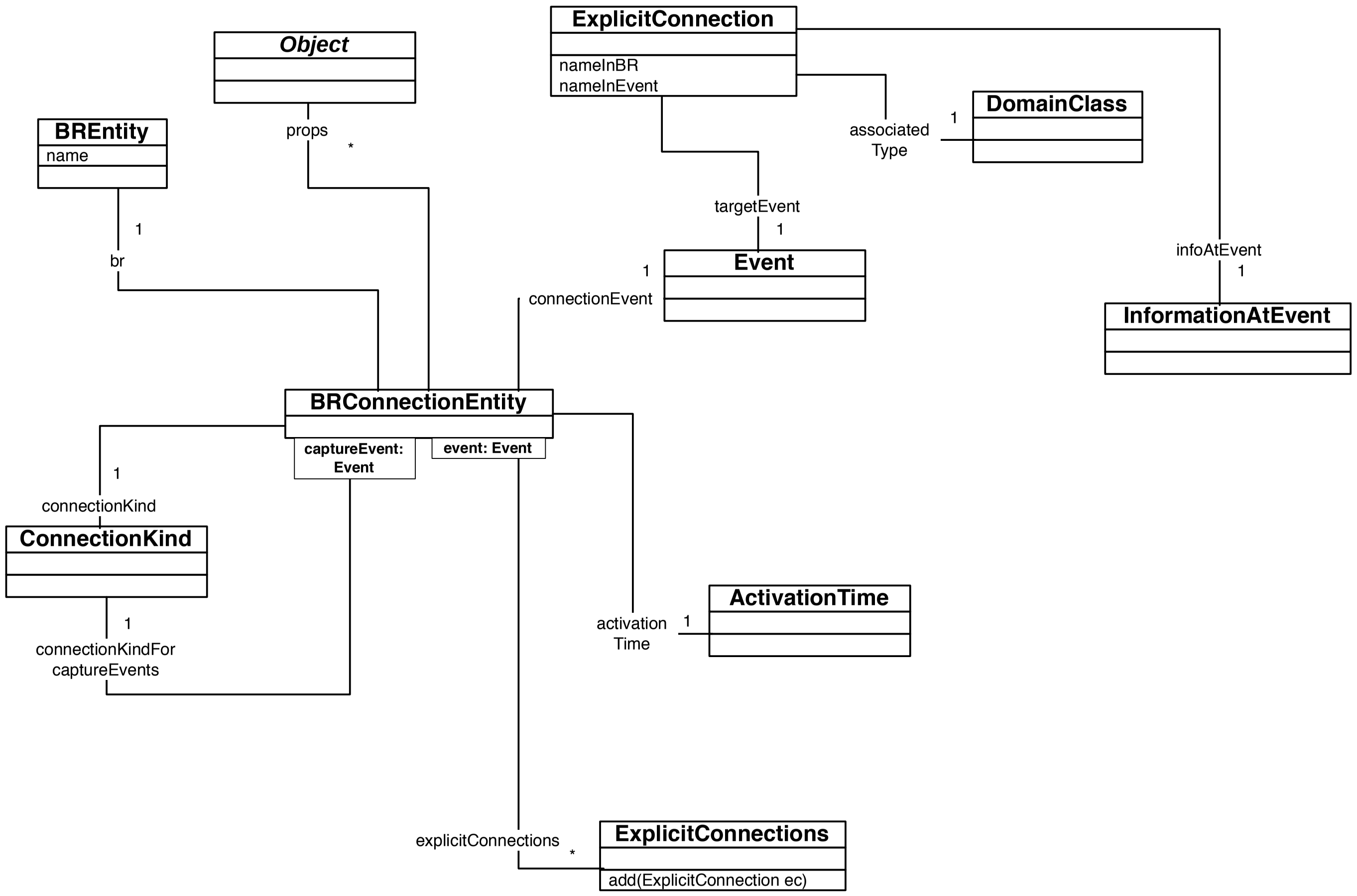
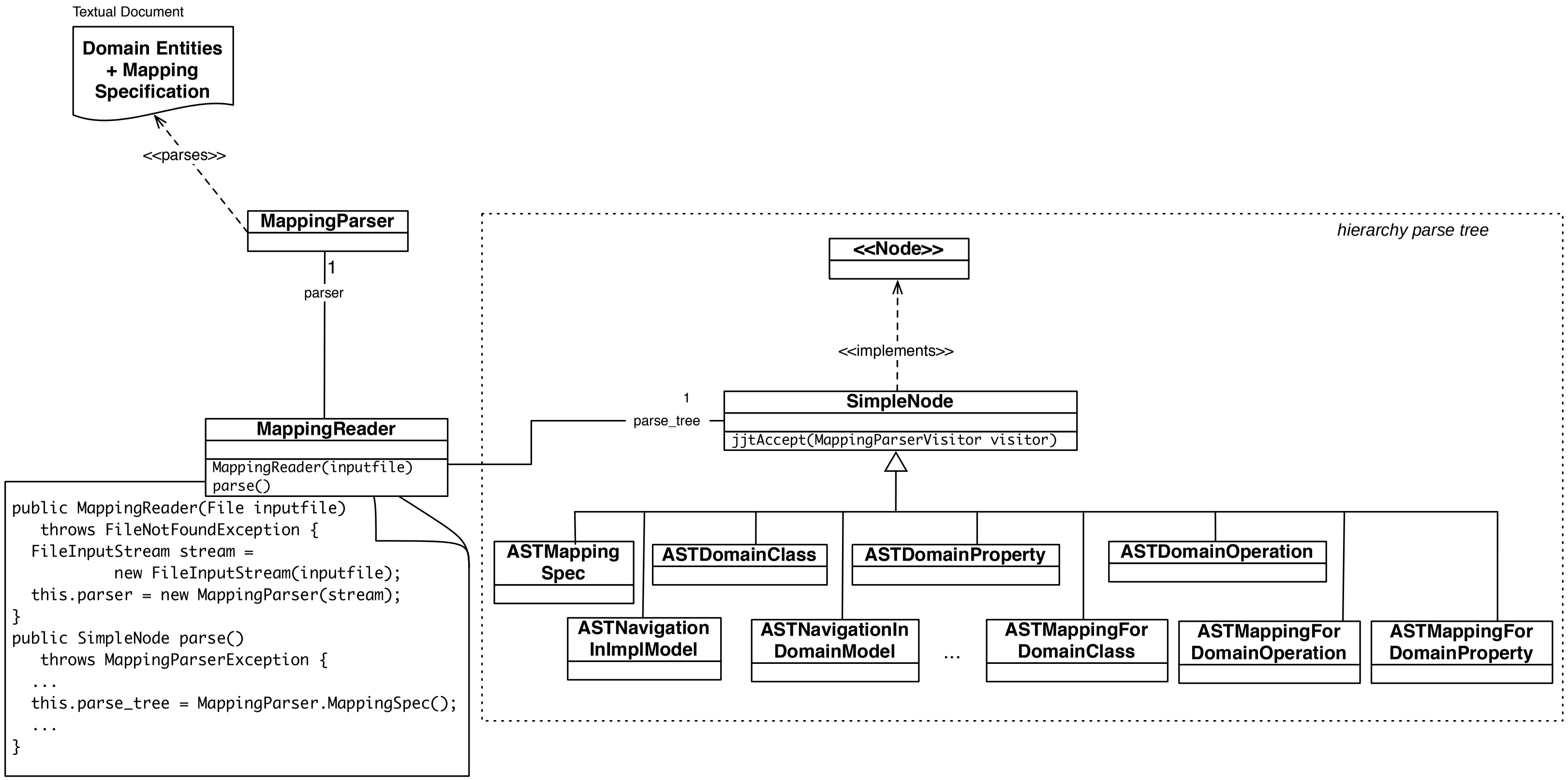


*Figure 7.7:* Main classes representing a parsed and translated high-level business rule connection



```
public MappingReader(File inputfile)
   throws FileNotFoundException {
  FileInputStream stream =
           new FileInputStream(inputfile);
  this.parser = new MappingParser(stream);
}
public SimpleNode parse()
   throws MappingParserException {
  ...
  this.parse_tree = MappingParser.MappingSpec();
  ...
}
```

*Figure 7.8:* Main classes implementing the parsing phase of the transformation process of a mapping specification

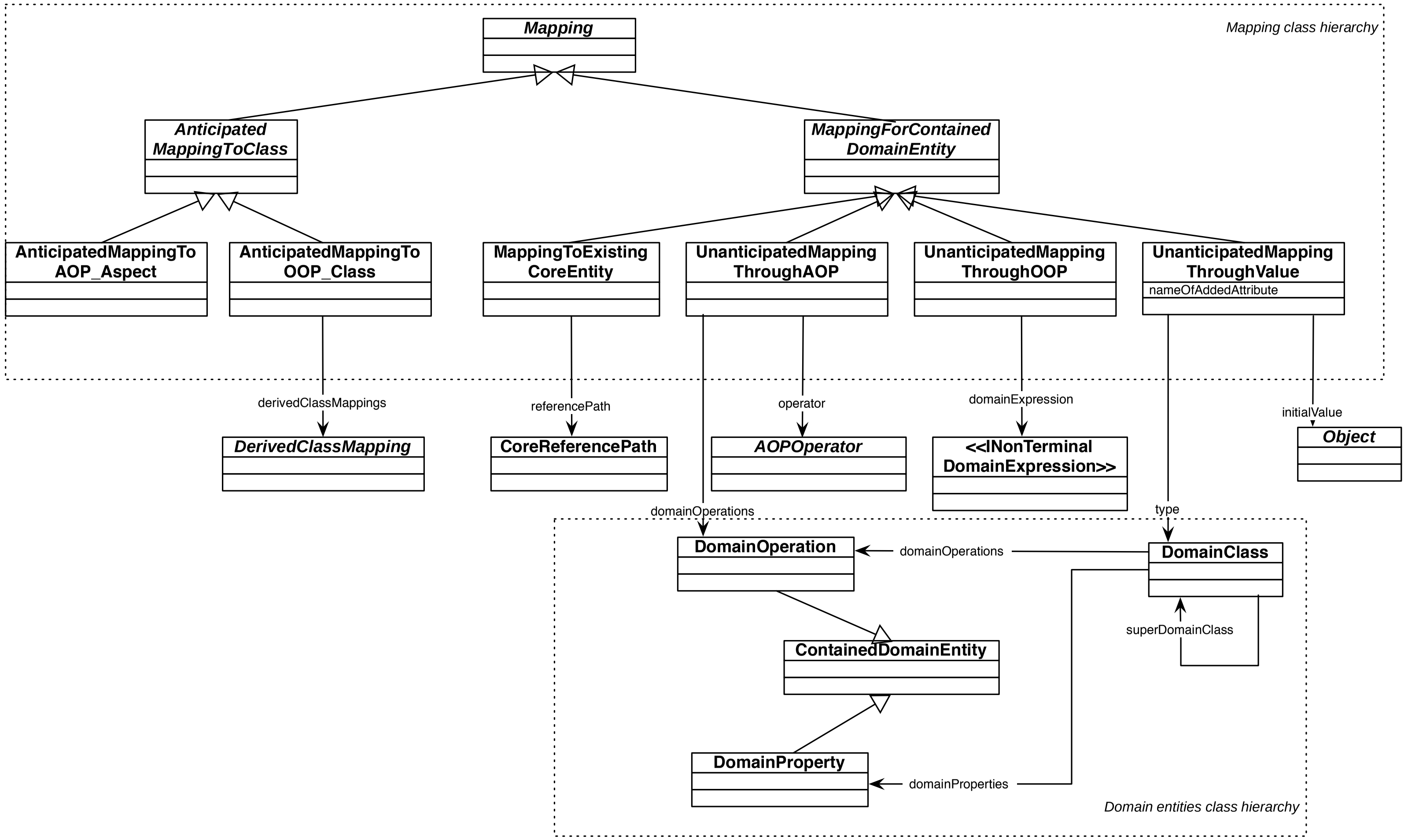

Mapping class hierarchy
Mapping
Anticipated MappingToClass
MappingForContained DomainEntity
AnticipatedMappingTo AOP_Aspect
AnticipatedMappingTo OOP_Class
MappingToExisting CoreEntity
UnanticipatedMapping ThroughAOP
UnanticipatedMapping ThroughOOP
UnanticipatedMapping ThroughValue
nameOfAddedAttribute
derivedClassMappings
referencePath
operator
domainExpression
initialValue
DerivedClassMapping
CoreReferencePath
AOPOperator
<<INonTerminal DomainExpression>>
Object
domainOperations
type
DomainOperation
domainOperations
DomainClass
superDomainClass
ContainedDomainEntity
DomainProperty
domainProperties
Domain entities class hierarchy

*Figure 7.9:* Main classes representing a parsed and translated mapping

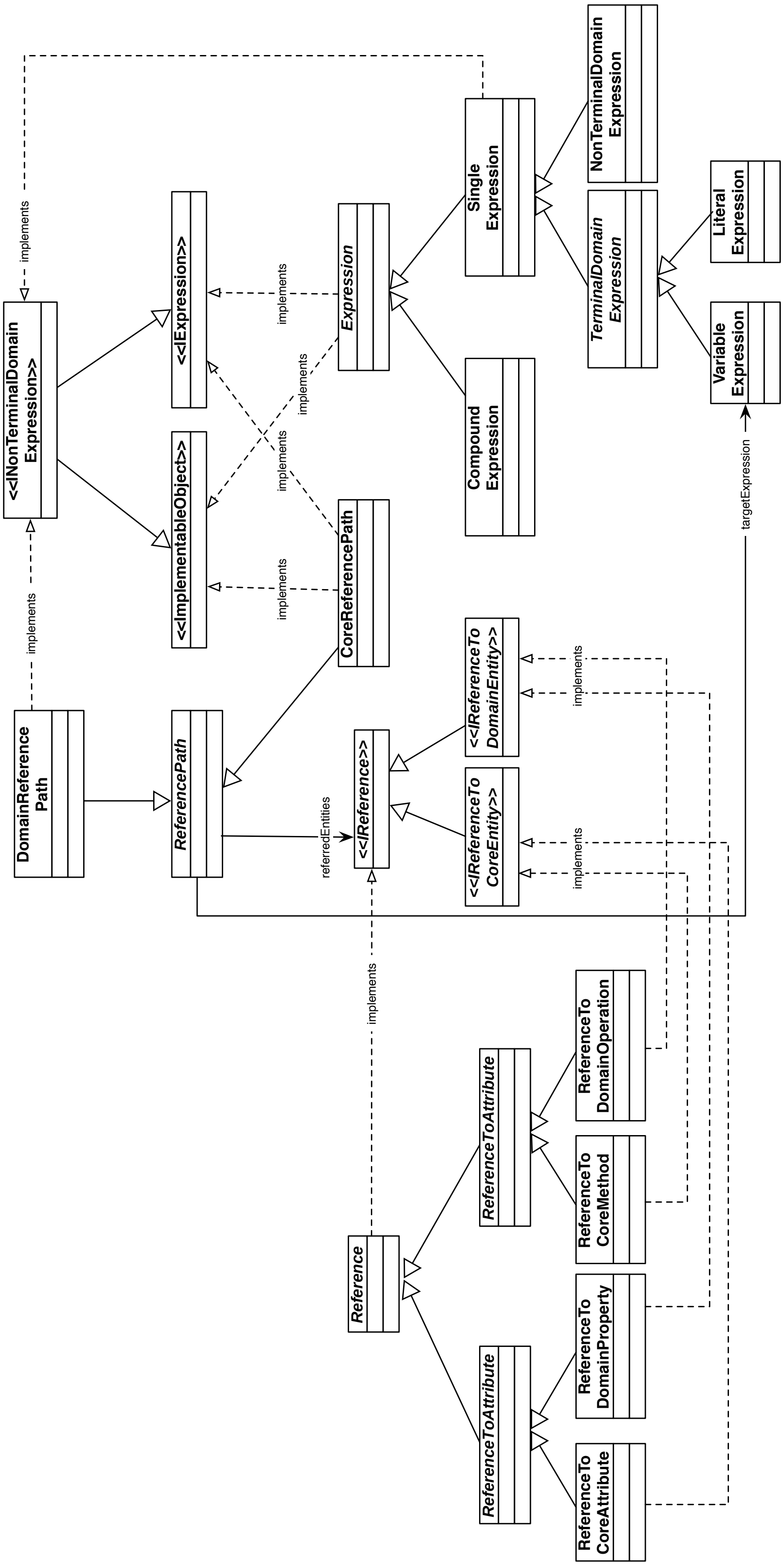


*Figure 7.10:* Main classes representing the kinds of expressions allowed in the definition of a mapping

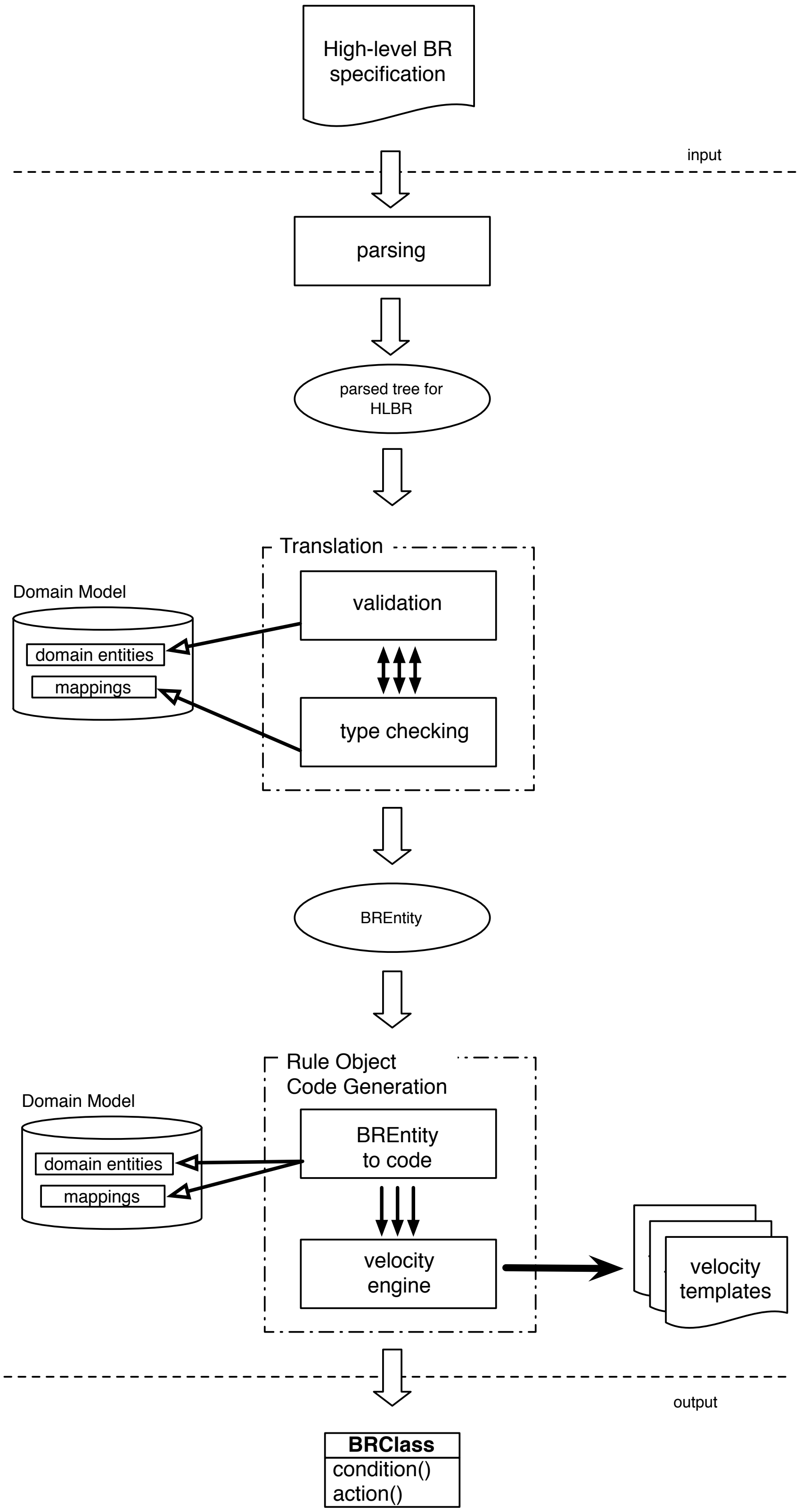
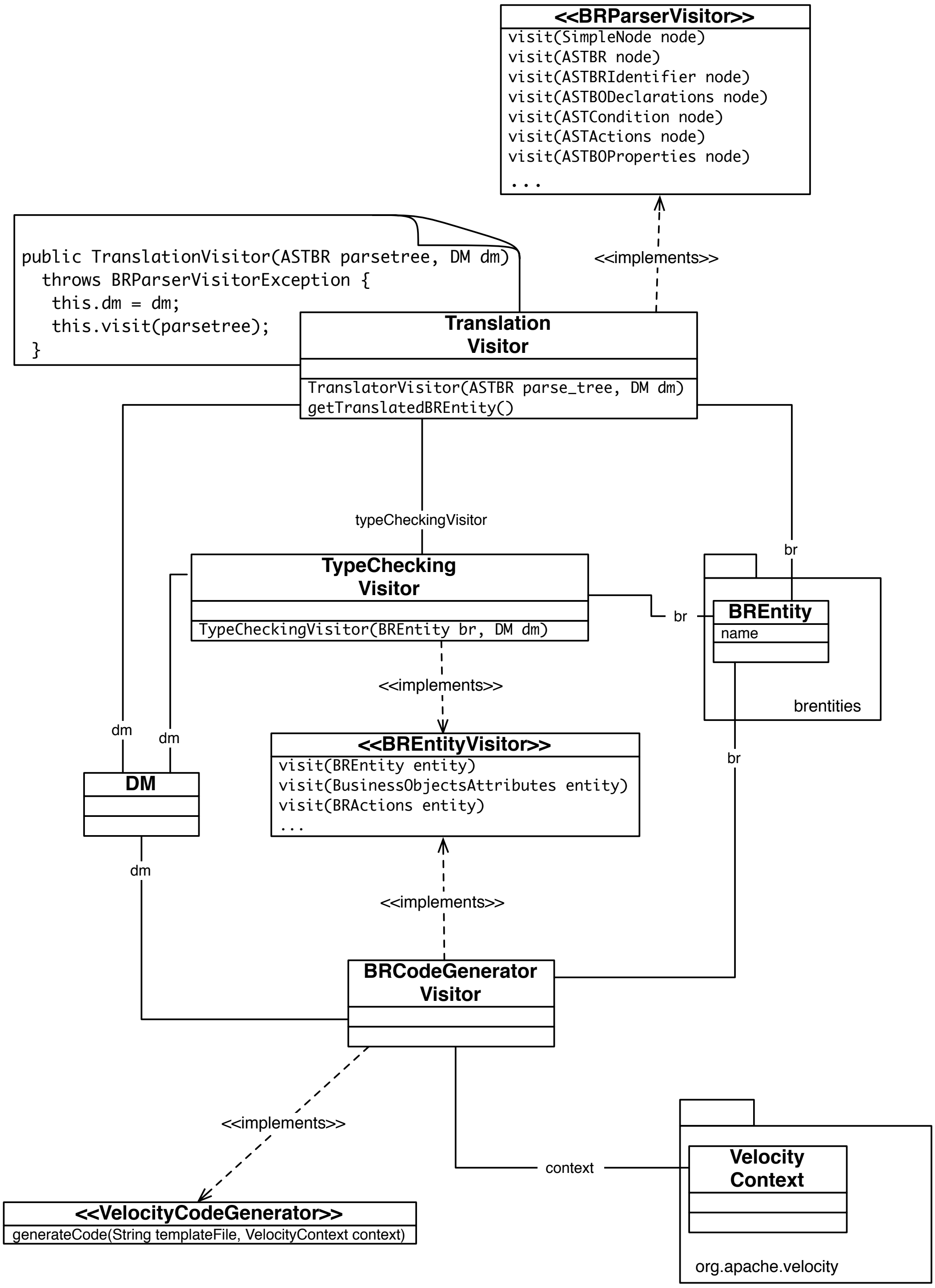
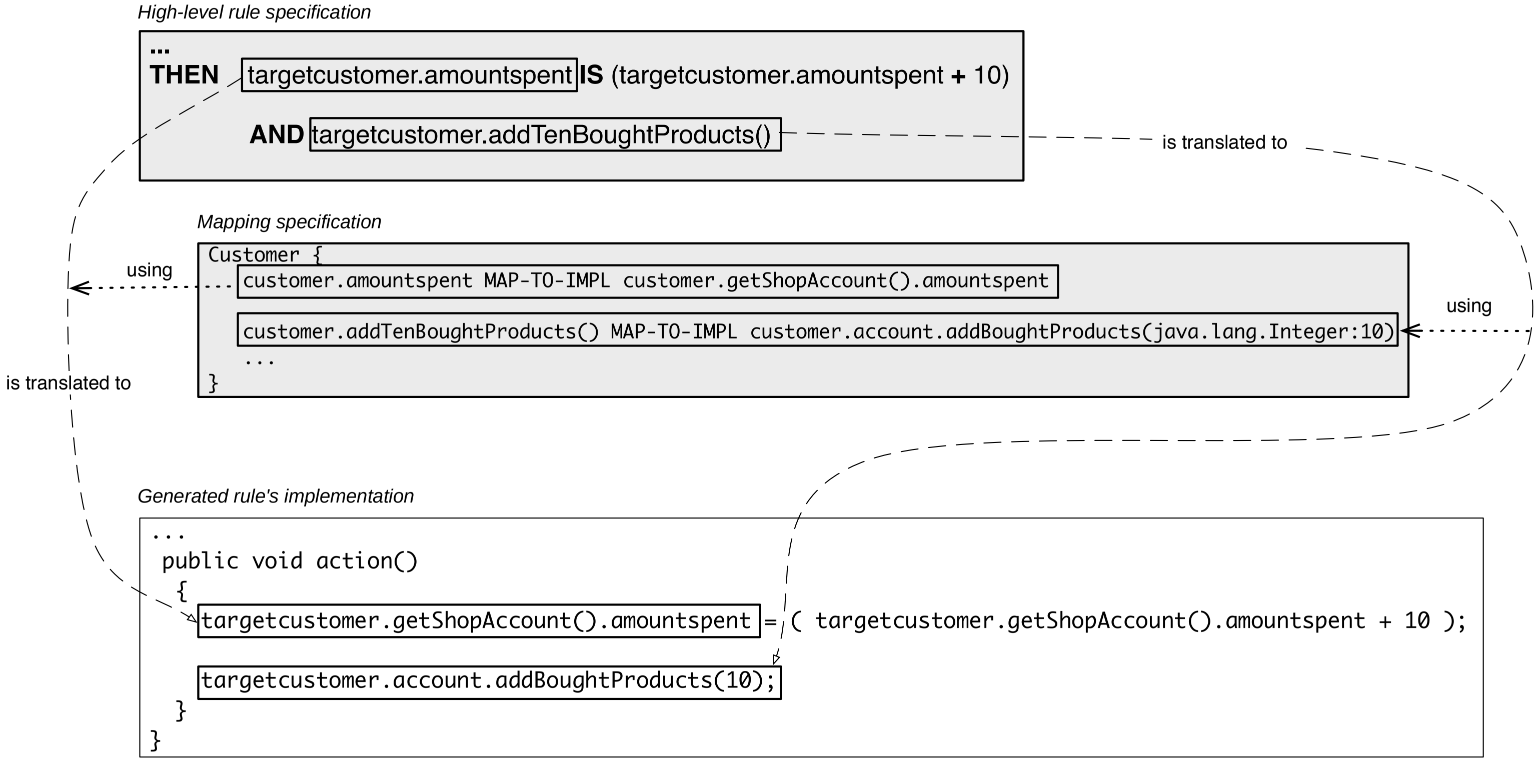


*Figure 7.11:* Overview of the transformation process of a high-level business rule



```
public TranslationVisitor(ASTBR parsetree, DM dm)
  throws BRParserVisitorException {
   this.dm = dm;
   this.visit(parsetree);
 }
```

*Figure 7.12:* Main classes implementing the translation phase in the transformation of a high-level business rule



```
...
THEN targetcustomer.amountspent IS (targetcustomer.amountspent + 10)

     AND targetcustomer.addTenBoughtProducts()
```

```
Customer {
  customer.amountspent MAP-TO-IMPL customer.getShopAccount().amountspent

  customer.addTenBoughtProducts() MAP-TO-IMPL customer.account.addBoughtProducts(java.lang.Integer:10)
  ...
}
```

```
...
 public void action()
  {
   targetcustomer.getShopAccount().amountspent = ( targetcustomer.getShopAccount().amountspent + 10 );

   targetcustomer.account.addBoughtProducts(10);
  }
}
```

*Figure 7.13:* Example of the translation from a concrete domain entity, involved in the definition of a high-level rule, to its implementation

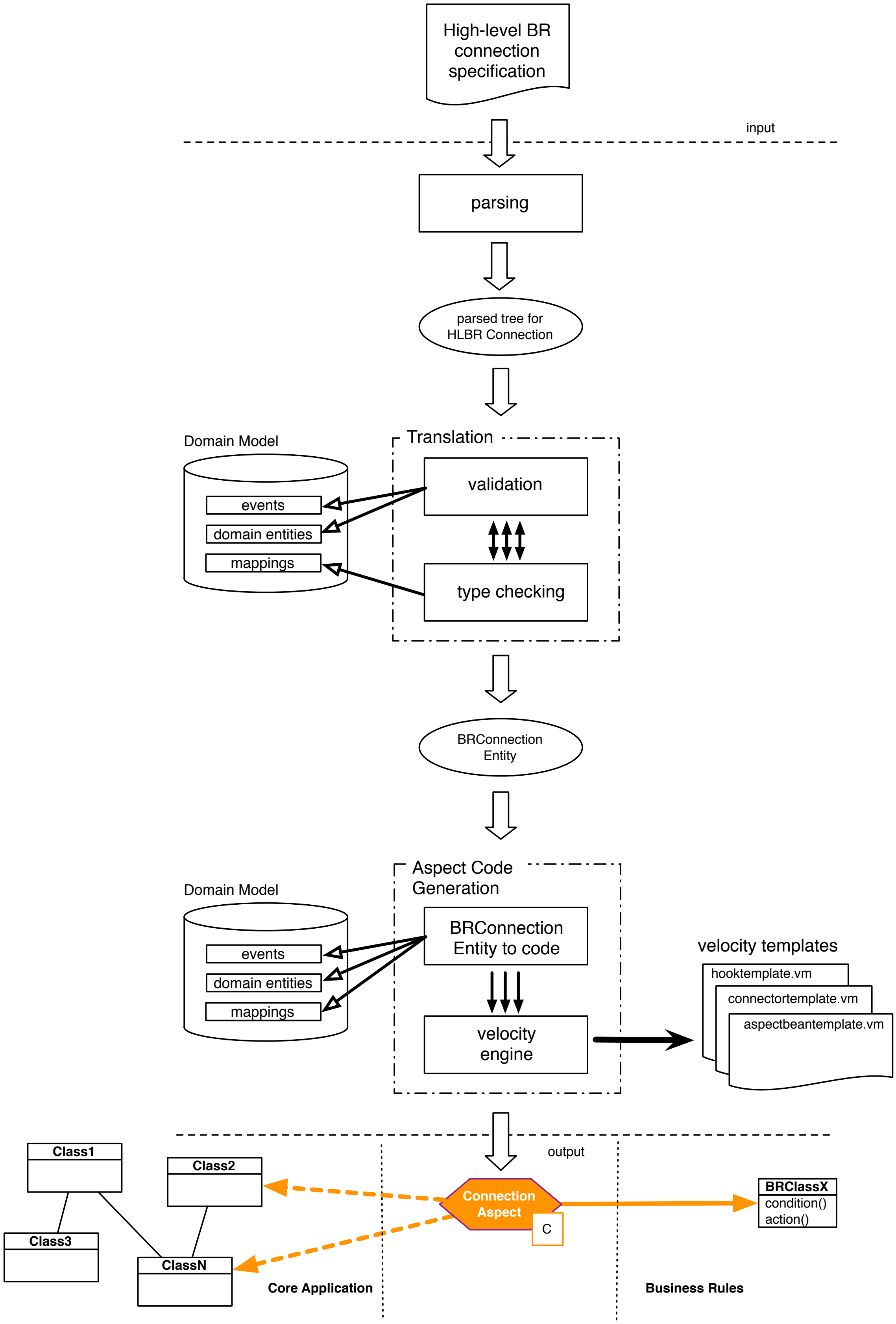
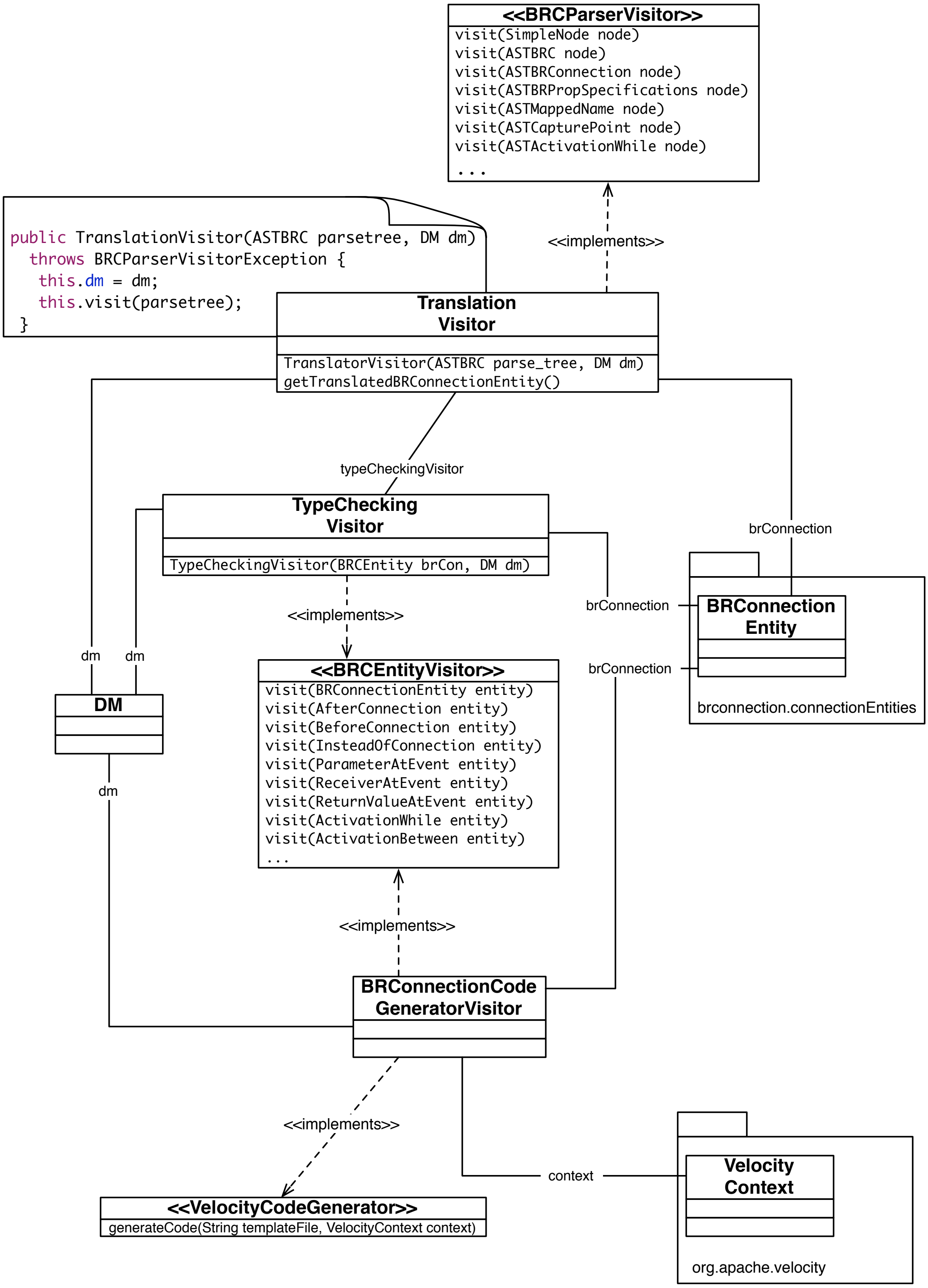


*Figure 7.14:* Overview of the transformation process of a high-level business rule connection



```
public TranslationVisitor(ASTBRC parsetree, DM dm)
  throws BRCParserVisitorException {
   this.dm = dm;
   this.visit(parsetree);
 }
```

<<BRCParserVisitor>>
visit(SimpleNode node)
visit(ASTBRC node)
visit(ASTBRConnection node)
visit(ASTBRPropSpecifications node)
visit(ASTMappedName node)
visit(ASTCapturePoint node)
visit(ASTActivationWhile node)
...
<<implements>>
Translation Visitor
TranslatorVisitor(ASTBRC parse_tree, DM dm)
getTranslatedBRConnectionEntity()
typeCheckingVisitor
TypeChecking Visitor
TypeCheckingVisitor(BRCEntity brCon, DM dm)
brConnection
brConnection
brConnection
BRConnection Entity
brconnection.connectionEntities
<<implements>>
dm
dm
DM
dm
<<BRCEntityVisitor>>
visit(BRConnectionEntity entity)
visit(AfterConnection entity)
visit(BeforeConnection entity)
visit(InsteadOfConnection entity)
visit(ParameterAtEvent entity)
visit(ReceiverAtEvent entity)
visit(ReturnValueAtEvent entity)
visit(ActivationWhile entity)
visit(ActivationBetween entity)
...
<<implements>>
BRConnectionCode GeneratorVisitor
<<implements>>
context
Velocity Context
org.apache.velocity
<<VelocityCodeGenerator>>
generateCode(String templateFile, VelocityContext context)

*Figure 7.15:* Main classes implementing the translation phase in the transformation process of a high-level business rule connection

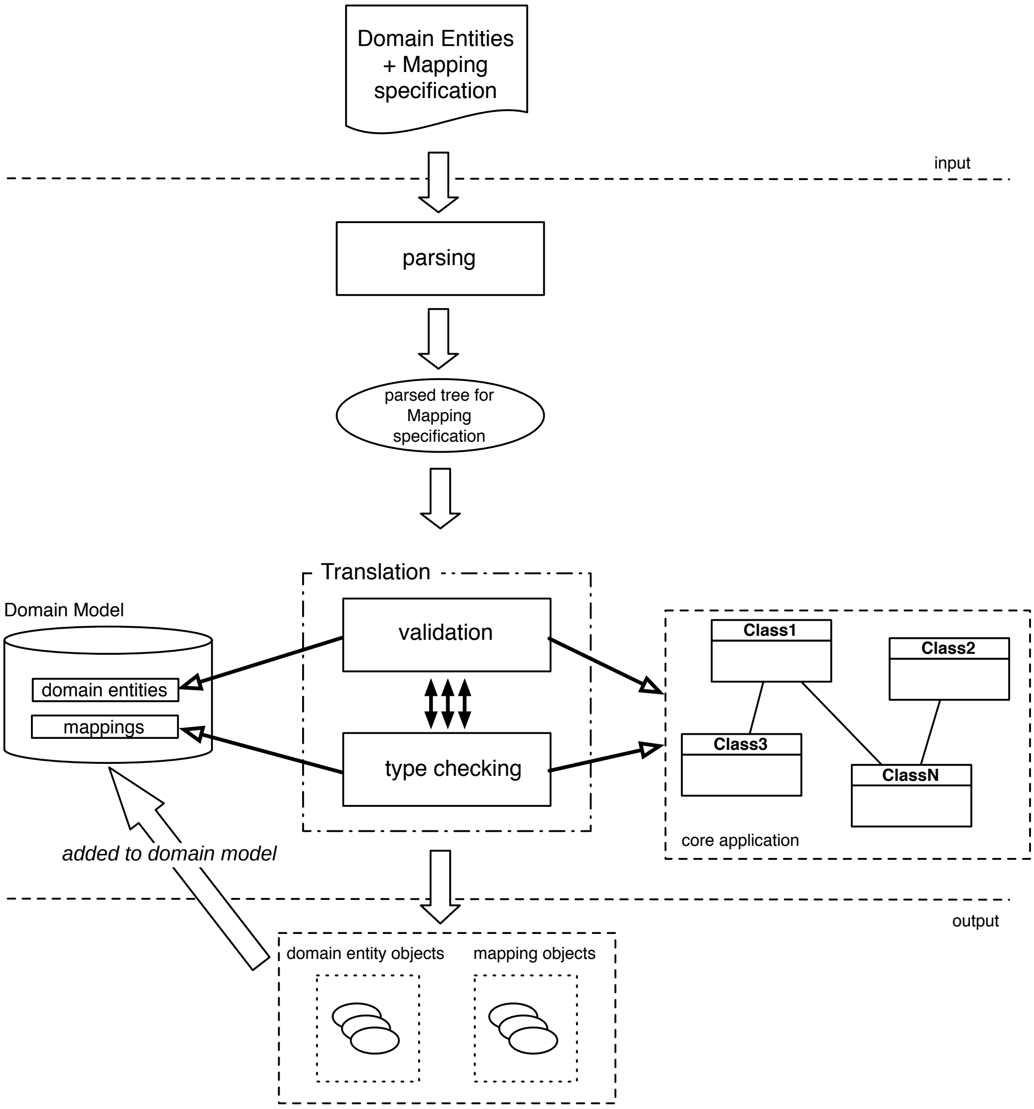


*Figure 7.16:* Overview of the transformation process from the mapping specification of a domain entity

# Chapter 8

# Evaluation

In this chapter an evaluation of our approach is presented which uses a case study in the domain of Service-Oriented Architectures (SOA), the Web-Services Management Layer (WSML). Unlike the real-world domains (e.g. financial, medical) typically found in state-of-the-art business rules systems, the chosen case study is based on a technical and challenging application domain, which let us show the expressive power of our approach. The same way as real-world domains, this domain suffers from the problems of the decoupling of business rules and therefore can benefit from our approach. Many rules need to be taken into account in order to cope with the inherent volatility of service-oriented applications. We particularly focus on QoS criteria that guide the selection and client-side management of Web services. In this chapter, an overview of the case study is presented in section 8.1. Then the motivation behind the WSML is presented in section 8.2, as well as a description of its general architecture (section 8.2.1), the way it supports selection, management and redirection (section 8.2.2) as well as its current limitations with respect to changing and adding new configuration business rules based on QoS (section 8.2.3). We then present two scenarios in sections 8.3 and 8.4, evolution and refactoring scenarios where we show, on the one hand, how our approach can realize the non-invasive addition of unanticipated business rules to the WSML and, on the other hand, how existing selection policies can be refactored and expressed at the high-level. We conclude in section 8.5 with a discussion.

## 8.1 Case Study: Web services Management Layer

In service-oriented computing, applications are often created by integrating third-party Web services. However, in order for client applications to achieve a high flexibility in this integration, advanced support for selection and client-side service management is fundamental. This support is rarely provided in standard state-of-the-art service integration approaches and tools. Moreover, we observe that the selection, integration and management of Web services are driven by criteria based on non-functional service properties. For instance, the service integration can be guided by rules that prefer fast and reliable services or give priority to services with the least number of failures; other rules govern the way management should be carried out, e.g. advising the activation of a caching mechanism for services that are too slow. Many of these business rules depend on dynamic service properties that are only known at run time. The explicit specification of these business rules is crucial to achieve a highly flexible integration of services that best fit the client application's needs.

As a first step towards achieving these goals, the Web service Management Layer (WSML) was proposed in [CVV$^+$07; VC05; VCJ04; VCV$^+$04; VC04; VCS$^+$04; CVJ03; VCJ03;

CV03]. The WSML is an AOP-based management framework that allows for the dynamic selection and integration of services in client-applications and the client-side service management. To this end, the WSML offers a reusable library of selection, management and monitoring concerns implemented as aspects that can be customized for different applications on which this layer is deployed. Although the WSML significantly enhances the overall service management, some limitations are observed with it.

Many decisions about how the framework needs to be configured and customized are taken manually either at deployment time, i.e. at the moment the WSML framework is deployed on a concrete client application, or at run time, i.e. by using the WSML administration interface. In any case, human interaction is required. Examples of these configuration decisions are: choosing which selection, management and monitoring concerns to plug in or out and under which conditions, which parameters to use for the configuration of such concerns, which concrete Web services to use for a given composition, etc. Because the WSML is developed having flexibility and adaptability in mind, the WSML implementation foresees the fact that the selection and the management can be guided by conditions based on QoS. These conditions are implemented as *isApplicable* methods in JAsCo aspects that carry out the corresponding selection and management tasks. For instance, the caching functionality is performed only when the average speed of the service that is being addressed falls under a threshold, whereas a certain Web service is approved for redirecting functional requests to it only when it is not expensive. Although these conditions allow the conditional execution of selection and management tasks, the added flexibility is limited, as analyzed and discussed in section 8.2.3. We observe that the presented limitations impede achieving a highly configurable and flexible adaptation, customization and configuration of the WSML framework.

This chapter shows how our approach can be used to express and enforce dynamic business rules that guide the adaptation, customization and configuration of the WSML. First of all, an *evolution scenario* is presented in section 8.3, showing that it is possible to extend the current WSML functionality in order to cope with new unanticipated business rules. This demonstrates that domain evolution is supported. More concretely, this scenario shows that:

- those unanticipated rules can be expressed externally and enforced non-invasively in the existing WSML, without having to change or insert code.
- the WSML becomes more evolvable as its implementation can be extended as a result of the consideration of new business rules and new service-oriented concepts.

Moreover, a second scenario is presented in section 8.4 that focuses on *refactoring.* The goal of this section is to show how the domain approach can be used to refactor the implementation of the WSML. The existing selection policies that are anticipated in the WSML can be externalized as high-level business rules. This scenario shows that:

- it becomes possible to change the conditions that guide the selection of services, even at run time, without having to change or extend the WSML implementation manually.
- it is easier to reason about those selection policies when they are expressed in terms of the domain.
- the WSML code becomes more understandable and maintainable.

## 8.2 Web-Services Management Layer (WSML)

Web services are modular applications that are described, published, localized and invoked over a network. The aim of the Web service technology is to facilitate the integration of different business processes regardless of the software and hardware used underneath. It offers a platform-independent solution to wire distributed applications of different enterprises. In the relatively short time that Web services have been around, a wide range of supporting tools have been developed that enable the creation and deployment of Web services and the development of service-oriented applications. Key technologies built around W3C standards such as the Simple Object Access Protocol (SOAP) [GH$^+$03], Web services Description Language (WSDL) [CC$^+$03] and Universal Description, Discovery and Integration (UDDI) [BC$^+$04] make it possible to publish, look up and consume services in a straightforward manner. Throughout this chapter, the terms service refers to either a single Web service or a Web service composition. Note that a Web service composition refers to the combination of many Web services which together are able to satisfy certain functional requests. Typically a service composition is described using a service composition language (such as BPEL [ACD$^+$03]).

Although an impressive range of development tools enable the creation, deployment and management of Web services on the server side, the just-in-time discovery and integration of Web services on the client side is still an issue. Client applications that integrate services using current technologies are rather inflexible because they cannot flexibly adapt to changes in the very volatile service environment (e.g. a service is abandoned or changed, a new service becomes available on the market or a service fails due to network problems). This constitutes a first limitation.

A second limitation is that Web services can only be selected based on the functionality they offer. The Web service documentation provided in WSDL format does not support the explicit specification of non-functional requirements such as constraints based on Quality of Service, classes of service, access rights, pricing information, Service-Level Agreements (SLAs), etc. The explicit specification of these non-functional properties on the server side, in a precise and unambiguous manner, allows for a more intelligent and customized selection and integration of services. This way, applications can base their decisions on business requirements when they integrate the services that best fit their needs. The WSML does not focus on improving service documentation, but rather on how selection policies based on these non-functional properties can be enforced in the client application. We also tackle the monitoring of properties that measure the way Web services behave at runtime. Examples of such properties are actual uptime, response time and number of failures. These monitored properties are taken into account by selection policies (as explained in section 8.2.2.1).

A third obstacle is that Web services are typically integrated by hardcoding proxy classes on the client side. As a consequence of treating Web services as internal software components, their specific requirements are completely ignored: Web services are organizationally fragmented, can be asynchronous and latent, can become unavailable due to unpredictable network conditions, and thus require more management [Szy05]. To deal with these issues, additional code has to be included manually in the client application. Even if this code is encapsulated in a separate reusable module, its execution has to be triggered repeatedly from the different points in the application where Web service functionality is required. As a result, management code is duplicated and scattered all over the application, which makes maintenance more difficult.

To overcome these limitations, a management layer between the application and the Web services is proposed in [CVV$^+$07; VCJ04; VCV$^+$04]: the Web services Management Layer (WSML). This intermediate layer allows for the dynamic selection and integration of services into an application, client-side service management, and support for service criteria based on the non-functional properties that govern the selection, integration and composition of Web services. The WSML is implemented using JAsCo since it realizes the dynamic addition and removal of aspects at run time, which is a crucial requirement in volatile service-oriented environments.

The benefits introduced by the WSML are:

- The application becomes more flexible, because it can continuously adapt to the changing business environment and communicate with new Web services that were unknown or unavailable at deployment time.
- By weakening the link between the application and the Web services, the hot-swapping of services becomes possible. When a service becomes unreachable due to network conditions or service-related problems, another equivalent Web service can be used instead.
- Replacing invocations of specific Web services by a generic way of requesting service functionality, and extracting all Web service selection and management code from the client applications, facilitate code maintenance and adaptability.

### 8.2.1 Architecture of the WSML

Figure 8.1 illustrates the architecture of the WSML. JAsCo aspects are used for implementing the generic functionality of the management layer. JAsCo connectors specify where these aspects need to be deployed. On the left-hand side of Figure 8.1, an application requesting Web service functionality is shown. To enable this application to make requests without referencing concrete services, the concept of *service type* is introduced. A *service type* is a generic description of the service functionality required by a client application. It is completely independent of concrete Web services and thus contains no references to concrete services. A service type can be seen as a contract between the application and the services. It hides the syntactical differences between semantically equivalent services. Concrete Web services that comply with the same service type can differ in many ways, for instance:

- Web method names;
- synchronous / asynchronous methods;
- parameter types and return types;
- semantics of parameters and return values; and
- method call sequencing.

The concept of service type makes it possible to hide the heterogeneity of the underlying concrete Web services.

The right-hand side of Figure 8.1 shows three semantically equivalent services that are available to fulfill the request for a particular service type, meaning that they offer the same functionality but possibly differ syntactically in the way they provide it (e.g., services using different method names or a different number of parameters). A mechanism based on redirection aspects enables requests to be redirected or services to be hot-swapped. Additional selection policies can be encapsulated in selection aspects. Finally, management aspects are used to deal with management concerns such as monitoring, caching and billing.

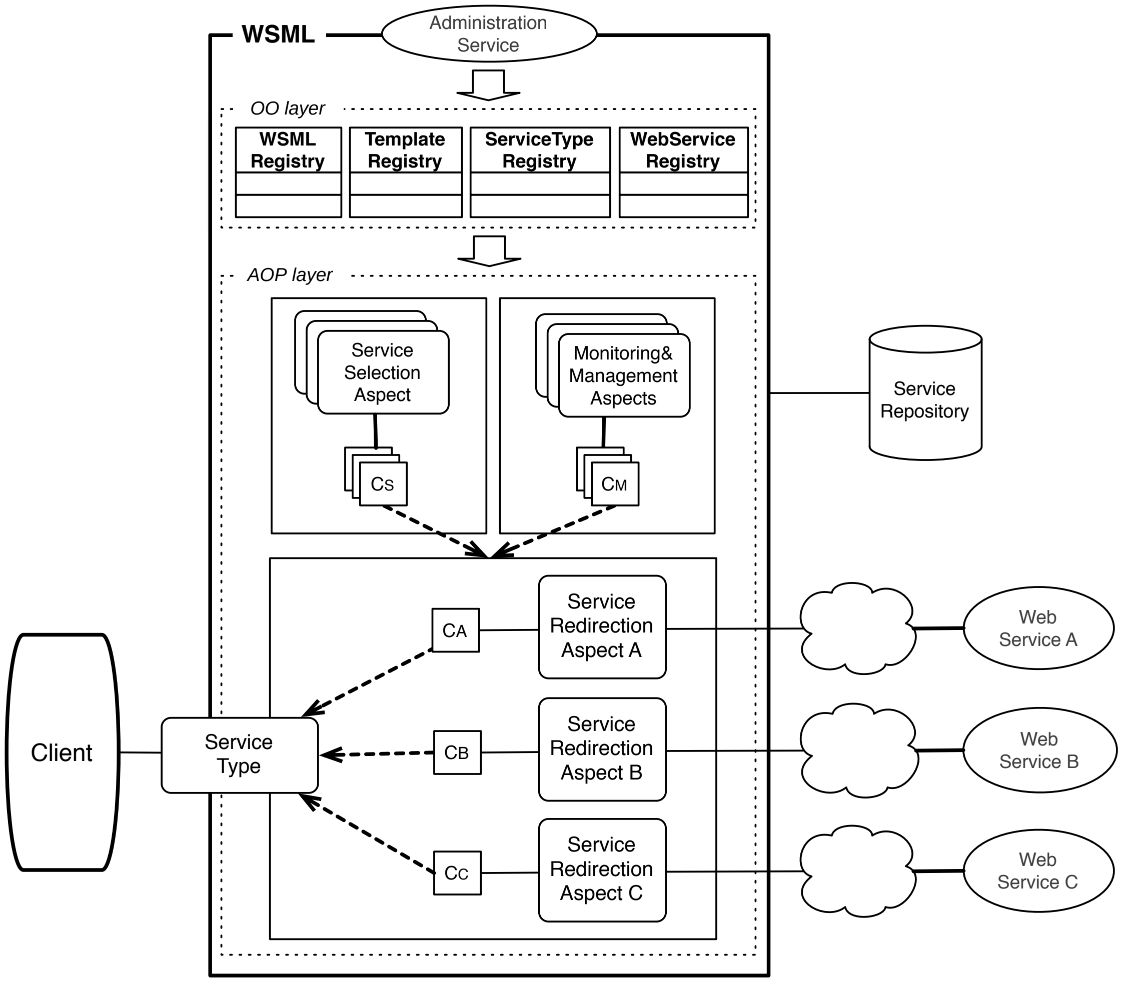


*Figure 8.1:* Architecture of the `WSML` (adapted from [Ver06])

On top of the aspect layer, a core OO layer is provided which offers the functionality to manage and configure the concrete aspects that exist in a concrete deployment of the WSML framework. The two coexisting layers in the WSML are depicted in Figure 8.1. The `WSMLRegistry` is the main class in the OO layer. This class maintains two references to the `TemplateRegistry` class under the names of *selectionModule* and *managementModule*. The `TemplateRegistry` class provides methods to add, remove and obtain aspect templates and concrete instances of those aspect templates as well as it allows for enabling, disabling and configuring a given aspect instance — represented as an instance of `TemplateInstance`. All operations that have an effect on the aspect layer are carried out through this core OO layer. Also, references to the `ServiceTypeRegistry` and `WebServiceRegistry` classes are kept in

`WSMLRegistry` which are in charge of maintaining the list of available service types defined in the system and concrete Web services registered with the WSML, as well as defining methods for their manipulation.

### 8.2.2 Selection, Management and Redirection in the WSML

In this section, the current WSML support for selection, management and redirection are described in detail. The current implementation of the WSML allows guiding the selection, the monitoring and management of services with conditions based on QoS.

#### 8.2.2.1 Web Service Selection

A limitation of the Web services Description Language (WSDL), the standard for the specification of the functionality of Web services [CC$^{+}$03] , is that it does not natively support the specification of non-functional properties. Several WSDL extensions have been proposed to enable the specification of non-functional properties on Web services (e.g., the Web service Offering Language [TPE$^{+}$02]). Selection policies can then be based on these non-functional service descriptions (e.g. they can pertain to cost). Another category of service properties includes average response time, number of successful invocations, network bandwidth and service reliability. These properties depend on the runtime behavior of the service itself and need to be monitored during the execution of the application. Current Web service integration approaches, however, provide little or no support for such non-functional properties in the service selection process.

The WSML improves on current approaches by supporting selection policies based on non-functional properties of services which are able to select the most appropriate service for a given functional request. These policies encapsulate business rules that filter and/or order available Web services according to values of their non-functional properties. They are triggered after the completion of a functional request invoked on a service type, which is ultimately delegated on a concrete Web service registered for that service type. As a result of applying a selection policy, the new ordered or filtered list of available services for that service type is taken into account in the next functional request performed on that same service type. Selection policies are made *explicit* in the WSML layer in the form of aspects.

Two kinds of selection policies based on QoS properties are supported in the WSML:

- *imperatives:* an imperative is a constraint on a service that must be satisfied. Imperatives describe mandatory conditions (e.g., the cost of the service must not exceed a fixed amount, or the response time of a service must not go beyond a certain threshold). They can also involve interrelationships with other services or the system (e.g. a service needs to be less expensive than the average cost of all registered services). In order for a Web service to be approved for selection and integration, it must comply with all the specified imperative selection policies. When a service does not satisfy an imperative selection policy, it gets disapproved and no longer considered for selection. The disapproved service has the chance to get approved again when it changes its state (e.g. its properties are updated).

- *guidelines:* from the list of filtered services that satisfy all the imperatives, some have to be given priority over others. This is where guidelines come into play. For example, one can give priority to the least expensive service, or the service with the highest security level. This implies that services are compared with each other and ranked. Note that if an approved service does not satisfy a specific guideline (e.g., it is not the least expensive at a given moment in time), it can still be considered for selection later on (e.g., if its cost goes down or if less expensive services fail).

Selection policies require the monitoring of non-functional properties to be up and running. In the WSML, monitoring is implemented in aspects which observe the behavior of the Web services and keep the results of their observations as service properties — stored in the core management framework. A schema depicting the use of selection and monitoring aspects in the WSML is shown in Figure 8.2.

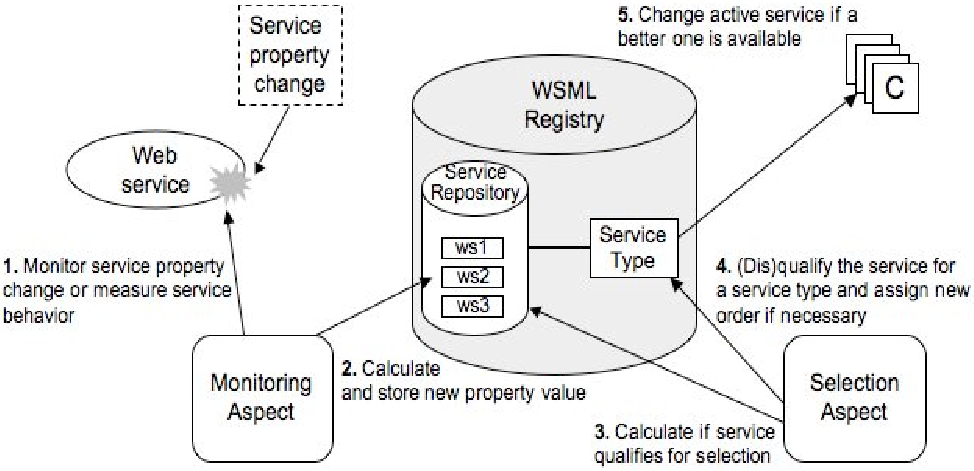

Service property change
5. Change active service if a better one is available
C
WSML Registry
Web service
Service Repository
ws1
ws2
ws3
Service Type
1. Monitor service property change or measure service behavior
4. (Dis)qualify the service for a service type and assign new order if necessary
Monitoring Aspect
2. Calculate and store new property value
3. Calculate if service qualifies for selection
Selection Aspect

*Figure 8.2:* Service monitoring and service selection aspects in WSML (taken from [CVV$^+$07])

For instance, if a selection policy specifies that the service with the shortest response time must be selected, the performance of all involved services needs to be observed by deploying a monitoring aspect at specific points within the application. The monitoring aspect supported in the WSML implementation and illustrated in Figure 8.2 can monitor changes in the property values of a service (step 1). When the value of a service property changes, the new value is computed and stored (step 2). Whenever this occurs, the imperative selection aspects are triggered to check whether the service that changed still satisfies the specified constraints (steps 3 and 4). Finally, the guideline selection aspects select the most appropriate Web service (step 5).

WSML provides a library of reusable and generic selection policy templates that can be used to approve a set of Web services. For example, the *ServicePropertyImperative* selection policy aspect is provided which can be instantiated and initialized with different parameters at runtime by deploying a new connector. For instance, when Web services are filtered according to their average response times, the *ServicePropertySelection* aspect is initialized in a connector using the following values:

- serviceType = <<*the name of the concrete service type*>>
- minimum = 0
- maximum = 5000

- property = “Average speed”

Note that changing the values of these selection policy parameters is the only flexibility supported by the WSML for guiding the service selection.

In addition to imperative selection policies, this library also contains a guideline selection policy named *ServicePropertyGuideline* that ranks registered services according to their property values (e.g., to select the cheapest service first). To actually rank the Web services for a particular service type, the connectors of their corresponding redirection aspects must also be reordered. This is achieved using JAsCo connector combination strategies, which are used to control the execution sequence of connectors. The chaining of the around behaviors of the redirection aspects can thus be altered in a straightforward and flexible way.

#### 8.2.2.2 Client-Side Web Service Management

The WSML is also able to deal with management concerns that need to be controlled on the client side of the application. To this end, a library of management aspects is provided as part of the framework, including support for caching, billing, fallback strategies, etc. In this section we consider the *caching* management concern as a representative example of management concerns supported in the WSML. By implementing caching, the number of remote Web calls to a service can be reduced and results can be provided to a client even if no services are currently available for fulfilling a particular request. The caching functionality works in cooperation with the basic redirection mechanism (explained in section 8.2.2.3).

As introduced in section 8.1, the execution of the management concerns can be controlled by conditions that are checked right before the actual management task is performed. This extra support is provided by subaspects that inherit from the aspects implementing the management functionality itself. In addition add a condition in the firm of an *isApplicable* method. This condition is based on non-functional QoS properties of services. For example, the *ConditionalCaching* aspect, which inherits from the more generic *Caching* aspect, first checks whether the value of a certain property (e.g. the service price) of the service that is being invoked is smaller than a threshold, in which case an around advice is executed which first proceeds with the interrupted service invocation and then stores the value returned by that invocation in a cache. Once the caching functionality is installed, the following requests performed on the same services for which values exists in the cache are addressed by directly returning those cached values instead of actually proceeding with the service invocations. As soon as the value of the QoS property (i.e. price in the example) becomes greater than the specified threshold, the caching aspect stops storing the results in the cache, which causes the services to be invoked again.

A predefined library of reusable management aspects is offered by the WSML framework. Instances of these aspects can be added and removed at run time through the OO management layer of the WSML. Management aspects can be configured to operate at different levels: at the level of service types, operating at the moment a service type is invoked, or at the level of concrete Web services, operating at the moment a concrete Web service is addressed and the level of a service composition, i.e. when a functional request is redirected to a service composition.

#### 8.2.2.3 Web Service Redirection

When using the WSML as an intermediate layer in between the services and the client applications, the latter are unaware of the details of specific services by invoking the desired functionality on generic service types. To translate these generic requests into specific Web service invocations, the WSML implements a mechanism based on service redirection aspects. A redirection aspect encapsulates all communication details for a specific Web service or service composition. A different redirection aspect is associated with each concrete Web service or service composition integrated in the system. Each service redirection aspect specifies which behavior to execute, i.e. which concrete methods to invoke on a specific Web service or service composition when a generic service functionality is requested by the client application. The details of how the different Web services involved in a service composition interact with each other in order to satisfy a functional request are encapsulated in a redirection aspect.

The basic redirection mechanism works as follows (see Figure 8.3). As soon as a client requests a service type, the first redirection aspect is executed and the corresponding Web service is invoked. If this invocation fails, an exception is thrown which is caught by a fallback aspect. Different fallback strategies can be implemented by this aspect. The default fallback strategy attempts to invoke the next available Web service, and so on until all available services are tried. If none of these invocations succeed, an exception is thrown back to the client. This whole process is transparent to the client application. As JAsCo allows aspects and connectors to be deployed at runtime, new services can be integrated dynamically. As a result, WSML can easily cope with changes in the Web service environment. When a new concrete Web service is registered in WSML for a given service type, a new redirection aspect is generated and added to the system. By creating a corresponding connector at runtime, the new service can be integrated with the client application. The current version of WSML supports fully automated connector generation and tools for creating redirection aspects.

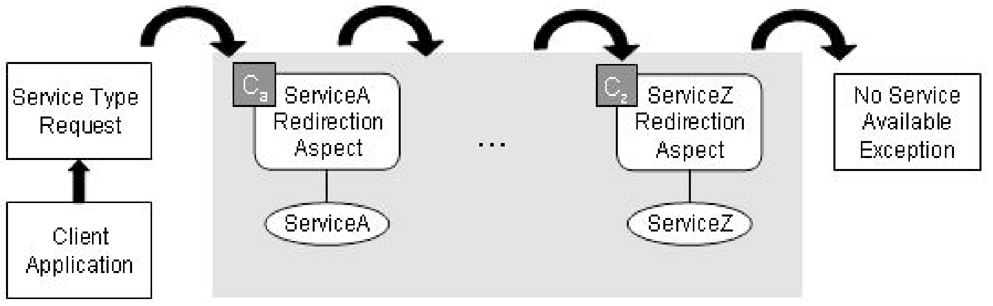

Service Type Request
Client Application
Ca
ServiceA Redirection Aspect
ServiceA
...
Cz
ServiceZ Redirection Aspect
ServiceZ
No Service Available Exception

*Figure 8.3:* Basic redirection mechanism implemented in the WSML (adapted from [CVV+07])

Similarly to the selection and the management, the redirection of functional requests to concrete Web services is also guided by hardcoded conditions. These conditions are based on the parameters of the request and thus are checked right before the redirection to a concrete Web service is performed.

### 8.2.3 Limitations of the WSML

The main limitation observed in the current WSML is in relation with how selection and management concerns are plugged in and out in a given application on which the WSML is deployed: enabling, disabling, configuring and removing monitoring, selection and management concerns that are currently supported in the WSML has to be done manually with human interaction. For example, the indication that a certain concern needs to be enabled or disabled can either be given in an XML configuration file — i.e. at deployment time — or via an administration console — i.e. at run time. In both cases, the decision of when and under which conditions this enabling or disabling action has to be carried out is taken manually, e.g. the WSML administrator in charge of configuring the layer.

Although some support is provided in the current WSML to achieve a more flexible pluggability of concerns by means of *'conditional aspects'* — aspects that are only triggered when a certain condition is met — the current support is limited:

- selection, management and redirection concerns can only be guided by conditions that are anticipated in the implementation of the conditional aspects realizing those concerns. These conditions check the values of QoS properties. For instance, current service selection policies are able to filter or order services only according to whether the value of a certain QoS monitored property falls into a range of expected values. Guiding these selection, management and redirection tasks with new conditions that were not anticipated in the current WSML implementation requires adding new aspects to the framework.
- not only are those conditions anticipated in the layer but also the information used in their definition is of course anticipated. Existing conditions in the WSML can only check an anticipated QoS property that can either be retrieved from the services themselves (and therefore are provided in the service documentation or API) or calculated by the WSML by means of monitoring the behavior of the services (in which case they are anticipated in the WSML). Selecting services, carrying out a management task or redirecting to a Web service based on QoS properties which were not anticipated is currently not possible. To add support for unanticipated QoS properties, the WSML needs to be extended.

In order to overcome these limitation we could of course extend the implementation of this management framework manually. For instance, new aspects implementing new selection policies realizing different filtering and/or ordering strategies (e.g. a round-robin selection strategy) could be added to the WSML as well as new monitoring aspects can be added to support unanticipated monitored properties. Because the WSML has been designed as an AOP framework, it is perfectly possible to plug-in and out new aspects, this way extending the framework's functionality. However, following such an approach has the following disadvantages:

- changing the conditions that guide the service selection and management implies invasive changes to the implementation of the current aspects or manually writing and adding new selection or management aspects.
- adding support for unanticipated QoS properties implies manually changing the implementation of the current monitoring aspects provided by the WSML or manually writing and adding new aspects that monitor service execution at different monitoring points than the ones that were anticipated.

Manually changing, writing and adding aspects is hard and error-prone since it involves getting acquainted with the details of the WSML implementation and requires having technical skills.

## 8.3 Evolution Scenario: Supporting Unanticipated Business Rules

In order to overcome the limitations of the WSML, we propose the use of the domain model approach presented in this dissertation. The aim is to be able to express high-level business rules that can automate the customization of this management framework. This way, the high-level rule and connection languages can be seen as a domain-specific languages for the customization of the WSML layer.

In this section we focus on showing how the implementation of the WSML can evolve without having to introduce invasive changes or the implementation of new aspects. In section 8.3.1 examples of new business rules that we would like to incorporate in the framework are presented. As a first step towards the expression of these rules at the domain level, the domain vocabulary of interest is made explicit at the high level as domain entities of a domain model (section 8.3.2). Because these initial domain entities are embodied in the implementation of the WSML, anticipated mappings for them can be defined. The next step consists of using those domain entities in the definition of the example business rules (section 8.3.3).

We also consider the case where new rules might want to be expressed for which new domain entities need to be added to the initial domain model (section 8.3.4). We show examples of domain entities that represent vocabulary of interest that is also embodied in the WSML but that has not been pulled up in the initial domain model (section 8.3.4.1) as well as domain entities that require capturing values at execution points (section 8.3.4.2). A second part of the experiment focuses on evolution (section 8.3.4.3): we consider the case where new rules need to be expressed for which the domain vocabulary of interest is not embodied in the existing implementation of the WSML. Thus, new domain entities are added to the initial domain model which are unanticipated in the current implementation. We then show how the initial domain model can be extended with extra domain entities representing the vocabulary of interest and how the desired rules can be defined in terms of those added domain entities.

### 8.3.1 Identifying Potential Configuration Business Rules

The following customization actions, which are currently supported in the WSML, are considered:

- enable a monitoring, selection or management concern;
- disable a monitoring, selection or management concern;
- add and configure a monitoring, selection and management concern; and
- remove a monitoring, selection and management concern.

The addition of a concern implies the addition of a new aspect in the system whereas the enabling of a concern implies the enabling of an aspect already existing in the system. The automating of these customization actions could be achieved when their execution is controlled by business rules. The case of conditions based on QoS properties is considered. Different ways in which those QoS properties are obtained are identified:

- QoS properties which are documented in the services WSDL;
- QoS properties which are retrieved from services and stored in the WSML;
- QoS properties which are monitored by the WSML; and
- QoS properties which are unanticipated in the current WSML implementation:
  - need to be monitored at unanticipated points in the execution; and
  - need to be calculated from existing QoS properties;

### 8.3.2 An Initial Domain Model

The process of defining the domain entities and the business rules in terms of them can be carried out in two different ways. The first one is a *per phase* style, which consists of defining first all the domain entities that are needed for expressing all the desired business rules. The second one is an *incremental* style, where domain entities are defined incrementally as business rules need to be added to the domain model. Our approach supports the two styles. In this section the *per phase* style is followed, whereas in section 8.3.4 an *incremental* style is preferred.

Listings 8.1, 8.2, 8.3, 8.4, 8.5, 8.6 and 8.7 show the definition of domain entities and their mappings to the existing WSML. Note that because this is not a real-world domain but a technical Web services domain, the domain expert in charge of defining these domain entities needs to be knowledgeable of Web services terminology (e.g. Web services, number of failures, service selection, monitoring and management). As mentioned in section 8.2.1, two co-existing layers can be found in the WSML implementation: the OO and AO layers, in which the former allows for the manipulation of the latter. Thus, the mappings of the initial domain entities are defined in terms of existing implementation entities in the OO layer of the WSML[1].

Different ways to populate the domain model are illustrated. For example, domain entities can be extracted automatically from a subset of WSML implementation entities that are relevant from the domain perspective. In this case, one-to-one mappings are created to link these domain entities to their corresponding implementation entities. This is illustrated in line 1 of the high-level specification 8.2, which indicates the pulling up of the complete `WSMLProperty` class under the name *MonitorableServiceProperty* — including its attributes and methods and the types involved in their definitions. Note that the *MonitorableServiceProperty* domain class adds the definition of two domain operations *smallerThan(value)* and *greaterThan(value)* which have a high-level mapping defined in terms of other existing domain entities.

[1] When support for mapping to AOP entities that exists in the implementation is supported, domain entities could also map to the AO layer of the WSML. This is however outside the scope of this dissertation and subject of future work (as described in section 10.2).

Other domain entities only pull up part of an implementation entity. This is the case of the *WSML*, *MonitorableService*, *ServiceType*, *ServiceComposition* and *WebService* domain classes defined in the high-level specifications 8.3, 8.4, 8.5 and 8.6. For them, the MAP-TO-IMPL feature is used, which only creates a domain class for those OO classes but does not pull up their contained entities (attributes and methods). The next step is to define the needed domain properties and operations in those domain classes one by one. For example, the *WSML* domain class (shown in the high-level specification 8.1) defines domain operations *enableCaching* and *disableCaching* which represent the actions of enabling and disabling the caching concern. At implementation level, management concerns — and therefore the underlying aspects — are enabled and disabled via static methods defined in the class `TemplateRegistry`. As an association exists between the *WSMLRegistry* and the *TemplateRegistry* classes, the navigation `WSMLregistry.getManagementModule()` is included as part of the mapping specification in order to obtain the right instance of `TemplateRegistry` on which the methods `enableTemplateInstance` and `disableTemplateInstance` need to be invoked. Moreover, besides the definition of navigations in the implementation, these mappings involve the specification of literal values and nested arithmetical expressions. The mappings for the domain operations *addMonitoring*, *removeMonitoring*, *addGuidelineSelection* defined in the *WSML* domain class are defined in a similar way.

The *MonitorableService* domain class, the definition of which is shown in the high-level specification 8.3 maps to an interface, the `wsml.CommonServiceInterface` (the high-level specification 8.3). This domain class represents all services for which their QoS properties can be monitored (i.e. not only Web services but also service types and service compositions). It defines properties representing monitored QoS properties and operations to set the value of those properties. The example of the *averageSpeed* domain property is shown in the high-level specification 8.3 which maps to the invocation of the method `getProperty(String)` where the parameter represents the name of the property of interest, which in this case is `"Average Speed"`. All other monitored QoS properties — i.e. number of invocations, number of failures and speed of last Invocation — can be defined in an analogous way, and therefore they are not explained here. Note that this domain class also adds two domain operations *slowerThan* and *fasterThan* which map to domain expressions, defined in terms of the *averageSpeed* and the *smallerThan* and *greaterThan* domain entities declared in the *MonitorableServiceProperty* domain class.

Inheritance is illustrated as well in this initial domain model: the domain classes *ServiceType* (the high-level specification 8.4), *ServiceComposition* (the high-level specification 8.5) and *WebService* (the high-level specification 8.6) inherit from the domain class *MonitorableService.* This is indicated by means of the *INHERITS-FROM* feature. This implies that all domain entities defined in the *MonitorableService* domain class are inherited by those domain classes that inherit from it.

In addition to the monitored properties, Web services define QoS properties that are not dynamic and that can be obtained by either inspecting the documentation of the Web service or requesting the Web service. An example of such properties is the property *price*, as shown in the high-level specification 8.6. These properties can also be retrieved by means invoking the method *getProperty* passing the name of the property as parameter. The mapping for the property *price* (shown in line 6 of the high-level specification 8.6) is defined by the method *getProperty(String)* with the literal *"price"* as parameter. In a similar way, the domain operation *setPrice(price)* (shown in lines 8 to 9 of the same high-level specification) maps to the OO method `setProperty(String, String)` where the first parameter is fixed to the literal value *"price"* and the second one is linked to the variable `price` specified at the LHS of that mapping.

```
WSML MAP-TO-IMPL wsml.WSMLregistry

WSML {
 WSML.enableCaching(monitorableServiceName) MAP-TO-IMPL
  (wsml.WSMLregistry).getManagementModule().enableTemplateInstance(
  java.lang.String:"Caching" + monitorableServiceName, java.lang.String:"Caching")

 WSML.disableCaching(monitorableServiceName) MAP-TO-IMPL
  (wsml.WSMLregistry).getManagementModule().disableTemplateInstance(
  java.lang.String:"Caching" + monitorableServiceName, java.lang.String:"Caching")

 WSML.introduceMonitoringServiceComposition(sc) MAP-TO-IMPL
  (wsml.WSMLregistry).getManagementModule().addTemplateInstance(
  java.lang.String:"monitoring" + sc.name, java.lang.String:"InvocationMonitoring",
  java.lang.String[]:{"SC", sc.serviceTypeName, sc.name, "*"}, java.lang.String[]:null,
  boolean:false)

 WSML.introduceMonitoringServiceType(st) MAP-TO-IMPL
  (wsml.WSMLregistry).getManagementModule().addTemplateInstance(
  java.lang.String:"monitoring" + st.name, java.lang.String:"InvocationMonitoring",
  java.lang.String[]:{"ST", st.name, "*", "*"}, java.lang.String[]:null,
  boolean:false)

 WSML.stopMonitoring(service) MAP-TO-IMPL
  (wsml.WSMLregistry).getManagementModule().removeTemplateInstance(
  java.lang.String:"monitoring" + service.name, java.lang.String:"InvocationMonitoring")
}
```

*High-Level Specification 8.1:* Definition of the `WSML` domain class

```
MonitorableServiceProperty ALIAS-FOR wsml.properties.WSMLproperty

MonitorableServiceProperty {
  monitorableServiceProperty.smallerThan(value) MAP-TO-DOMAIN
      monitorableServiceProperty.getValue().compareTo(value) < 0

  monitorableServiceProperty.greaterThan(value) MAP-TO-DOMAIN
      monitorableServiceProperty.getValue().compareTo(value) > 0
}
```

*High-Level Specification 8.2:* Definition of the `MonitorableServiceProperty` domain class

The domain model allows defining events of interest at which rules might be applied. These definitions are completely specified at the domain level, in terms of domain entities. Listing 8.7 shows events that are defined as part of this initial domain model. An example is the *setAverageSpeedToWebServiceEvent* event which captures the moment the domain operation *setAverageSpeed(speed)* defined in the *WebService* domain class is executed and exposes the target Web service as *WebService* and the first parameter as *speed.* The other events are defined analogously.

```
MonitorableService MAP-TO-IMPL wsml.CommonServiceInterface

MonitorableService {

   service.name MAP-TO-IMPL service.getName()

   service.averageSpeed MAP-TO-IMPL service.getProperty(java.lang.String:"Average Speed")

   service.setAverageSpeed(speed) MAP-TO-IMPL
      service.setProperty(java.lang.String:"Average Speed", java.lang.Object:speed)

   service.slowerThan(value) MAP-TO-DOMAIN service.averageSpeed.smallerThan(value)

   service.fasterThan(value) MAP-TO-DOMAIN service.averageSpeed.greaterThan(value)

   service.numberOfFailures MAP-TO-IMPL
      service.getProperty(java.lang.String:"Invocation Failures")

   service.numberOfInvocations MAP-TO-IMPL
      service.getProperty(java.lang.String:"Invocations")
}
```

*High-Level Specification 8.3:* Definition of the `MonitorableService` domain class

```
ServiceType MAP-TO-IMPL wsml.ServiceType

ServiceType INHERITS-FROM MonitorableService
```

*High-Level Specification 8.4:* Definition of the ServiceType domain class

```
ServiceComposition MAP-TO-IMPL wsml.ServiceComposition

ServiceComposition INHERITS-FROM MonitorableService

ServiceComposition {
   serviceComposition.serviceTypeName MAP-TO-IMPL
      serviceComposition.getServiceType().getName()

   serviceComposition.price MAP-TO-IMPL
      serviceComposition.getProperty(java.lang.String:"price")

   serviceComposition.calculatePrice() MAP-TO-IMPL
      serviceComposition.calculateAutomaticProperty(java.lang.String:"price")
}
```

*High-Level Specification 8.5:* Definition of the `ServiceComposition` domain class

```
WebService MAP-TO-IMPL wsml.WebService

WebService INHERITS-FROM MonitorableService

WebService {
   WebService.price MAP-TO-IMPL WebService.getProperty(java.lang.String:"price")

   WebService.setPrice(price) MAP-TO-IMPL
      WebService.setProperty(java.lang.String:"price", java.lang.String:price)

   WebService.increasePrice(delta) MAP-TO-IMPL
      WebService.setPrice(Web.Service.price + delta)
}
```

*High-Level Specification 8.6:* Definition of the `WebService` domain class

```
EVENT setAverageSpeedToWebServiceEvent AT WebService.setAverageSpeed(speed)
EXPOSING TARGET AS WebService
       PARAMETER 0 AS speed

EVENT setAverageSpeedToServiceTypeEvent AT ServiceType.setAverageSpeed(speed)
EXPOSING TARGET AS serviceType
       PARAMETER 0 AS speed

EVENT calculatePriceOfServiceCompositionEvent AT ServiceComposition.calculatePrice()
EXPOSING TARGET AS serviceComposition
```

*High-Level Specification 8.7:* Definition of events

### 8.3.3 Business Rules in Terms of Initial Domain Entities

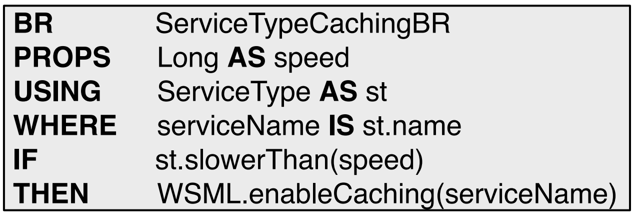
The focus of this section is to show how the high-level business rule and connection languages can be used to express configuration business rules. Examples involving the conditions and actions described in section 8.3.1 are given. The high-level rules and connections use domain entities defined in the initial domain model (Listings 8.1, 8.2, 8.3, 8.4, 8.5, 8.6 and 8.7).

#### 8.3.3.1 Enabling Service Type Caching Based on Average Speed

Consider the following example business rule:

***ServiceTypeCachingBR**: **if** a service type is slower than 1000 ms **then** enable the caching functionality for that service type*

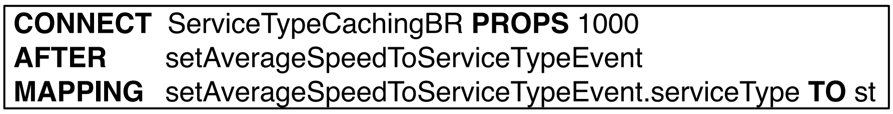
The concepts involved in this business rule are the concepts of *service type*, whether a service type is *slower than* the value 1000 and the concept of *enabling the caching* for a service type. Domain entities are present in the initial domain model which represent these concepts: the domain class *ServiceType*, the domain operation *slowerThan(value)* defined in the domain class *MonitorableService* and inherited by *ServiceType*, and the *enableCaching(name)* domain operation defined in the *WSML* domain class. Using these domain entities and the features of the high-level rule language, the *ServiceTypeCachingBR* can be expressed as shown in Figure 8.8. Note that this high-level rule is more generic than the rule expressed in natural language. This is because in order to improve reusability, the *ServiceTypeCachingBR* rule is defined as a rule template which defines the threshold *speed* as a rule parameter and not as a hardcoded value. This rule expects to receive a service type entity *st* at rule connection time, as it is specified in the *USING* clause. The *WHERE* clause defines a variable *serviceName* which keeps track of the name of the received service type. The condition is defined as the invocation of the domain operation *slowerThan(speed)* on the service type *st*. The action invokes the domain operation *enableCaching(serviceName)* on the domain class *WSML*.



```
BR        ServiceTypeCachingBR
PROPS     Long AS speed
USING     ServiceType AS st
WHERE     serviceName IS st.name
IF        st.slowerThan(speed)
THEN      WSML.enableCaching(serviceName)
```

*High-Level Specification 8.8:* The `ServiceTypeCachingBR` rule

Note that, although omitted here, an analogous rule must be defined in charge of disabling the caching concern when the service becomes faster than the threshold. In order to connect the *ServiceTypeCachingBR* rule to the WSML application, an event capturing the right application time must exist in the domain model. The decision of whether the caching has to be enabled is taken after the value of the average speed of that service type is changed, since it is then when one can determine that a service type is slow or fast. Thus, a suitable event on which to express this connection is *setAverageSpeedToServiceTypeEvent* (high-level specification 8.7). A possible high-level rule connection using this event is shown in Figure 8.9. This is an *AFTER* connection, as the rule has to be applied after the average speed of the service type is changed. The *setAverageSpeedToServiceTypeEvent* exposes the target service type which is mapped to the one expected by the rule in the *MAPPING* clause.



```
CONNECT  ServiceTypeCachingBR PROPS 1000
AFTER    setAverageSpeedToServiceTypeEvent
MAPPING  setAverageSpeedToServiceTypeEvent.serviceType TO st
```

*High-Level Specification 8.9:* Connection for the `ServiceTypeCachingBR` rule

As a result of defining, loading and translating these specifications, Java and JAsCo code is obtained for the rule and rule connection respectively, which is shown in the code fragment 8.1 for the rule object and code fragments 8.2 and 8.3 for the aspect and connection code[2].

#### 8.3.3.2 Adding Service Composition Monitoring Based on Price

Consider the following rule:

[2] Note that for simplification reasons we assume the aspect code shown in fragment 8.2 not to be executed simultaneously by multiple threads.

```
package Rules;

public class ServiceTypeCachingBR {

  final String name = "ServiceTypeCachingBR";

  //Properties
  java.lang.Long speed;

  //Business Objects
  wsml.ServiceType st;

  //Business Objects Attributes/Properties
  java.lang.String serviceName;

  //Constructor
  public ServiceTypeCachingBR(java.lang.Long speed)
  {
    this.speed = speed;
  }

  public wsml.ServiceType getSt() {
        return st;
  }

  public void setSt(wsml.ServiceType st) {
    this.st = st;
  }

 public String toString() {
    String st = this.name;
     st += " (";
     st += " Parameters: " + this.speed + " ";
     st += ";";
     st += " Business Objects: " + this.st + " ";
     st += ")"; return st;
    }

  //Fires the BR
  public void initializeRule()
  {
    //initialize bo attributes aliases
    this.serviceName = st.getName() ;
  }

  public boolean condition()
  {
    return st.getProperty("Average Speed").getValue().compareTo(speed) < 0;
  }

  public void action()
  {
    wsml.WSMLregistry.getManagementModule().enableTemplateInstance(
            "Caching" + serviceName, "Caching");
  }
}
```

*Code Fragment 8.1:* Generated rule object for ServiceTypeCachingBR rule

```
package Linking;
import Rules.ServiceTypeCachingBR;

class ServiceTypeCachingBRConnection {
 //BR Object initialization
 ServiceTypeCachingBR rule = new ServiceTypeCachingBR(new java.lang.Long(1000));

 //Attributes for Captured information
 wsml.ServiceType setAverageSpeedToServiceTypeEvent_serviceType;

 public String toString() {
   return ("Connection named " + "ServiceTypeCachingBRConnection" + " for rule " + rule);
}

 hook Hook0 {
   Object[] params;
   String methodName;

   //Constructor
   Hook0(method0(..args0)) { execution(method0) && target(wsml.ServiceType); }

   //Advice
   public refinable boolean mappingRestrictions();

   isApplicable() {
     params = thisJoinPoint.getArgumentsArray();
     methodName = thisJoinPoint.getName();
     return mappingRestrictions();
   }

   public String toString(){
     String st = "rule connection " + "ServiceTypeCachingBRConnection" +
                 + " hooked on method " + methodName + "(";
     for(int i=0; i<params.length; i++){
      st += params[i];
        if(i<params.length-1) st += ", ";
     }
     st += ")";
     return st;
   }

   after() {
     //associate local name to required connection info
     global.setAverageSpeedToServiceTypeEvent_serviceType = thisJoinPointObject;

     global.rule.setst(global.setAverageSpeedToServiceTypeEvent_serviceType);

     global.rule.initializeRule();
     //Rule conditional
     if(global.rule.condition()) {
       //Rule Triggering
       global.rule.action();
    }
    }
  }
}
```

*Code Fragment 8.2:* Generated aspect bean for `ServiceTypeCachingBRConnection`

```
static connector ServiceTypeCachingBRAtsetAverageSpeedToServiceTypeEventConnector{

  Linking.ServiceTypeCachingBRConnection.Hook0 hook0 =
    new Linking.ServiceTypeCachingBRConnection.Hook0(
      void wsml.CommonServiceInterface.setProperty+(java.lang.String, java.lang.Object)){
        public refinable boolean mappingRestrictions(){
          return thisJoinPoint.getArgumentsArray()[0].equals("Average Speed");
        }
    }
}
```

*Code Fragment 8.3:* Generated connector for the deployment of the `ServiceTypeCachingBRConnection` aspect

***AddMonitoringToServiceCompositionBR*: *if* *the price of a service composition is greater than 100* *then* *add monitoring functionality for that service composition***

The concepts involved in this business rule are *service composition*, *price* of a service composition and the concept of *adding monitoring* for a service composition. Domain entities are present in the initial domain model which represent these concepts: the domain class *ServiceComposition*, the domain property *price* defined in the domain class *ServiceComposition*, and the domain operations *greaterThan(value)* — defined in *MonitorableServiceProperty* — and *introduceMonitoringServiceComposition(sc)* — defined in the *WSML* domain class. Using these domain entities, the *AddMonitoringToServiceCompositionBR* can be defined at the high level as shown in the upper part of Figure 8.10. Analogous to the previous example, the *AddMonitoringToServiceCompositionBR* rule is defined as a rule template, defining the threshold *X* as a rule parameter. A service composition is expected by the rule, which is referred to as *sc* and is made available at rule connection time. The *WHERE* clause defines a local variable to refer to the price of the service composition received as parameter. The condition checks whether the price of the service composition is greater than *X*, in which case the action invokes the domain operation *introduceMonitoringServiceComposition* on the *WSML* domain class, passing as parameters the actual service composition to be monitored.

```
BR        AddMonitoringToServiceCompositionBR
PROPS     Integer AS x
USING     ServiceComposition AS sc
WHERE     price IS sc.price
IF        price.greaterThan(x)
THEN      WSML.startMonitoringServiceComposition(sc)
```

```
CONNECT   AddMonitoringToServiceCompositionBR
PROPS     100
AFTER     calculatePriceOfServiceCompositionEvent
MAPPING   calculatePriceOfServiceCompositionEvent.serviceComposition TO sc
```

*High-Level Specification 8.10:* The `AddMonitoringToServiceCompositionBR` rule and its high-level connection

This rule needs to be connected with the core application after the price of a service composition changes. The event *calculatePriceOfServiceCompositionEvent* captures the moment the domain operation *calculatePrice()* is invoked on a service composition (high-level specification 8.7). Using this event, a possible high-level rule connection is defined which is shown in the lower part of Figure 8.10. Again, an *AFTER* connection is needed, as the rule has to be applied after the price of the service composition changes. The *calculatePriceOfServiceCompositionEvent* exposes the target service composition which is mapped to the one expected by the rule in the *MAPPING* clause.

### 8.3.4 Adding New Business Rules

The existing domain entities in the initial domain model can be combined in different ways, giving rise to new business rules. Moreover, other business rules can be defined which require the initial domain model to be extended with the definition of extra domain entities.

#### 8.3.4.1 Anticipated Domain Entities

In this section we consider the case where these extra domain entities have a counterpart implementation entity in the existing WSML implementation. Low-level as well as high-level mappings are illustrated in this section. Consider the following example business rules:

***Rule A**: **if*** *service.isUnreliable()* ***then*** *WSML.enableCaching(service.name)*

***Rule B**: **if*** *service.fasterThan(X)* ***then*** *WSML.disableCaching(service.name)*

***Rule C**: **if*** *service.numberOfSuccessfulInvocations* > *X* ***then*** *WSML.stopMonitoring(service)*, where *service* is a monitorable service.

***Rule D**: **if*** *service.numberOfInvocations* < *X* ***then*** *WSML.enableCaching(service.name)*, where *service* is a Web service.

***Rule E**: **if*** *service.price* < *X* ***then*** *WSML.stopMonitoring(service)*, where *service* is a service composition.

***Rule F**: **if*** *service.numberOfFailures* < *X* ***then*** *WSML.orderServicesBasedOnInvocationFailures(service)*, where *service* is a service type.

For some of these rules, events used for the connection of other rules can be reused. For instance, *Rule B* can reuse the event at which *serviceTypeCachingBR*) is connected. Otherwise, new connection events can be defined for these new rules. Note that *Rule A* uses a domain operation *isUnreliable()* which is not defined in the initial domain model. However, this new entity can simply be added to the definition of the *MonitorableService*, as follows:

```
MonitorableService {
   ...
  service.isUnreliable() MAP-TO-DOMAIN service.numberOfFailures.greaterThan(10)
}
```

The mapping for the *isUnreliable()* domain operation is anticipated since it involves existing concepts. However, it is completely defined at the domain level, without having to point to implementation entities. Note that the value 10 is used in this mapping to show that literals are allowed in mapping specifications. Of course when needed, this hardcoded value can be replaced by a variable which must be included as a parameter of the domain operation in the $< LHS >$ of the mapping specification. In the same way, other anticipated domain concepts that require mappings that are defined similarly to the ones presented in this section can be defined. Existing domain entities can be combined in expressions to define new domain entities which then can be used in new rules. The high-level definition of the business rules presented in this section clearly suffices to change the behaviour of the WSML.

Another new concept used in *Rule F* is the concept of ordering the services associated to a given service type according to their number of invocation failures. In order to represent this new concept, a new domain operation is added to the *WSML* domain class, as follows:

```
WSML {
   ...
 WSML.orderServicesBasedOnInvocationFailures(st) MAP-TO-IMPL
  (wsml.WSMLregistry).getSelectionModule().addTemplateInstance(
  java.lang.String:"guideline"+ st.name + "InvocationFailures",
  java.lang.String:"ServicePropertyGuideline",
  java.lang.String[]:{"st.name", "*", "Invocation Failures", "n/a", "true", "true"},
  java.lang.String[]:null,
  boolean:false)
}
```

*Rule C* refers to a new domain property that defines the *number of successful invocations* of a service composition. This concept can be derived from two existing domain properties defined in the *MonitorableService* domain class: the *numberOfInvocations* and *numberOfFailures*. Thus, this new domain attribute can be mapped to an expression that calculates the substraction between these two existing domain properties, as shown in the fragment below. The translation of this high-level mapping to a concrete implementation (in terms of values stored in aspects) is automatic and transparent for the domain expert.

```
MonitorableService {
  ...
  service.numberOfSuccessfulInvocations MAP-TO-DOMAIN
              service.numberOfInvocations - service.numberOfFailures
}
```

#### 8.3.4.2 Calculating Values at Execution Points

In this section we consider the case where new domain entities are required which imply capturing values at different points in the execution of the WSML.

Consider the following example:

***if*** *service.downtime* > *X* ***then*** *WSML.enableCaching(service.name)*

This rule uses a *downtime* domain property which represents the amount of time a service is unavailable. This is a dynamic service property since it requires monitoring the time between the moment a service becomes unavailable and the moment it becomes available again. However, the current monitoring aspects in the WSML are not able to monitor this property as it requires different monitoring points and other logic than the ones anticipated in the implementation of the current monitoring aspect. Although the WSML can be extended by introducing a new monitoring aspect, this aspect has to be written manually. Our solution performs the automatic generation of the aspect that realizes this domain property. The input for this generation is a high-level mapping specification in terms of the *timeBetween* domain operator — predefined in the domain model infrastructure — which states that this dynamic property represents the time that elapsed between the moment the service becomes unavailable until the moment it becomes available again. This mapping is defined as follows:

```
WebService {
   ...
  service.becomeUnavailable() MAP-TO-IMPL service.becomeUnavailable()
  service.becomeAvailable() MAP-TO-IMPL service.becomeAvailable()
  service.downTime MAP-TO-DOMAIN
        timeBetween(WebService.becomeUnavailable(), WebService.becomeAvailable())
}
```

An aspect is transparently and automatically generated out of this mapping definition. Similarly to the *downtime*, one can define the *uptime* domain property representing the time between the moment the service becomes available until the moment it becomes unavailable.

This new domain vocabulary can be used in new business rules, for instance:

***if*** *service.downtime* > *service.uptime* ***then*** *enableCaching(service.name)*

#### 8.3.4.3 Unanticipated Domain Entities

Consider the *WebServiceSpeedCategoryBR* rule specified below which classifies services into categories *fast* and *slow* according to their average speed:

***WebServiceSpeedCategoryBR**: **if** a Web service is slower than 500 **then** the Web service is slow*

The condition of this rule can be defined using the *slowerThan(value)* domain operation defined in the *MonitorableService* domain class and inherited by the *WebService* domain class. However, the action talks about the concept of *speed category* which is unanticipated in the WSML implementation. In order to incorporate this concept in the existing domain model, a new domain property *speedCategory* can be added to the *MonitorableService* domain class, for which an unanticipated mapping is required, as shown below:

```
MonitorableService {
   ...
   service.speedCategory MAP-TO-VALUE String:"normal"
}
```

Because the logic under which to determine the values of the *speedCategory* property is driven by business decisions, the approach in this dissertation advocates encapsulating that logic in business rules. For example, the high-level rule *WebServiceSpeedCategoryBR*, shown in the upper part of Figure 8.11, sets the service category to *slow* when the average speed of that service falls under a *threshold*. The threshold is defined as a rule parameter in the PROPS clause. This rule can be triggered at the moment the domain operation *setAverageSpeed(speed)* is invoked on a Web service, which is captured by the *setAverageSpeedToWebServiceEvent* event defined in the high-level specification 8.7. Note that when the same domain operation is invoked on other services than Web services — i.e. service compositions and service types —, those invocations are not captured by the *setAverageSpeedToWebServiceEvent* event, and thus the rule is triggered only for the case of Web services. The *setAverageSpeedToWebServiceEvent* event exposes the target Web service as *WebService* and maps it to the Web service variable expected by the rule. The high-level connection for this rule can be defined as shown in the lower part of Figure 8.11. In a similar way, the rule *“**if** a Web service is faster than 500 **then** the Web service is* fast*”* can also be expressed in terms of the domain (omitted here).

```
BR        WebServiceSpeedCategoryBR
PROPS     Long AS speedThreshold
USING     WebService AS ws
IF        ws.slowerThan(speedThreshold)
THEN      ws.speedCategory IS "slow"
```

```
CONNECT   WebServiceSpeedCategoryBR PROPS 500
AFTER     setAverageSpeedToWebServiceEvent
MAPPING   setAverageSpeedToWebServiceEvent.webService TO ws
```

*High-Level Specification 8.11:* `WebServiceSpeedCategoryBR` business rule and its connection

Note than when the classification under the categories *fast* and *slow* needs to be applied to all monitorable services, the previous rules and connections need to be defined in terms of the *MonitorableService* class instead of *WebService*.

We can imagine other unanticipated Web services properties. For instance, services can be classified according to their price into *expensive* or *not expensive*, or according to their number of invocations into *frequently invoked* or *not frequently invoked*. Again, these new categories are represented as new domain properties in the domain classes that are being classified. In this case, new domain properties *expensive* and *frequentlyInvoked* are simply added to, for instance, the *WebService* domain class, as follows:

```
MonitorableService {
   ...
   service.expensive MAP-TO-VALUE boolean:false
   service.frequentlyInvoked MAP-TO-VALUE boolean:false
}
```

New business rules can then be written in order to determine under which conditions the value of these categories must change:

***if*** *service.price >  X* ***then*** *service IS expensive*

***if*** *service.price <  X* ***then*** *service IS NOT expensive*

***if*** *service.numberOfInvocations >  X* ***then*** *service IS frequentlyInvoked*

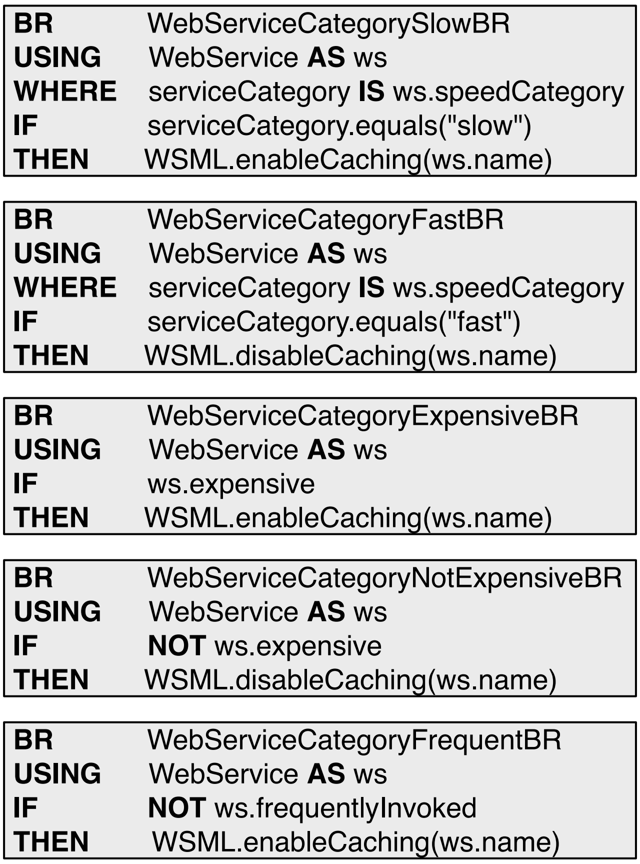
***if*** *service.numberOfInvocations <  X* ***then*** *service IS NOT frequentlyInvoked*

An alternative solution to defining new domain properties (i.e. the categories in the examples) and writing business rules for determining their value, is to repeat the same calculation logic in all places where those domain properties are needed. However, this solution causes redundancy of the same logic among several business rules that rely on those same domain properties, which is not desirable. Thus, the definition of domain properties is a more suitable solution. Also, one might wonder at this point why not to go for a solution that uses ordinary expressions in domain property mappings to define how the values of these service categories are calculated. This solution is possible when the way of calculating the values of these categories is not meant to change often. However, when the calculation changes frequently or when different ways of calculating these values are possible, a more suitable solution is to encapsulate the calculation in business rules (as shown above).

Once business rules have been defined to determine the values of these categories, other rules can be considered to trigger actions based on those values. Figure 8.12 shows some example rules that check service categories and react accordingly.

## 8.4 Refactoring Scenario: Externalizing Anticipated Selection Policies

In this section, a second thought experiment is carried out which consists of using the business rule dedicated languages for refactoring the existing WMSL layer implementation. It is observed that the existing selection policies *ServicePropertyGuideline* and *ServicePropertyImperative* are examples of anticipated business rules which respectively order and filter Web services according to the values of their monitored properties. The evolution scenario presented in section 8.3 shows how configuration business rules can be defined in order to guide the configuration of these existing policies, i.e. by specifying conditions that guide the addition or removal of a policy, by specifying which monitored property needs to be checked in the ordering or filtering process, or any other configuration decision of that kind. However, these configuration rules cannot completely override the conditions predefined in those existing selection policies, since they are hardcoded in the existing selection aspects which are part of the current WSML infrastructure.



```
BR       WebServiceCategorySlowBR
USING    WebService AS ws
WHERE    serviceCategory IS ws.speedCategory
IF       serviceCategory.equals("slow")
THEN     WSML.enableCaching(ws.name)
```

```
BR       WebServiceCategoryFastBR
USING    WebService AS ws
WHERE    serviceCategory IS ws.speedCategory
IF       serviceCategory.equals("fast")
THEN     WSML.disableCaching(ws.name)
```

```
BR       WebServiceCategoryExpensiveBR
USING    WebService AS ws
IF       ws.expensive
THEN     WSML.enableCaching(ws.name)
```

```
BR       WebServiceCategoryNotExpensiveBR
USING    WebService AS ws
IF       NOT ws.expensive
THEN     WSML.disableCaching(ws.name)
```

```
BR       WebServiceCategoryFrequentBR
USING    WebService AS ws
IF       NOT ws.frequentlyInvoked
THEN     WSML.enableCaching(ws.name)
```

*High-Level Specification 8.12:* Business rules that trigger actions according to the values of unanticipated Web service categories

In this section the goal is to externalize the selection policies existing in the WSML and express them at the domain level, i.e. as high-level business rules and connections expressed in the high-level dedicated languages. The aspects that result from the automatic translation of these high-level specifications would then replace the predefined selection aspects existing in the WSML today.

By expressing selection policies at the domain level, it becomes easier to reason about those selection concerns. Moreover, the WSML code becomes more understandable and maintainable. Also, besides being able to express the existing selection policies at the high level, new selection policies can be added by simply specifying new high-level rules that guide the service selection.

### 8.4.1 Extending the Initial Domain Model

As a preliminary step in the process of pulling up the existing selection policies, the domain model presented in section 8.3.2 has to be extended with the definition of new domain entities representing the concepts involved in service selection. These extensions are shown in the high-level specifications 8.13 and 8.14. They include domain operations for approving, disapproving and prioritizing services which are added to the *ServiceType* domain class as well as a domain operation for the addition of a service composition to a service type. These domain operations map to existing OO methods defined in the `ServiceType` class. Also, a high-level event is defined on the *addService* domain operation, which is going to be used in the expression of high-level connections in the coming sections.

```
ServiceType MAP-TO-IMPL wsml.ServiceType

ServiceType INHERITS-FROM MonitorableService

ServiceType {
 serviceType.prioritizeServicesAccordingToProperty(property)
  MAP-TO-IMPL serviceType.prioritize(Collection.sort(serviceType.getServiceCompositions(),
  new ServiceCompositionComparator(property, true)))

 serviceType.approve(service)
  MAP-TO-IMPL serviceType.approve(wsml.ServiceComposition:service)

 serviceType.disapprove(service)
  MAP-TO-IMPL serviceType.disapprove(wsml.ServiceComposition:service)

 serviceType.addServiceComposition(concreteService)
  MAP-TO-IMPL serviceType.addServiceComposition(wsml.ServiceComposition:concreteService)
}
```

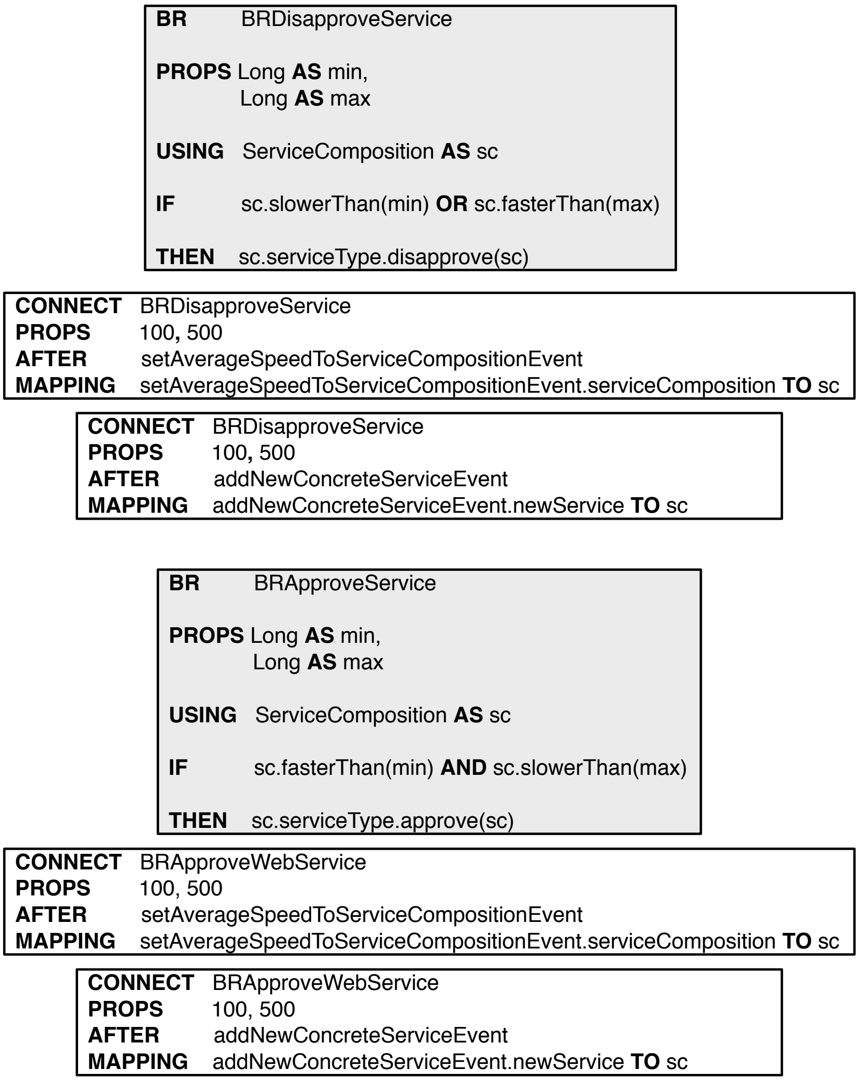
*High-Level Specification 8.13:* Extensions of the `ServiceType` domain class to express selection policies a the domain level

```
EVENT addNewServiceCompositionEvent AT ServiceType.addServiceComposition(sc)
EXPOSING PARAMETER AS newService
```

*High-Level Specification 8.14:* New event capturing the executing of the `addServiceComposition` domain operation

### 8.4.2 Expressing Selection Policies and their Connections at the Domain Level

The next step in this scenario is to provide high-level specifications for the selection policies and their connections in terms of domain concepts. Figure 8.15 shows the definition of the *BRDisapproveService* and *BRApproveService* rules and their connections in terms of the domain. These high-level specifications realize the imperative selection policy currently predefined in the WSML and implemented in the `ServicePropertyImperative` aspect. As a consequence of this experiment, the existing WSML implementation is refactored.



```
BR       BRDisapproveService

PROPS Long AS min,
         Long AS max

USING   ServiceComposition AS sc

IF        sc.slowerThan(min) OR sc.fasterThan(max)

THEN    sc.serviceType.disapprove(sc)
```

```
CONNECT   BRDisapproveService
PROPS       100, 500
AFTER        setAverageSpeedToServiceCompositionEvent
MAPPING    setAverageSpeedToServiceCompositionEvent.serviceComposition TO sc
```

```
CONNECT   BRDisapproveService
PROPS       100, 500
AFTER        addNewConcreteServiceEvent
MAPPING    addNewConcreteServiceEvent.newService TO sc
```

```
BR       BRApproveService

PROPS Long AS min,
         Long AS max

USING   ServiceComposition AS sc

IF        sc.fasterThan(min) AND sc.slowerThan(max)

THEN    sc.serviceType.approve(sc)
```

```
CONNECT   BRApproveWebService
PROPS       100, 500
AFTER        setAverageSpeedToServiceCompositionEvent
MAPPING    setAverageSpeedToServiceCompositionEvent.serviceComposition TO sc
```

```
CONNECT   BRApproveWebService
PROPS       100, 500
AFTER        addNewConcreteServiceEvent
MAPPING    addNewConcreteServiceEvent.newService TO sc
```

*High-Level Specification 8.15:* Existing `WSML` selection imperative expressed as a high-level rule

Let us now refactor the guideline selection policy currently implemented in the `ServicePropertyGuideline` aspect. Figure 8.16 shows a possible refactoring for this policy. Note that the action involves triggering the *prioritize(property)* domain method which in turn implies the execution of the OO method with the same name, predefined in the `ServiceType` class of the WSML, in charge of ranking the existing service compositions of the service type according the their values of the property received as parameter. The implementation of this OO method ensures that a ranking of service compositions is made with respect to the property defined as parameter. The advantage of pulling the guideline logic up to the domain level in the form of a high-level rule is that other condition can be specified under which the ranking of services is performed. In the example shown in Figure 8.16, the guideline logic only triggers when the service type is unreliable. These conditions can be varied as needed, without having to change the current WSML implementation.

```
BR       BRPrioritizeServicesBasedOnSpeed

PROPS Long AS speedThreshold,

USING   ServiceComposition AS sc

IF       sc.serviceType.fasterThan(speedThreshold)

THEN    sc.serviceType.prioritize("Average speed")
```

```
CONNECT  BRPrioritizeServicesBasedOnSpeed
PROPS    500
AFTER    setAverageSpeedToServiceCompositionEvent
MAPPING  setAverageSpeedToServiceCompositionEvent.serviceComposition TO sc
```

```
CONNECT  BRPrioritizeServicesBasedOnSpeed
PROPS    500
AFTER    addNewServiceCompositionEvent
MAPPING  addNewServiceCompositionEvent.newService TO sc
```

```
BR       BRPrioritizeServicesBasedOnFailures

USING   ServiceComposition AS sc

IF       sc.serviceType.isUnreliable()

THEN    sc.serviceType.prioritize("Invocation Failures")
```

```
CONNECT  BRPrioritizeServicesBasedOnFailures
AFTER    setNumberOfFailuresToServiceCompositionEvent
MAPPING  setNumberOfFailuresToServiceCompositionEvent.serviceComposition TO sc
```

```
CONNECT  BRPrioritizeServicesBasedOnFailures
AFTER    addNewServiceCompositionEvent
MAPPING  addNewServiceCompositionEvent.newService TO sc
```

*High-Level Specification 8.16:* Existing `WSML` selection guideline expressed as a high-level rule

### 8.4.3 Open Issues

It might be that some services have been added to the system before the actual addition of the selection rules. As the rules are only triggered on the occurrence of the connection events, if for those existing services the pertinent events do not occur after the addition of the rules, the rules are not going to be applied on those services. As a consequence, the selection will work partially, only for those services for which the events are to occur at some point in time after the addition of the selection rules.

A possible solution to this problem involves defining "meta rules", i.e. rules that state something about the rules. Such a meta rule can be triggered on events that occur not on the core application but on the high-level domain infrastructure. Such an event would be for instance the addition or removal of a high-level rule or connection. At those events, other rules might trigger. In the case of the selection, a meta rule could be defined which would ensure that the selection rules are applied right after the moment they are added or removed in the system. This would enable applying the rules for those services that were already incorporated in the system before the actual rules.

## 8.5 Discussion

Besides enhancing flexibility and configurability of the WSML framework, the experiments presented in this chapter illustrated the following advantages of our approach:

1. *improved understandability*: all business rules and their connections are expressed at the domain level, this way hiding the technical complexity of the WSML (shown in sections 8.3.3 and 8.3.4).
2. *extension of the domain model*: new domain entities that embody domain knowledge existing in the WSML implementation can be pulled up to the domain level (shown in sections 8.3.4.1 and 8.3.4.2).
3. *support for the unanticipated evolution of the WSML*
    - the core functionality of the WSML is extended non-invasively with the realization of unanticipated domain concepts (shown in section 8.3.4.3).
    - new business rules — about both anticipated and unanticipated domain concepts — can be added at run time (shown in sections 8.3.4).
4. *improved variability*: the conditions that guide the selection, integration and management tasks offered by the WSML can be automatically varied at run time, i.e. without having to stop the execution of the WSML (shown in section 8.4).

Still our approach requires defining mappings to implementation that rely on having knowledge about the WSML implementation, which might be tedious (e.g. mappings defined in the high-level specification 8.1). In order to define these low-level mappings, a domain expert might need to collaborate with a developer. However, one has to incur the effort of defining these mappings only once. When these mappings are put in place, high-level mappings can be more easily added (most likely) by only a domain expert.

As several aspects are generated for the high-level rule connections, conflicts might occur between them. Furthermore, given that aspects exists and are managed by the WSML, possible interference can occur between those WSML aspects and the aspects generated by our framework. Extra mechanisms might be needed to tackle these issues, as explained in section 10.2.3.2.

Other WSML configuration rules of interest can be identified which were not tackled in this chapter. This is because these rules require mappings that are not currently supported in the domain model approach. For example, mappings to AOP are needed in order to pull up management functionality that is currently implemented in the aspects themselves and not manageable from the OO layer of the WSML. Moreover, the complex nature of this AOP management framework makes it impossible to replace the way some management tasks are carried out. For example, the WSML implements aspects that hook on other aspects, aspects that share their triggering points, reordering of aspects at run time, etc. Dealing with these aspect issues requires a perfect synchronization between the different framework's components in order for things to work as expected. Thus, expressing these tasks at the high level is currently not possible.

**Contextual rules.** We can imagine rules that involve checking certain conditions based on the information available at the client side, e.g. the client context, information about the users of the client application, etc. Although these rules are not supported in the WSML, they could be expressed at the high level. Expressing these rules at the high-level however requires deploying the WSML layer on a concrete client application. When this deployment is ensured, domain entities can be defined which map not only to the OO layer of the WSML but also to the OO entities defined by the client application.

**Request/response rules.** We can imagine rules that check certain conditions on information that is available at the moment a functional request is performed on a service. Whether this condition is validated or not can imply proceeding or not with the actual service redirection. This mechanism enables selecting the most appropriate service for a given request on a per-request basis. In order these rules to be expressed at the domain level our approach should be able to express events that correspond to the execution of advices in aspects.

**Composition rules.** The WSML supports reactive service compositions, i.e. compositions in which the roles are represented by service types instead of concrete services. This allows binding the roles to concrete services at run time. We have seen in this chapter that business rules can trigger selection on service types. These rules can be used to ensure that certain conditions are satisfied by the services involved in the reactive composition. However, more global business rules that check conditions to be satisfied by the overall composition are not supported, e.g. a rule that ensures that the overall execution time of the service composition does not exceed a certain threshold.

## 8.6 Summary

In this chapter we have evaluated the domain model approach presented in this dissertation by showing how it can enhance the support for business rules in a complex and technical case study, the WSML. This validation has been done in two scenarios: *evolution* and *refactoring*. In the first scenario, we have built a domain model capturing an initial set of selection and management concepts that are anticipated in the implementation of the WSML and shown how those concepts can be used to define new high-level configuration business rules. Furthermore, we have demonstrated that domain evolution is supported by means of extending the existing domain model with new domain entities. In this endeavor, different kinds of mappings introduced in chapter 6 have been illustrated. A second scenario, the refactoring one, has shown that existing selection policies in the WSML can be refactored using the domain model approach and expressed as high-level business rules. These high-level rules are automatically translated to aspects that can replace the existing selection aspects foreseen in the WSML.

# Chapter 9

# Related Work

In this chapter we analyze different approaches that relate, in one way or another, to the work presented in this dissertation. In section 9.1, several commercial business rules systems are described with respect to their business rule languages, their support for expressing business rules at the domain model and the rule execution model supported. In section 9.2, lightweight approaches to business rules are described. Section 9.3 describes related approaches that use AOP for decoupling business rules. Section 9.4 discusses several approaches that aim at combining MDE and AOP. In section 9.5, related approaches that research the mapping between several knowledge representation mechanisms are described. Section 9.6 enumerates some related approaches that also aim at externalizing business rules in the domains considered in this dissertation, i.e. the e-commerce and service-oriented domains. Finally, section 9.7 presents some work on business rule methodologies, vocabularies and rule engine standards.

## 9.1 Business Rules Systems

In this section we analyze several state-of-the-art approaches, some of them commercial BRMS systems. They are related to our approach in that they have the same goal, i.e. the decoupling of business rules from software applications. Moreover, they are relevant because they propose expressing business rules at different levels of abstraction and even some of them aim at defining rules in terms of domain concepts which are mapped to an implementation. The execution model supported by these approaches is different to ours because it is based on a rule engine that has to be triggered explicitly from the core application code. Although we situate our work differently with respect to the actual rule execution, it is still of interest to look at these approaches in detail. The criteria taken into account for their analysis are:

- *Business rules*: are rules expressed at the low or high level, or both?
- *Domain model*: how is the mapping of domain knowledge to implementation specified?
- *Rule Integration and Execution*: how are rules triggered from core applications?

### 9.1.1 JRules

ILOG JRules [ILO] is a complete BRMS for the Java environment. It includes tools for modeling, writing, testing, deploying and maintaining business rules. ILOG JRules allows application developers to combine rule-based and object-oriented programming to add business rule processing capabilities to new and existing applications. It is built on a set of foundation classes that provide Java application programming interfaces (APIs) that allow creating, managing and customizing the rule repository, and manage the business rules contained in it. In addition, the APIs provide the classes for deploying the rule engine in any Java environment. ILOG BRMS provides a repository for organizing and storing business rules, and a rule engine for executing them [ILO06].

#### 9.1.1.1 Business Rules

In JRules, a business rule defines one or more conditions, which when met, result in one or more actions. Several business rule languages are provided which allow the expression of business rules at different levels of abstraction.

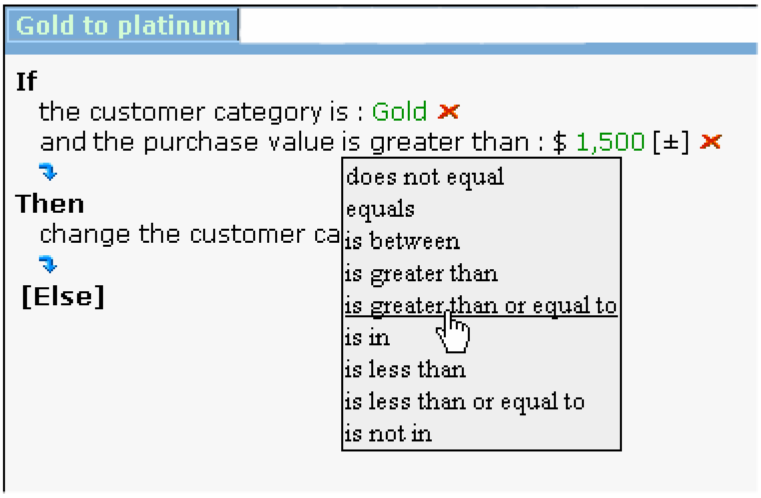
The rule language provided at the low level is called Ilog Rule Language (IRL), which is the language understood by the rule engine. This language is a rule-based programming language targeted at developers. Rules written in IRL can directly reference any application object, like a Java object or an object derived from XML data. This language offers full support for Java operators to be used in expressions as well as support for Java data structures.

At a higher level of abstraction, a default high-level rule language is provided, the Business Action Language (BAL). This language includes default concepts and entities needed in any domain or business, and thus it is quite general-purpose. In order to define a more domain-specific business rule language, a first possibility is to extend this default BAL language. A second possibility is to define a complete new custom-made high-level business rules language from scratch by using the capabilities offered by the Business Rules Language Definition Framework (BRLDF). This framework enables the definition of new languages in XML files. For either extending BAL or defining a new language, programming skills are needed as the translation from the custom language to IRL must be specified using either eXtensible Stylesheet Language Transformations (XSLT) or Java code. Also, the BRLDF is built on top of another framework called the Token Model, which specifies the syntax of a business rule language. In this framework, the different parts of the rule are represented as tokens — implemented in token classes. In order to instantiate these token classes for the definition of new domain-specific rule languages, programming skills are again needed. Our approach improves on this in that the definition of some domain concepts that form the vocabulary of new domain-specific languages can be specified completely at the domain level, without the need for programming skills.

Similar to our approach, JRules supports rule templates, which partially define a business rule and contains placeholders for missing information. In addition, it offers a default template library containing a set of templates and a basic business vocabulary used in the templates. This basic predefined vocabulary is comparable to the features of our high-level rule language. Figure 9.1 depicts how this predefined basic vocabulary can be used in the definition of JRules' rules.

#### 9.1.1.2 Domain Model

In order to write high-level rules, besides a business rule language (either BAL or a custom-made one), a Business Object Model (BOM) is needed. A BOM defines the entities (classes, attributes and methods) used in the definition of the business rules, and maps the natural



Gold to platinum

If
the customer category is : Gold
and the purchase value is greater than : $ 1,500 [±]

Then
change the customer ca

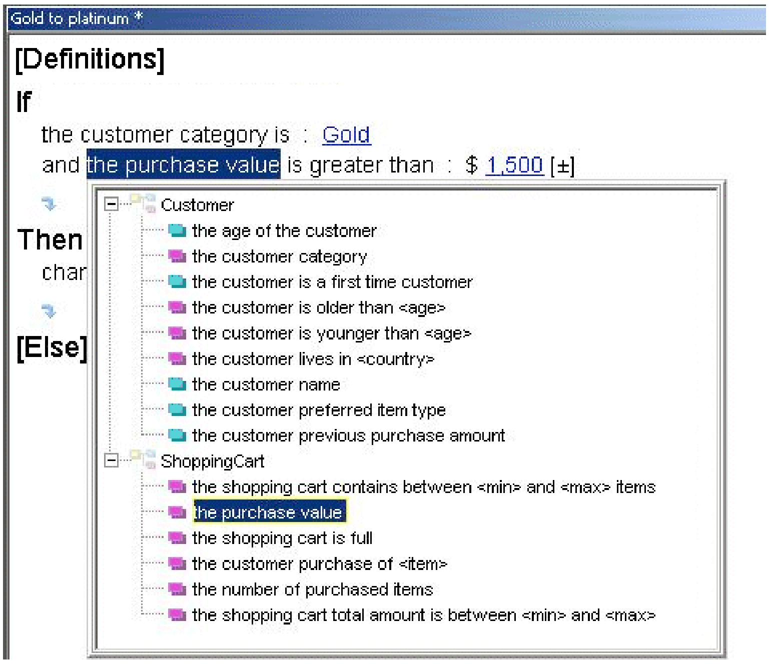
[Else]

*Figure 9.1:* Using predefined business vocabulary in the definition of a high-level rule in JRules (taken from [ILO04])

language-like syntax of the business rule language to them. These entities are based on business terms an uses an intuitive structured syntax. Figure 9.2 illustrates how a high-level rule can be simply created by pointing and clicking on elements defined in a BOM.

The BOM is constructed as projection of existing Java object models, XML schemas or Web services, or from a set of business-oriented classes that do not necessarily map directly to the underlying Java classes or XML schemas (known as "virtual classes"). Information derived from Java classes and databases use the ILOG JRules Java binding and database binding functions respectively, while XML and Web service schemas are converted to Java-like objects by means of the XML binding function. As a consequence, rules can reference data mapped to both Java objects and XML objects, and these objects can coexist in working memory. The BOM is translated into eXecution Object Model (XOM) which defines the application classes that the rules can act upon. The classes in the XOM can be bound to different types of data, including a Java object model, XML or Web service schemas or a combination of the previous ones. The XOM and its bindings to data must be set up by the developers. Contrary to our approach, derived domain concepts could be defined but only at the low level, in terms of existing data in different sources. The definition of derived domain concepts at the high level, in terms of existing domain concepts in a BOM, is not possible. Moreover the BOM falls short at defining domain vocabulary that is not present in the existing sources. This is a limitation as domains evolve and new unanticipated concepts cannot be incorporated to the BOM.

In order to execute high-level business rules, they need to be translated into the ILOG Rule Language (IRL) execution rules. This translation requires the BOM-XOM binding specifications to be previously defined.



Gold to platinum *

[Definitions]

If

the customer category is : Gold

and the purchase value is greater than : $ 1,500 [±]

Then

char

[Else]

- Customer
  - the age of the customer
  - the customer category
  - the customer is a first time customer
  - the customer is older than <age>
  - the customer is younger than <age>
  - the customer lives in <country>
  - the customer name
  - the customer preferred item type
  - the customer previous purchase amount
- ShoppingCart
  - the shopping cart contains between <min> and <max> items
  - the purchase value
  - the shopping cart is full
  - the customer purchase of <item>
  - the number of purchased items
  - the shopping cart total amount is between <min> and <max>

*Figure 9.2:* Using domain concepts in the definition of a high-level rule in JRules (taken from [ILO04])

#### 9.1.1.3 Rule Integration and Execution

IRL rules are managed and executed by a rule engine. Moreover, in order to define how rules need to be combined, ruleflows can be defined specifying rule ordering strategies (using dynamic priorities, static priorities, or following an explicit sequence defined by the user), rule firing strategies (e.g. fire all the eligible rules or fire one rule and exit the task) and an execution algorithm (either Rete-based or using sequential byte-code generation for optimal performance). However, the actual connection of the rules with the existing application still occurs at the implementation level: the rule engine needs to be invoked from the existing code. Thus, the rule connection is crosscutting in the core application and, moreover, excludes the domain expert.

### 9.1.2 Haley Rules

Haley's approach to business rules [Halb] focuses on bridging the gap between requirements and design. It focuses on structuring the decision making process by means of the definition of policies and the analysis of how those policies relate to each other. It also focuses on the methodology for capturing the minimal set of statements that will execute a business goal. Ultimately — in accordance with our goal — this approach also aims at integrating executable business rules in a running application.

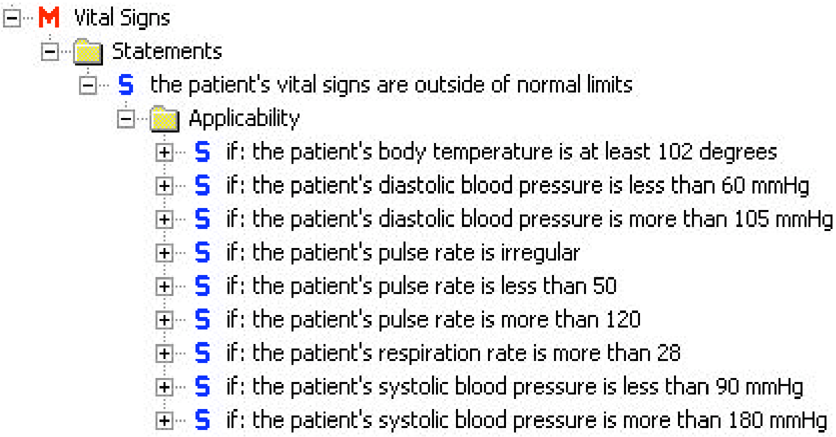
The main tool in the Haley suite is HaleyAuthority, a BRMS targeted at managers in order to gain control of their business processes and to adjust quickly, easily and continually to their operations, and to key IT systems to adapt to changing business environments. Besides HaleyAuthority, other tools and APIs are provided: Café Rete [Hala] is a Java class library that provides an inference engine. Authorete is a graphical and speech-driven interface for authoring and managing business logic in structured English sentences. Authorete and Café Rete work together. Sentences written in Authorete are automatically generated for and dynamically loaded by Café Rete.

#### 9.1.2.1 Business Rules

Haley business rules are high-level. Rules are entered in structured English into the Authorete interface and are automatically translated into a low-level representation which is internal and not meant to be regarded or edited by users (contrary to JRules, where developers could directly write IRL rules, if needed). The translation process is done as follows: HaleyAuthority understands the English statements by comparing the terms and phrases in a statements to a Business Concept Model, which is basically a semantic model. English terms and domain concepts defined in the domain model are linked automatically.

HaleyAuthority's emphasis on the English language facilitates decision-support systems, which tend to be diagnostic and prescriptive. It allows arranging and grouping policies and rules in a very flexible way. Thus, how rules relate to each other is one of its main focuses, whereas it is not the main concern in this dissertation (section future work 10.2).

Haley allows to structure statements in the form of a consequence enabled by any number of dependent or independent conditions (see figure 9.3).



- M Vital Signs
  - Statements
    - S the patient's vital signs are outside of normal limits
      - Applicability
        - S if: the patient's body temperature is at least 102 degrees
        - S if: the patient's diastolic blood pressure is less than 60 mmHg
        - S if: the patient's diastolic blood pressure is more than 105 mmHg
        - S if: the patient's pulse rate is irregular
        - S if: the patient's pulse rate is less than 50
        - S if: the patient's pulse rate is more than 120
        - S if: the patient's respiration rate is more than 28
        - S if: the patient's systolic blood pressure is less than 90 mmHg
        - S if: the patient's systolic blood pressure is more than 180 mmHg

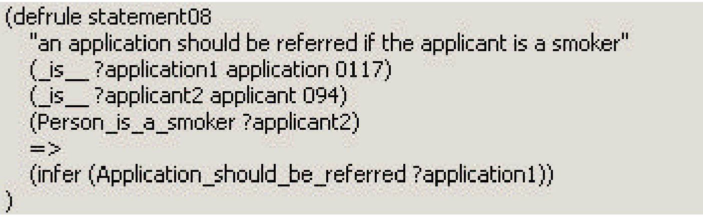
*Figure 9.3:* Example of the definition of a Haley business rule: different conditions conclude a single action (taken from [Hal05])

Another alternative for the definition of a rule is to write the rule in structured English, as for instance:

**"an application should be referred if the applicant is a smoker"**.

This is a well-defined rule only if all the concepts used in the rule exist in the domain model. Given a high-level rule, HaleyAuthority can automatically generate Eclipse code — the low-level Haley rule language — for that business rule. The generated low-level rule for the example high-level rule is shown in 9.4. Besides Eclipse code, HaleyAuthority can also generate business logic that is implemented without a rules engine. HaleyAuthority can generate Java or C++ code that implements business logic within an object model.

HaleyAuthority can also generate SQL code that implements business logic using stored procedures within a database. However, if there will be many such statements, it is most efficient to implement them with a rule language whose performance is asymptotically independent of the number of rules (i.e., as the number of rules increases runtime performance becomes constant).



```
(defrule statement08
   "an application should be referred if the applicant is a smoker"
   (_is__ ?application1 application 0117)
   (_is__ ?applicant2 applicant 094)
   (Person_is_a_smoker ?applicant2)
   =>
   (infer (Application_should_be_referred ?application1))
)
```

*Figure 9.4:* Example of low-level Eclipse code representing the translation of a Haley business rule (taken from [Hal05])

#### 9.1.2.2 Domain Model

The elements referred to in the high-level rules need to be defined in a business model. A business model captures how words (linguistics), that may be grouped and order with proper grammar (syntax), relate to each other. With this model, one understands how, when, and where a term may be used, compared, aggregated, etc. The elements in this business model are created when new terms are referred to in the natural language expression of the rules. One can say “a driver is high risk if the driver has more than two tickets in the last 12 months” and HaleyAuthority will understand the following terms (examples taken from [Hal05]):

that “the driver” refers to *a driver*
that “a driver” can be *high risk*
that “a driver” can have *tickets*
that “tickets” can have a *quantity* and it understands the *value* of the quantity
that “12 months” is a *period of time*
that “in the last” is a *relative period of time*

These terms are elements of a business model. With this understanding, along with a few other terms and actions, HaleyAuthority could then allow other “derived” statements such as, “Send all high risk drivers notification form F”, or “if the number of tickets for a driver is 0 for the past 2 years, send notification form A, unless they are under 18 years old” without constraint of fixed text string menus, tabular forms, and the like. This can be compared to the high-level derived domain entities supported in our approach. For every new English term that does not have a counterpart in the domain model, a new domain entity has to be added manually (through the Authorete interface) to the domain model. Similarly to our approach, the Haley domain model supports inheritance between domain concepts.

However, what is not evident in this approach is the link between those domain elements and the implementation. This link has to be specified manually by the developers who can map those domain elements to data stored in different mediums such as XML, SQL, COM, Java and .NET models. Using these mappings, the high-level rules are automatically generated to Eclipse code which is written to a set of files that will be used by the other tool Café Rete to affect the behavior of the application.

#### 9.1.2.3 Rule Integration and Execution

The rule integration is based on the explicit invoking the rule engine from the application code every time one needs to assert or update a fact in the knowledge base, trigger rules, etc. Thus, the rule integration is low-level and crosscutting. This is a limitation which is observed and tackled in this dissertation. Similarly to our approach, in the Haley approach changes in the business logic can be incorporated at run-time. This is done by explicitly invoking the rule engine, which is able to load any changes that might occur while the application is running. These changes typically are in relation to deploying new code for new rules defined in Authorete, which implies writing new or additional Eclipse code files and informing the runtime application to load these files, as well as to unload old files that are obsolete.

Although support is provided to accommodate changes at run time, API methods must still be invoked in order to bridge to the rule engine. These invocations have to be anticipated in the code of the application at all places where needed, and are thus crosscutting. We improve on this approach, by supporting the non-invasive addition and removal of rules and domain concepts.

### 9.1.3 VisualRules

Visual Rules [Inn] provides approaches to business rule development. Visual Rule isolates and modularizes business rules and provides developers and business analysts with a high-level and abstract business rule language in which to express them.

#### 9.1.3.1 Business Rules

The business rule language provided by Visual Rules is of course visual. It allows the definition of rule trees, the central element in a Visual Rules project, as graphical models of logic. Besides rule trees, other elements can be found in a project: parameters, variables, static variables, constants, actions and data types.

*Parameters* are the data passed to the rule trees from the context where they are integrated (equivalent to the domain entities specified in the USING clause in our approach). The *variables* and *static* variables folders contain data definitions used within the rules. In addition to the normal variables which are re-initialized at each new rule tree call, *static variables* are supported which retain their value between rule tree calls. *Constants* define values that are not modifiable within rule trees.

As shown in the front panel of Figure 9.5, in Visual Rules a rule tree is defined by selecting and chaining different rule nodes. Visual Rules predefines different kinds of rule nodes, among them *start nodes*, *decisions* (i.e. condition nodes), *assignments* (used to calculate and keep partial results), and *actions* (to execute program code and thus trigger any process).

Note that only a few predefined action types are supported by Visual Rules. User-defined action types can be added through programming. Both configuring predefined action types and writing new action types is done by manually writing code. The configuration of a predefined action type is depicted in Figure 9.6. Having to write code for configuring or adding actions is a drawback of this approach. Our approach improves on this, as it is possible to add actions in terms of elements of a domain model, without having to write code.

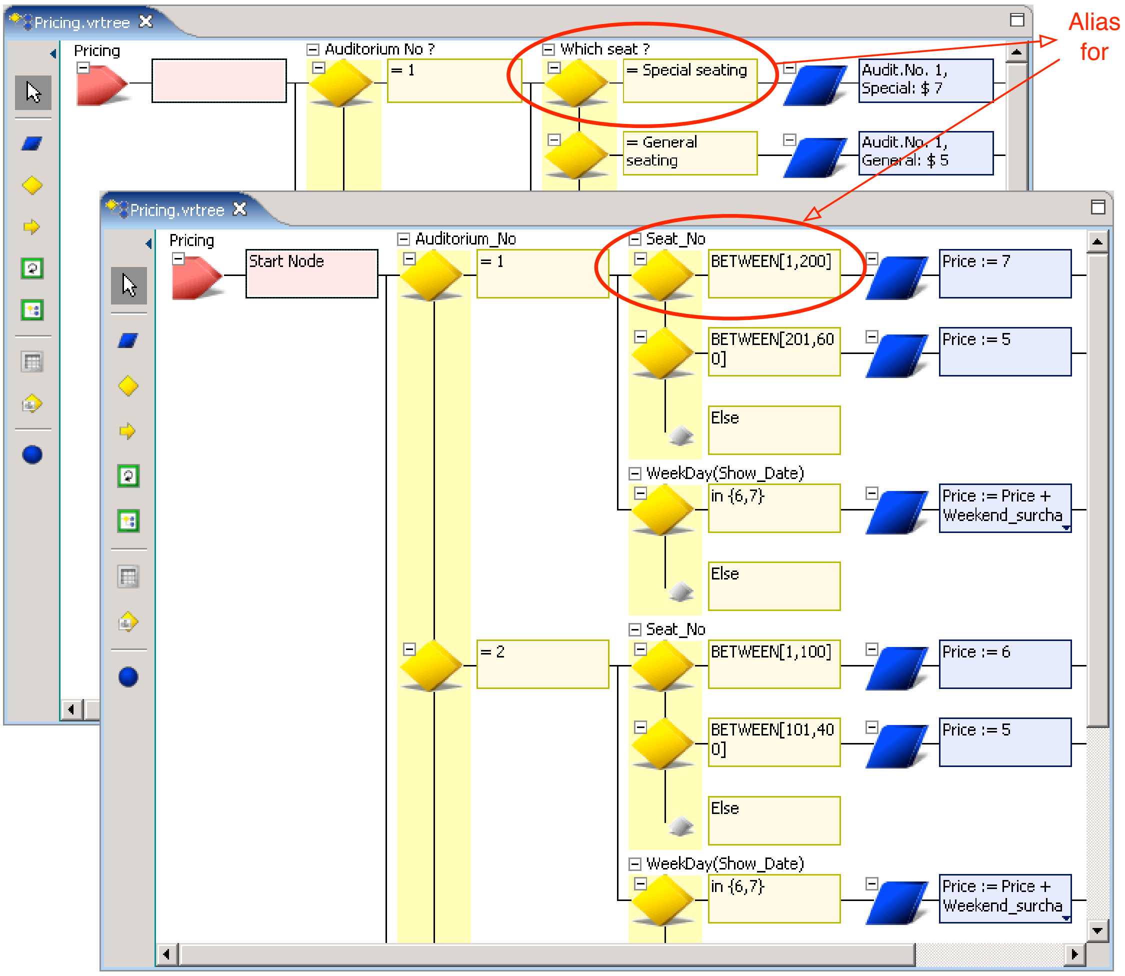
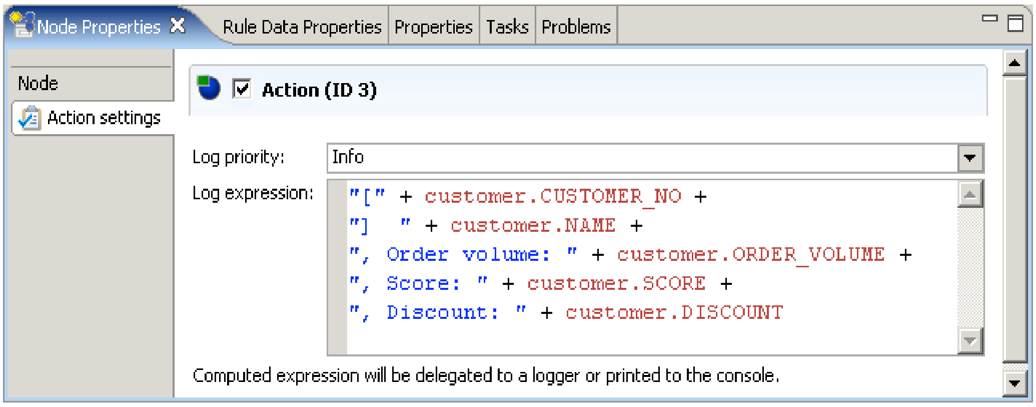


*Figure 9.5:* Example rule trees in Visual Rules (adapted from [Gmb06])

Code is automatically generated for a rule tree. This rule code consists of programs, modules or classes in a specific programming language that precisely map the logic in the rule trees and can be directly integrated into application systems.

### 9.1.3.2 Domain Model

Each data item has a specific *data type* associated, either one of the integrated data types (e.g. Number, String, Boolean, Date, etc.) or a user-defined data type (which can be added by means of adding programming extensions to the framework). This leads to the first



☑ Action (ID 3)

Log priority: Info

Log expression:

```
"[" + customer.CUSTOMER_NO +
"]  " + customer.NAME +
", Order volume: " + customer.ORDER_VOLUME +
", Score: " + customer.SCORE +
", Discount: " + customer.DISCOUNT
```

Computed expression will be delegated to a logger or printed to the console.

*Figure 9.6:* Example of the configuration of a predefined action type in Visual Rules (taken from [Gmb06])

difference with our approach, as our domain model is not statically typed (the types of the domain concepts are determined by the mappings from those concepts to implementation). Another difference is that data types are data structures that exist at the implementation level. This let us conclude that a domain model does not exist in this approach.

The only support at a higher level this approach provides is the possibility to associate a higher-level syntax — in the form of an alias — to both conditions and actions in a rule tree. This support is very limited, since these aliases have a predefined associated meaning. Thus, a one-to-one mapping exists between, on one hand, the kind of condition (or action) and its associated alias, and on the other hand, a (predefined) implementation for that condition (or action). For example, a rule can be written in terms of higher-level aliases which represent lower-level terms, such as the high-level term “special seating”, which in fact is an alias for the comparison `Seat_No BETWEEN[1, 100]`, where BETWEEN is a predefined operator in the framework (depicted in Figure 9.5). All user extensions require programming and thus cannot be added at the domain level.

#### 9.1.3.3 Rule Integration and Execution

The triggering of rule trees is done from the Visual Rules framework and not from an existing core application. The input data and output results are dealt with in XML files. The aim of this approach is not to integrate rules in an existing application but to develop an application from scratch with business rules in mind. Also note that in Visual Rules it is possible to control the execution of the rule tree by means of activating and deactivating specific branches in the rule tree.

### 9.1.4 JBoss Rules

JBoss Rules [JBob] (aka Drools 3.0) is a Rule Engine implementation based on Charles Forgy’s Rete algorithm tailored for the Java language. Drools is written in Java, but is able to run on Java and .Net. JBoss Rules is designed to allow pluggeable language implementations.

#### 9.1.4.1 Business Rules

The language supported by JBoss Rules is called Drools Rule Language (DRL). A DRL rule has the following structure:

```
rule "my rule"
       attributes
    when
        LHS
    then
        RHS
end
```

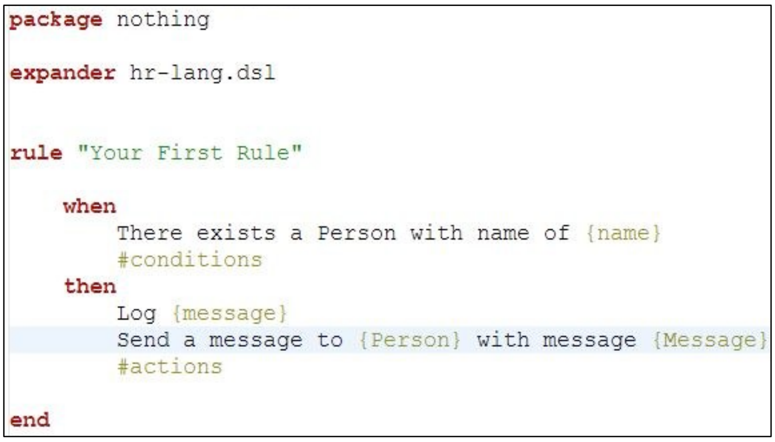
where ⟨attributes⟩ define the information used by the rule, the ⟨LHS⟩ defines the conditional part of the rule whereas the ⟨RHS⟩ defines the rule's action. How the different parts of a DRL rule are filled in depends on the level of abstraction in which the rule is expressed. JBoss Rules supports two levels for the writing of the rules:

- *low-level*: DRL uses Java to express Field Constraints, Functions and Consequences; support for other languages is envisioned (such as Groovy and Python). In low-level rules the ⟨LHS⟩ and ⟨RHS⟩ are Java blocks. In addition to the Java syntax, special keywords can also be used in the RHS for asserting, retracting or modifying facts as well as any variables bound in the ⟨LHS⟩. In addition to the DRL rules, JBoss Rules also allows capturing and managing rules directly expressed in XML, by using the following tags:

  ```
  <rule name="my rule">
    <rule-attribute name="..." value="..." />

    <lhs>
          ...
    </lhs>
    <rhs>
           ...
    </rhs>
  </rule>
  ```

  XML rules are low-level. The tags enclose the actual implementation of the rule in Java. XML rules are kept in JBoss Rules for backwards compatibility reasons (Drools 2.x was entirely based on XML) and therefore its is not one of its main current contributions.

- *high-level*: JBoss Rules allows the definition of domain-specific languages. Concepts which are relevant for a specific domain can be defined to be used in rules. The mappings from those domain concepts to actual Java entities in the implementation is kept in files which are used by an "expander mechanism" in charge of extending the native language with the domain terms. The expander used in each case has to be explicitly specified with the rule definition. An example of a rule defined in terms of a DSL is shown in Figure 9.7. Both the standard DRL and natural language-like extensions are supported by JBoss Rules Workbench. Again, the actual features of the domain-specific languages supported by JBoss Rules are similar to the features of our high-level rule language. The actual difference lies on the kinds of domain concepts that can be expressed as terms of the domain-specific language. This is explained in the following section.



```
package nothing

expander hr-lang.dsl


rule "Your First Rule"

    when
        There exists a Person with name of {name}
        #conditions
    then
        Log {message}
        Send a message to {Person} with message {Message}
        #actions

end
```

*Figure 9.7:* Example of a domain-specific rule in JBoss Rules (taken from [JBoa])

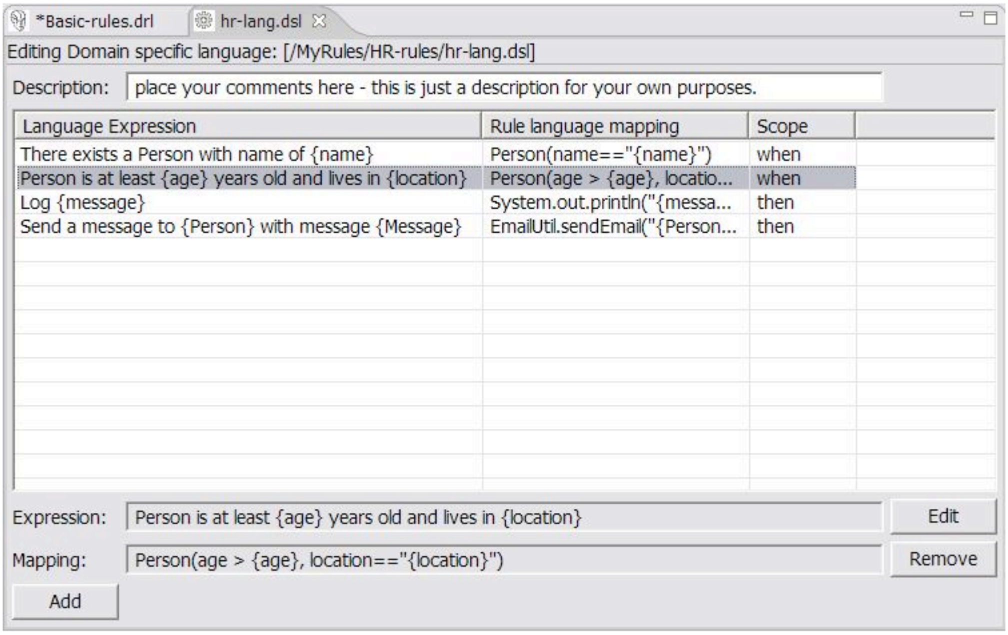
#### 9.1.4.2 Domain Model

A domain model is supported which enables the definition of domain concepts and properties for those concepts. However, contrary to our approach, drools does not support domain operations. Mappings for those concepts and properties can be defined manually through a specialized GUI, as illustrated in Figure 9.8. Domain concepts are defined in a kind of structured natural language syntax[1]. The LHS of a mapping definition states a domain expression with special fields for the variable parts (such as *name* in the first example definition), whereas the RHS designates the implementation of that domain expression. Only mappings to implementation (called low-level mappings in our approach) are supported in JBoss Rules. Contrary to our approach, high-level mappings defined in terms of other existing domain entities are not supported in JBoss Rules. Also, more advanced mappings such as mappings from domain concepts to multiple implementation entities, mappings from properties to complex expressions that might traverse multiple classes in the implementation model, and mappings that are realized using AOP are not supported either.

#### 9.1.4.3 Rule Integration and Execution

The rule engine supported in JBoss Rules complies with the standard Java Rule Engine API (known as JSR94 [Jav]). Analogously to the approaches presented so far in this chapter, in this approach rules are also triggered by means of invoking methods on the rule engine from Java code. Thus again it suffers from the problems imposed by low-level and crosscutting rule connections.

[1] Note that this is new in JBoss Rules (Drools 3). Previous versions of Drools, such as version 2, support for DSLs was based on XML.



Editing Domain specific language: [/MyRules/HR-rules/hr-lang.dsl]

Description: place your comments here - this is just a description for your own purposes.

| Language Expression | Rule language mapping | Scope |
|---|---|---|
| There exists a Person with name of {name} | Person(name=="{name}") | when |
| Person is at least {age} years old and lives in {location} | Person(age > {age}, locatio... | when |
| Log {message} | System.out.println("{messa... | then |
| Send a message to {Person} with message {Message} | EmailUtil.sendEmail("{Person... | then |

Expression: Person is at least {age} years old and lives in {location}

Mapping: Person(age > {age}, location=="{location}")

*Figure 9.8:* Example of the definition of domain specific concepts and their mappings in JBoss Rules (taken from [JBoa])

### 9.1.5 RuleML

RuleML [Rul] is a XML-based Rule Markup Language that has been proposed by the Rule Markup Initiative as a canonical language for publishing and sharing rules on the Web. RuleML is implemented with XML Schema, XSL Transformations (XSLT) and reasoning engines. The RuleML Initiative collaborates with several standards bodies including W3C, OMG and OASIS. Because it is XML-based, RuleML inherits some of its benefits directly from XML, including platform independence and interoperability. RuleML is also extensible, a prime example being its combination with OWL to form the Semantic Web Rule Language (SWRL) [HPSB$^+$04] and its Object-Oriented extension called OO RuleML [Bol03].

RuleML is also translatable to and from other Semantic Web standards (e.g. RDF, OWL) via XSLT. Various tools are also available, including OO jDREW [Bal05], Mandarax[2] , and NxBRE4[3].

RuleML covers the entire rule spectrum, from derivation rules to transformation rules to reaction rules and to integrity checking. RuleML can thus specify queries and inferences in Web ontologies, mappings between Web ontologies, and dynamic Web behaviors of workflows, services, and agents.

[2] Available at http://mandarax.sourceforge.net/.

[3] Available at http://www.agilepartner.net/oss/nxbre/.

#### 9.1.5.1 Business Rules

The rule language is based on XML and is low-level. As an example, consider the following rule:

*"The discount for a customer buying a product is 5.0 percent if the customer is premium and the product is regular"*

This rule is expressed in RuleML as follows:

```
<imp>
  <_head>
    <atom>
      <_opr><rel>discount</rel></_opr>
      <tup><var>customer</var>
        <var>product</var>
        <ind>5.0percent</ind></tup>
    </atom>
  </_head>
  <_body>
    <and>
      <atom>
        <_opr><rel>premium</rel></_opr>
        <tup><var>customer</var></tup>
     </atom>
     <atom>
       <_opr><rel>regular</rel></_opr>
       <tup><var>product</var></tup>
     </atom>
    </and>
  </_body>
</imp>
```

where the tags `<_head>` and `<_body>` represent the action and the condition of the rule respectively. Because RuleML is XML-based, it is hard to adopt for domain experts. This seems to have been a critical factor in the success of this language, as recognized in [Hir06]. In this work, the TRANSLATOR [Hir06] initiative is presented which is aimed at raising the level of abstraction of RuleML. TRANSLATOR is an open source tool which is able to automatically translate natural language-like sentences written in Attempto Controlled English [FHK$^+$05] into the RuleML rules. Similarly to this approach, we can imagine adding a layer on top of our high-level rules to allow expressing them in a natural language-like format.

#### 9.1.5.2 Domain Model

In RuleML, domain concepts are represented as facts which can be used in rules. By default, the facts are not linked to a core OO application. However, some RuleML extensions do support the concept of mappings, such as for example OO RuleML. OO RuleML is a frame-like knowledge representation with facts and rules. In this approach, a correspondence exists between the concepts and rules and the OO Programming: facts correspond to instances, signatures can be viewed as classes, and rules correspond to methods [Bol03]. Analogously to our approach, this correspondence can be seen as a mapping between the knowledge expressed in OO RuleML and the OO entities of an OO program. The difference is that in

OO RuleML, rules are declarative, and therefore they only query or derive information. In our approach, rules ultimately trigger some behavior in an OO program, following a more procedural style, which can affect the execution of the base core application.

The way the mapping from facts and rules to OO is specified varies from concrete implementations to another. For example, in Mandarax, the mapping is done by splitting domain expressions into subexpressions until the level of terminal elements is reached. These terminal elements are then manually mapped to Java entities, as shown in Figure 9.9. This mechanism is called "wrapping" [Die03]. Wrapping is done by programing, more concretely, by instantiating classes provided by the framework which wrap java entities (i.e. methods and attributes). For instance, the term *indexOf("abc","a")* — where indexOf is a wrapper for the `String.indexOf()` method — is translated to the constant term "0" using reflection. Thus, only low-level mappings are supported, which is a limitation of this approach compared to ours.

Support for rules expressed in natural language is provided by an extension of Mandarax, the Oryx tool, which is a graphical front-end that supports verbalization of knowledge (shown in Figure 9.10). Oryx manages a repository of domain concepts to be used in rules. A knowledge base is associated with a repository that contains "meta" information about predicates, functions, data sources, knowledge verbalization, etc.

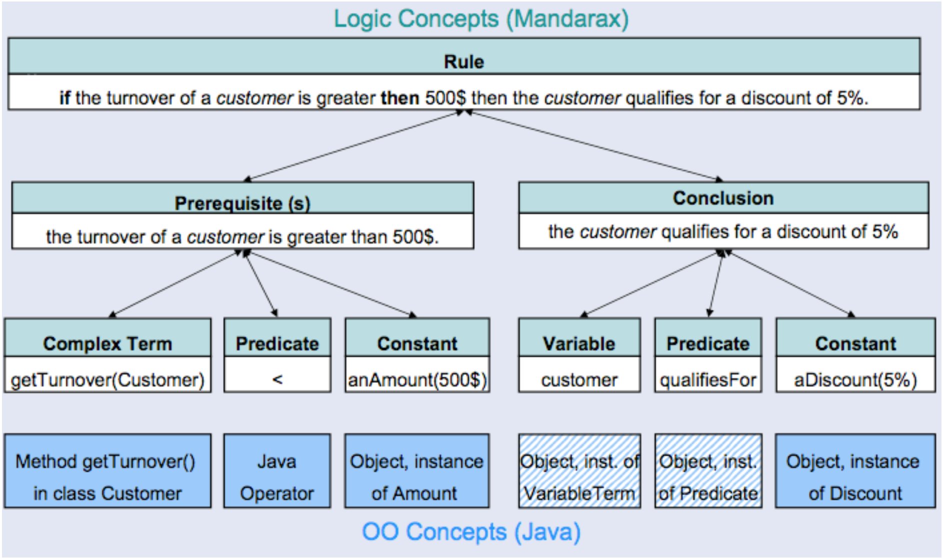
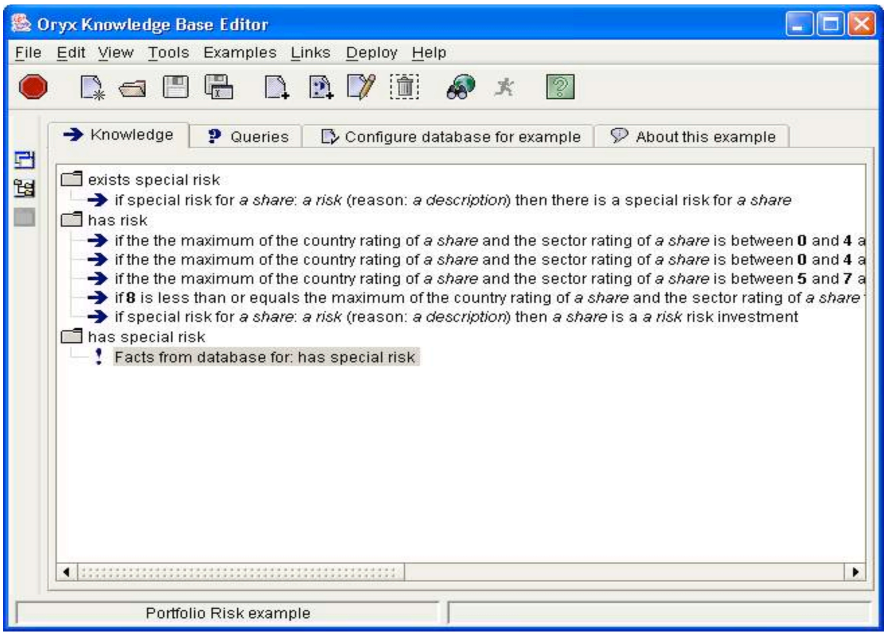

Logic Concepts (Mandarax)
Rule
if the turnover of a customer is greater then 500$ then the customer qualifies for a discount of 5%.
Prerequisite (s)
the turnover of a customer is greater than 500$.
Conclusion
the customer qualifies for a discount of 5%
Complex Term
getTurnover(Customer)
Predicate
<
Constant
anAmount(500$)
Variable
customer
Predicate
qualifiesFor
Constant
aDiscount(5%)
Method getTurnover() in class Customer
Java Operator
Object, instance of Amount
Object, inst. of VariableTerm
Object, inst. of Predicate
Object, instance of Discount
OO Concepts (Java)

*Figure 9.9:* Mapping domain concepts to Java entities in Mandarax (taken from presentation titled "MANDARAX+ ORYX: An Open-Source Rule Platform", by J. Dietrich and G. Wagner (2004))

#### 9.1.5.3 Rule Integration and Execution

RuleML's execution model is based on rule engines. Several rule engines implementations exists for RuleML. Typically rule engines are controlled from core applications by means of the invocation of methods provided in well-defined APIs. This means that rules are triggered and results are retrieved through APIs. Again, the rule connection is low-level and crosscutting.


Oryx Knowledge Base Editor
File Edit View Tools Examples Links Deploy Help
Knowledge
Queries
Configure database for example
About this example
Portfolio Risk example

exists special risk
if special risk for *a share*: *a risk* (reason: *a description*) then there is a special risk for *a share*
has risk
if the the maximum of the country rating of *a share* and the sector rating of *a share* is between **0** and **4** a
if the the maximum of the country rating of *a share* and the sector rating of *a share* is between **0** and **4** a
if the the maximum of the country rating of *a share* and the sector rating of *a share* is between **5** and **7** a
if **8** is less than or equals the maximum of the country rating of *a share* and the sector rating of *a share*
if special risk for *a share*: *a risk* (reason: *a description*) then *a share* is a *a risk* risk investment
has special risk
Facts from database for: has special risk

*Figure 9.10:* The Oryx natural language front-end for Mandarax (taken from presentation titled “MANDARAX+ ORYX: An Open-Source Rule Platform”, by J. Dietrich and G. Wagner (2004))

### 9.1.6 QuickRules

QuickRules [YAS] offers a Business Rules Management Solution (BRMS) for the Java/J2EE platform. It provides an Eclipse-based graphical user interface targeted at domain experts and developers which allows for the design, implementation, testing, and deploying of business rules

#### 9.1.6.1 Business Rules

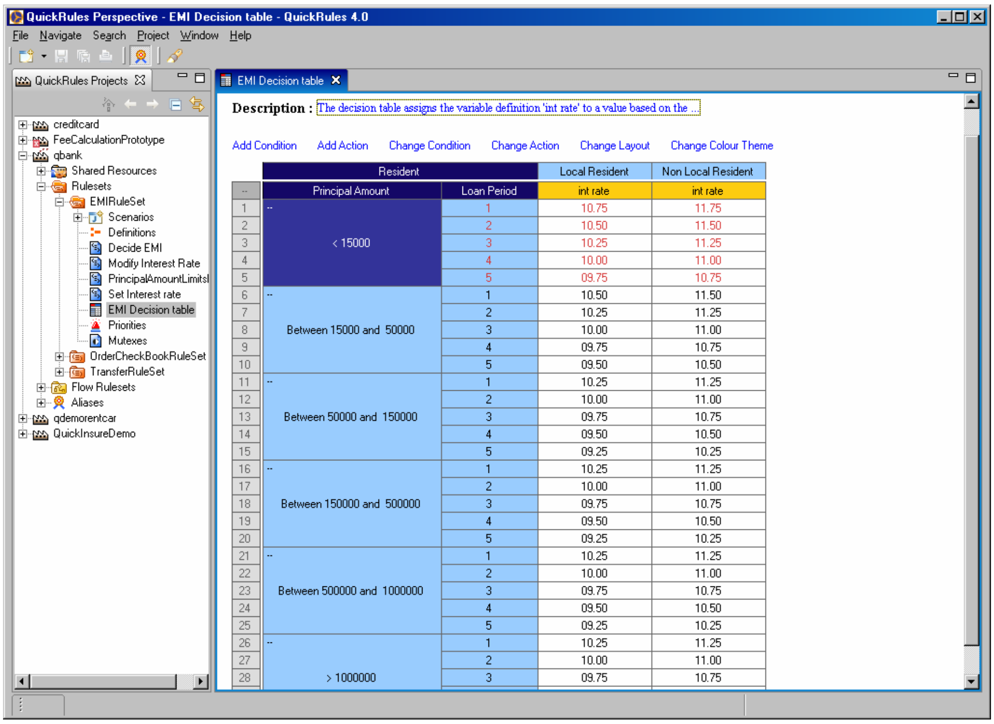
Rules are high-level expressed in terms of natural language terms, which are aliases for implementation entities. QuickRules BRMS enables capturing business policies as a combination of Sequential Rules (FlowRulesets), spreadsheet-like Decision Tables, and plain IF-THEN rules. An example of a decision table in QuickRules is shown in Figure 9.11. QuickRules BRMS allows business rules to be associated with a “validity” period. Using this feature, the client application can execute previous versions of business rules by querying on the data on which they were valid. Versioning is not provided in our approach.

#### 9.1.6.2 Domain Model

A simple domain model is supported which only allows the definition of aliases that have a one-to-one mapping to data objects. This feature is called “Aliasing”. More complex mappings are not supported.

#### 9.1.6.3 Rule Integration and Execution

QuickRules’s rule engine is compliant with JSR-94 Specification [Jav]. Rule sets are executed by means of invoking the rule engine through the methods supported by the API. The



**Description :** The decision table assigns the variable definition 'int rate' to a value based on the ...

|  | Resident |  | Local Resident | Non Local Resident |
|---|---|---|---|---|
| -- | Principal Amount | Loan Period | int rate | int rate |
| 1 | --<br>< 15000 | 1 | 10.75 | 11.75 |
| 2 |  | 2 | 10.50 | 11.50 |
| 3 |  | 3 | 10.25 | 11.25 |
| 4 |  | 4 | 10.00 | 11.00 |
| 5 |  | 5 | 09.75 | 10.75 |
| 6 | --<br>Between 15000 and 50000 | 1 | 10.50 | 11.50 |
| 7 |  | 2 | 10.25 | 11.25 |
| 8 |  | 3 | 10.00 | 11.00 |
| 9 |  | 4 | 09.75 | 10.75 |
| 10 |  | 5 | 09.50 | 10.50 |
| 11 | --<br>Between 50000 and 150000 | 1 | 10.25 | 11.25 |
| 12 |  | 2 | 10.00 | 11.00 |
| 13 |  | 3 | 09.75 | 10.75 |
| 14 |  | 4 | 09.50 | 10.50 |
| 15 |  | 5 | 09.25 | 10.25 |
| 16 | --<br>Between 150000 and 500000 | 1 | 10.25 | 11.25 |
| 17 |  | 2 | 10.00 | 11.00 |
| 18 |  | 3 | 09.75 | 10.75 |
| 19 |  | 4 | 09.50 | 10.50 |
| 20 |  | 5 | 09.25 | 10.25 |
| 21 | --<br>Between 500000 and 1000000 | 1 | 10.25 | 11.25 |
| 22 |  | 2 | 10.00 | 11.00 |
| 23 |  | 3 | 09.75 | 10.75 |
| 24 |  | 4 | 09.50 | 10.50 |
| 25 |  | 5 | 09.25 | 10.25 |
| 26 | --<br>> 1000000 | 1 | 10.25 | 11.25 |
| 27 |  | 2 | 10.00 | 11.00 |
| 28 |  | 3 | 09.75 | 10.75 |

*Figure 9.11:* Example of a decision table defined in QuickRules (taken from [YAS03])

QuickRules BRMS runtime can be deployed as a plain-Java component, in EJB mode, as a Message-Driven Bean (MDB), or as a WebService. Again, the rule integration is low-level and crosscutting.

### 9.1.7 Summary

Some approaches, such as JRules, HaleyRules, RuleML and QuickRules provide a rule-based language for expressing rules, which is more declarative than for example an object-oriented programming language. However, rule-based languages are *programming* languages, requiring the user to have programming skills, as opposed to high-level languages, which can be used by domain experts. Nevertheless, some approaches provide a high-level, declarative language for expressing rules in addition to the rule-based programming language. The former is typically translated into the latter. Examples of systems that support this are JRules, QuickRules, VisualRules and HaleyRules. The high-level rules are also defined in terms of domain concepts captured by a business model. This model is the result of either manually or automatically extracting domain knowledge from an existing object-oriented implementation or XML schemas. Basically, the business model defines the OO classes and methods to which the business rules are applied, and maps the natural language-like syntax of the business rule language to these implementation entities.

Thus, the domain entities are simply aliases for implementation entities, requiring a one-to-one mapping between them. As a consequence a tight coupling exists between the domain model and the implementation model by supporting only anticipated one-to-one mappings. This is a problem since a high-level specification of business rules can be discrepant from the implementation of the core application as the business rules are not always anticipated in the original application. Thus, one-to-one mappings are not enough to realize unanticipated business rules. Moreover, connections are crosscutting in the core application and cannot be expressed at the high level. Our approach extends this idea in several dimensions: we consider the case where domain concepts can be derived from other existing domain concepts — and therefore their mappings can be expressed completely at the domain level—, mapping to many entities and unanticipated mappings.

Rule-based programming languages (e.g., the ILOG Rule Language (IRL) supported in JRules) typically allow writing object-oriented code directly in the rules themselves. This makes the rules more powerful but also more complex, since the whole complexity of the object-oriented paradigm is added. Rules are not high-level since they directly refer to implementation entities. In our approach, rules are very simple because only references to domain attributes and domain method invocations of a domain model are possible. All the complexity that would otherwise appear in the rules themselves, is encapsulated in a mapping. This mapping can be very sophisticated, making the rules very powerful.

In some existing approaches, an intermediate language is used during the rule translation. This is the case of JRules for instance, where high-level rules expressed in terms of a business model map to low-level executable rules expressed in irl, the language understood by JRule's engine. This implies that translated rules can only be reused in JRules-enabled applications.

Moreover, all previous approaches are based on rule engines that are in charge of asserting and executing the rules. The rule engines need to be triggered explicitly from the applications that integrate the rules. Also, application objects must be asserted into, retracted from or updated in the working memory before pattern matching in rules can begin. This is typically done either by using keywords within rules or through APIs invoked within application objects themselves. As a consequence, the connection of the rules hampers the reusability and maintainability of the application code, since every time rules/connection change, the application has to do so.

## 9.2 Lightweight Business Rule Approaches

In this section some lightweight approaches to business rules are discussed. Lightweight approaches are typically adopted by applications for which adding the whole power of a rule engine and rule-based languages might be overkill. We touch upon some approaches that model rules in databases (section 9.2.1), others that tackle the decoupling of business rules in object-oriented applications and others that focus on identifying the variability points and externalizing business rules from applications (section 9.2.2).

### 9.2.1 Business Rules and Databases

In [vH01], the distinction between "service-oriented" and "data-change-oriented" business rule approaches is made.

In "service-oriented" approaches, rules are activated when certain events occur in the core application functionality. Rules in these approaches can be very complex as they can embody an entire knowledge-intensive subtask of the application functionality. Typically data or objects are passed to the rules from the application and results are passed back after the rule execution. This is the kind of rules considered in this dissertation.

On the contrary, in "data-change-oriented" business rule approaches, the design and implementation of the rules are data-centric. Typically the data or object model is defined first, on which the rules are later on attached. The rules can refer to or manipulate the data or objects on which they are attached. Their execution is also data-centric, as they are activated when the data or objects they refer to are changed by the running application or by the other rules' actions. The sequence in which the rules are executed is determined by the sequence in which the information is processed by the application. Moreover, these rules have a short life-cycle since it is determined by the life-cycle of the data or objects they are attached to.

Some approaches in this last category represent business rules in the database layer, as advocated in Business Rules books such as [Dat00]. When this approach is followed, business rules are encapsulated as store procedures. These approaches typically trigger rules at events that occur in the database, which basically correspond to the moment a value is added, removed or modified in a table. Thus, these events are only a subset of the possible core application events that are considered in service-oriented approaches.

Other data-centric approaches implement business rules as entries in database tables. An example is the *Adaptive Object-Model* (AOM) [YJ02], a reflective architecture alternative to traditional object-oriented design which represents classes, attributes, relationships, behavior as well as business rules as metadata. AOM heavily relies on the use of the traditional design patterns [GHJV95]. AOM advocates that some business rules — structural constraints such as cardinality of relationships and if a certain attribute is required or not — can be stored in shared databases that are accessed and manipulated by a core application. Changing business rules is simply done by changing database entries, which can be done dynamically realizing a high flexibility and run-time configurability. However, AOM recognizes that other rules cannot be implemented as database entries — more functional or procedural rules — for which it advocates the use of the Rule-Object pattern [Ars01].

### 9.2.2 Business Rules and Design Patterns

As motivated in this dissertation, having to introduce the extra overhead induced by the use of a dedicated rule-based technology might be overkill for some applications. Some approaches deal with this issue and propose taking a lighter approach to business rules. For instance, some approaches decouple business rules in the standard object-oriented paradigm. An example is the approach used in this dissertation, the *Rule-Object Pattern* which suggests reifying business rules as *Rule Objects*. Many approaches exist which are based on the use of this pattern, for instance in architectural approaches such as the Adaptive Object Model [YJ02] and Web personalization approaches such as [RFCS01].

Following the idea of the Rule-Object pattern, other design patters have ben proposed for modelling business rules in OO. In particular the *Encapsulated Business Rules* pattern proposes encapsulating the implementation of business rules for their incorporation in J2EE applications [Bel03]. Again, business rules are implemented in classes defining a condition and an action methods. The Observer pattern [GHJV95] is used to communicate changes in the problem state, i.e. changes in the information the business rules use. The triggering of rulesets is done by evaluators which observe and react to changes in the problem state. A hierarchy of evaluators can be defined to trigger different rulesets.

The Business Rule Beans (BRBeans) approach[4] aims at externalizing business rules from core applications [RDR$^+$00]. BRBeans is a framework which is part of IBM WebSphere Enterprise Edition[5]. It suggests extending business modeling and analysis to include the identification of tangled business rules and points of variability in core applications. At each of those points, the rule is externalized and the point of variability is represented as a *trigger point*, which is the mechanism to facilitate the dynamic attachment of the externalized rules. Rules in BRBeans are generic pieces of business logic which are written by programmers and stored in code libraries for future reuse (e.g. rules can be connected at different trigger points). A trigger point is a piece of code in a method that interfaces with the BRBeans runtime to attach and execute business rules dynamically during application execution. Thus, unlike our approach, rules and trigger points are low level. At runtime, when a trigger point is encountered the BRBeans produces the exact collection of rules that need to be fired. The trigger point then fires each rule in the collection and manages the aggregation of results. A GUI is provided for non-programmers in order to manipulate externalized business rules without the need for programming skills. However, this support is limited to deploying existing rule templates in new contexts or with different values (i.e. parameters), expiring existing rules or scheduling rules to become effective. However it does not allow defining completely new rules from scratch at the domain level. The BRBeans framework is capable of handling a collection of common patterns often encountered when producing rule-enabled applications. These patterns are deployed by means of invocations on APIs.

## 9.3 AOP for Business Rules

### 9.3.1 Decoupling Business Rules at Implementation Time

Business rules are today one of the most well-recognized examples of crosscutting concerns, as identified in well-know AOP books such as [Lad03]. Moreover, Tarr et al. [OT01] indicate how business rules can be separated using *HyperJ*, an AOP approach based on symmetric AOP. In that approach, business rules are encapsulated in different hyperslices, which are the HyperJ's modularization mechanism for crosscutting concerns. Hyperslices are loosely coupled with the base model, which implies that the business rules they encapsulate are reusable in different contexts. In this approach it is possible to specify a separate module (hypermodule) to encapsulate the details of how the business rules are linked to the core application. However, not much support for hyperslice relations is provided, limiting the combination of business rules. Moreover, mapping concerns is done statically, by matching structural units present in different hyperslices. This characteristic does not allow the connection of business rules to core application events that depend on the dynamics of an application.

Other related research focuses on the suitability of AOP for integrating object-oriented and rule-based programming languages [D'H04; DJ04]. In that work, hybrid aspects are proposed to achieve a very loose coupling between both paradigms at the program and the language level. This approach allows expressing both the rules and the core application functionality in their most appropriate paradigms, the rule-based and the object-oriented paradigms. Also, linguistic symbiosis is ensured between rule-based and object-oriented languages.

[4] More information at http://www.research.ibm.com/AEM/brb.html.

[5] Fully supported since version 4.0.

### 9.3.2 Decoupling Constraints at Design Time

Some existing approaches focus on the separation of crosscutting constraints at the design level [GBNT01; GBN99; Str00]. Constraints that specify global system properties (such as latency, precision, timing) crosscut the boundaries of the model hierarchy typically appear tangled and scattered in the base design [GBNT01; GBN99]. This is because typically the same constraint is repeatedly applied in many different places of the model with slight variations. In order to overcome these problems, constraints are described in a modular manner and woven into base designs using AOP.

When constraints are expressed at the modelling level by means of an OCL-like language, the responsibility to ensure the checking of those constraints at the implementation level is left to the programmer. The modular specification of constraints at the design level enables the automatic checking and enforcement of constraints in object-oriented applications. In [Str00] an approach for the automatic checking of crosscutting concerns at the implementation level is proposed. AspectJ code is automatically generated which checks the design constraints expressed at the design level.

## 9.4 Combining MDE and AOP

The use of MDE in this dissertation is threefold. MDE is used for generating the rule-objects implementing high-level rules, the aspects implementing high-level rule connections and the aspects realizing the implementation of some complex mappings for domain concepts. When looking at related work that attempts to combine MDE and AOSD, one mainly encounters approaches that focus on modelling crosscutting concerns explicitly as part of design models. For example, a possible direction is to extend a general purpose modelling language (e.g. UML) with explicit support for aspects. Within the MDE - AOSD research area, we classify approaches into two relevant categories: *top-down* and *bottom-up*. Top-down approaches aim at representing crosscutting concerns at the modelling level independently of the implementation model whereas bottom-up approaches have AOP as starting point (i.e. aspects at the implementation level) and aim at representing AOP concepts at the modelling level.

- *top-down:* Clarke et. al. extend UML to specify composition patterns [CW01] to explicitly capture reusable patterns of crosscutting behavior. Composition patterns are based on a combination of the subject-oriented model for composing separate and overlapping designs and UML templates. In order to use these patterns in a concrete application, the different *subjects* involved in the patterns need to be bound to concrete implementation entities in that application. This binding step is similar to our mapping. On the one hand, the binding language supported by the composition patterns approach is more expressive as it is similar to a pointcut language (e.g. support for wildcards is allowed for realizing mappings to many elements). On the other hand, subjects in patterns can only be bound to existing entities, and therefore support for derived and unanticipated mappings is not provided, whereas it is one of the contributions of our approach. Although composition patterns are independent of a concrete implementation model, it closely relates to the principles advocated by AOP. Thus, the authors show how the translation from composition patterns to AOP can be performed, in particular for the case of AspectJ as target AOP language. This is related to our approach in the sense that AOP is used as an implementation technique but its features are transparent at the high-level, i.e. AOP features are not exposed at the modelling level.

Other work in this category can be found in [SSR$^+$05; SRF$^+$05], where the authors present an MDD framework that incorporates ideas from AOSD. Separation of concerns is pursued at two levels, horizontal and vertical. At the PIM level, this approach defines a primary model that addresses the business logic of the application, as well as a set of generic aspect models, each of them describing a crosscutting feature in a generic way (horizontal SoC). A set of bindings determines how to compose the aspectual models with the primary model. An initial step for the weaving of primary and aspectual models consists of instantiating the aspect model on a concrete application domain. To this end, model elements are mapped to elements in the application domain. This mapping step can be compared to the mapping of domain entities presented in this dissertation. Moreover, this approach also addresses the transformation from PIMs to PSMs by means of separating the transformation process into separate transformations for the primary model and each of the aspect models (vertical SoC). This approach shows that the use of AOSD techniques can facilitate the separation of concerns and ease the modeling and model transformation specification tasks.

- *bottom-up:* the approaches in this category typically start from the aspect-oriented programming paradigm and attempt to extrapolate the AOP ideas at the modelling level. In general this is achieved by extending a general-purpose modelling language such as UML with constructs for representing aspects, pointcuts and advices and other AOP concepts. In [HJPP02] standard UML stereotypes are used to model aspects whereas in [PSD$^+$02] extensions of UML are proposed. They both differ from our approach in that the same AOP constructs that are present at the implementation level are exposed at the modelling level. Thus, at the modelling level, the use of AOP is not transparent. In our approach, writting or connecting a rule at the domain level does not reveal the underlying use of AOP. Another difference is that transformations in these approaches occur at the model level (i.e. the weaving occurs at the modelling level and not the code level).

  Another approach in this category is the ECL transformation language by Gray et al. [GLZ06] which can be used to model aspects that quantify the modeling elements that need to be transformed and apply the desired changes upon them, obtaining a new model at the same level of abstraction as the input model. Besides the fact that transformations occur at the modeling level, this approach differs from ours in that the modeler is in charge of specifying the desired modeling aspects, writing and varying the set of rules considered by the transformation engine. In our approach the modeler is unaware of the use of aspects. In addition, the set of transformations is part of the proposed framework, encapsulating expert knowledge on how rule connections are translated into aspect code.

The combination of AOSD and MDE has been the focus of several workshops organized at many international conferences in the last few years, among them the European Conference on Object-Oriented Programming (ECOOP) and the international conference on Aspect-Oriented Software Development (AOSD). Also, other work in this area can be found in [Völ05] where the author proposes several patterns that can be used to handle crosscutting concerns in the context of model-driven software development environment that range from using code generation templates as a simple means to separate concerns to using aspect-oriented modeling concepts to separate concerns in the models. The author also analyzes the relation between these two paradigms, points out their commonalities and differences and identifies scenarios where their combination can be useful.

## 9.5 Mapping Domain Knowledge To Implementation

Mapping domain knowledge to implementation has been the focus of many other areas of knowledge representation. For example, existing approaches focus on linking UML conceptual models to data models expressed in XML [RBG02]. These approaches have to deal with the extra complexity of linking two techniques which are aimed at tackling completely different problems: software design on the one hand and data modeling on the other hand. Unlike these approaches, our mappings do not have to deal with different paradigms, as both the domain and implementation models are developed in an object-oriented style.

All OO analysis and development techniques advocate building models that represent software systems at different levels of abstraction [FS03]. For example, when following the RUP process [JBR99], models are defined at the conceptual, specification and implementation levels. Moreover, OO methodologies are proposed for the refinements of models. For example, in [HBR00], refinements are defined for the mapping of UML models and Java code. The more recent MDE research stream aims at automating the mapping between models by means of model transformations [MB02]. However, this mapping is only made explicit at transformation time. When traceability is pursued (such as in RTE approaches), support is needed to make the link between models explicit even after the models have been transformed. Most existing RTE tools achieve this by including traces in the output models (e.g. in the form of annotations or extra documentation) to indicate which decisions were taken at transformation time. However, this is not sufficient to support full RTE [Pae06]. The MDE research community has therefore increased its focus on the explicit link between models which is currently an active research topic. An example of this is the *ECMDA Traceability Workshops* organized as part of the *European Conference on Model-Driven Engineering*[6].

Other domains where the mapping of domain knowledge is pursued are the Semantic Web and databases. In the context of the Semantic Web, interoperability among different ontologies is an essential issue that is gaining more and more attention. In order to achieve interoperability, mapping and merging of ontologies becomes a core question. Automatic or at least semi-automatic techniques have to be developed to reduce the burden of manual creation and maintenance of ontological mappings. These techniques rely on metadata about semantic similarities and appropriate rules. In [ES04] a methodology for identifying mappings between two ontologies is presented which is based on the intelligent combination of manually encoded rules. This approach only considers one-to-one mappings between single entities of the ontologies.

In [vdSH06], Semantic Web technology is used to support the exchange of user model data between applications. Assuming the user model schemas to be modeled in OWL, this approach translates instances of one schema into instances of a second schema. This translation is driven by mappings which are expressed in SWRL [HPSB$^+$04]. Currently these

[6] http://modelbased.net/ecmda-traceability/index.php

mappings need to be constructed with human interaction, although the authors envision the mappings to be done semi-automatically using schema-matching techniques.

In the context of databases, data integration and query mediation among distributed and heterogeneous sources have always been important issues. Also in this field, Semantic Web technology seems beneficial. For example, in [CTHB05] OWL is used to cope with structural and semantical heterogeneity between data sources, offering extra advantages over traditional relational languages thanks to OWL's rich semantics.

## 9.6 Business Rules in Specific Application Domains

In this section approaches that focus on the decoupling of business rules in the domains considered in this dissertation, e-commerce and SOA, are presented.

### 9.6.1 Business Rules in e-commerce Applications

One of the first initiatives that introduced the idea of Inference rules for e-commerce applications is CommonRules [IBM]. Following this initiative, the Rule Markup Initiative has taken steps towards defining a shared Rule Markup Language (RuleML) (presented in section 9.1.5). Some approaches have focused on the personalization of complex Web applications [RFCS01; RSG01a; KRS00; VTH06], focusing on constructing flexible and adaptable designs that can help coping with the increasing complexity of Web applications and simplify the process of adding personalization features to them. In [GGBH05] a solution to the lack of reusability of personalization specifications is provided which consists of the definition of a high-level rule language, the Personalization Rules Modeling Language (PRML). This language allows the specification of the personalization at design time which can then be mapped to different web design approaches. This specification is independent of functional application concerns and thus reusable.

### 9.6.2 QoS Business Rules in Service-Oriented Applications

The case study presented in this dissertation lets us combine two lines of research: on the one hand, research on the design of a high-level business rule language which is independent of the particularities of a specific domain; on the other hand, research on improving the flexibility of client-side service management. In SOA, to our knowledge, this combination has not been explored so far. Related state-of-the-art approaches focus on the design of either business rule languages to be used in real-world domains (e.g. financial, e-commerce), as presented earlier on in this chapter, or dedicated languages for expressing QoS constraints in service-oriented applications, as explored in the rest of this section.

Some existing research proposes dedicated languages to address the explicit specification of QoS requirements for web services. For instance, the GlueQoS approach is proposed [WTM$^+$04] which focuses on the dynamic reconciliation of QoS conflicts between interacting components. They propose a high-level and declarative language, based on WS-Policy [BIA03], to specify QoS features, preferences and conflicts. These high-level specifications involve elements specified in an ontology of QoS features. The high-level specifications are taken into account by a middleware-based mediator mechanism in charge of finding a compromise between the QoS of the communicating services. This way, QoS considerations are taken out of application logic and a better separation of concerns is achieved. The disadvantage of this approach is that the existence of a fixed ontology of features with all possible interactions is assumed. Contrary to our approach, it is not the focus of this approach to be able to define new domain concepts (e.g. new QoS properties) nor to analyze how concrete Web services map to those concepts. Thus, unanticipated features cannot be added at runtime.

Another related approach is AO4BPEL [CM04] which supports selective web service composition by modularizing business rules as aspects. In this approach, however, the core service composition description is tangled with the definition of the business rules, as they are specified as part of the same process description. Moreover, contrary to our approach, AO4BPEL rules are XML-based and thus low-level, implying the need of having programming skills.

## 9.7 Business Rules Methodologies, Vocabularies and Standards

An important issue in business rules concerns how to keep business rules at the business level inline with the rules that are implemented at the implementation level. Methodologies have been proposed which address the problem of incorporating business rules in information systems from the more general organizational point of view. For example, in [BK05] Bajec et al. suggest making the mapping between the business rule implementation and the business element (goal or vision) that represents the source of that business rule explicit.

In the area of business rules vocabularies, several approaches have been proposed. Some initiatives only focus at the business level and are not concerned with the technical realization of the vocabulary. For example, the Rulespeak [RL01] aims at expressing rules to improve the communication at the business level. This includes the problem of rule management, especially where changes to rules must be traced to and from the business side. The Semantics of Business Vocabulary and Business Rules (SBVR) [OMG05] is an OMG specification that defines the vocabulary and rules for documenting the semantics of business vocabulary, business facts, and business rules, as well as an XMI schema for the interchange of business vocabularies and business rules among organizations and between software tools. The SBVR is positioned to be entirely within the business model layer of the OMG's Model Driven Architecture (MDA). SBVR is targeted at business people rather than automated rules processing, and is designed to be used for business purposes, regardless of whether the rules could be automated in IT systems or not. In our approach this distinction is of course important as we are concerned with transforming business rules from their specification in terms of a domain model to their implementation. Note that business rule methodologies and vocabularies, although important, is outside the scope of this dissertation.

The Java Rule Engine API (JSR 94) [Jav], developed through the Java Community Process (JCP) program, represents the "least common denominator" in features across rule engines. JSR 94 defines a simple API to access a rule engine from a Java SE or Java EE client, including functionality for: registering and unregistering rules, parsing rules, inspecting rule metadata, executing rules, retrieving results, filtering results. Note that JSR 94 does not standardize the semantics of rule execution, i.e. the rule engine itself, the execution flow for rules, the language used to describe the rules, the deployment mechanism for Java EE technology. Efforts are under way to standardize a common rule language (e.g.

W3C is working on the Rule Interchange Format (RIF) — a format that allows rules to be translated between rule languages and thus transferred between rule systems[7] — and the OMG has started to work on a standard based on RuleML[8]).

[7] More information can be found at http://www.w3.org/TR/rif-ucr/.

[8] More information can be found at http://www.ruleml.org/.

# Chapter 10

# Conclusions

This chapter first summarizes the work presented in this dissertation while stressing our contributions (section 10.1). It continues with a discussion on future work (section 10.2).

## 10.1 Summary and Contributions

The goal of this dissertation is to achieve a highly flexible connection of high-level and executable business rules with existing object-oriented applications. To this end, ideas from AOSD and MDE have been recuperated and combined.

A first dimension of the work presented in this dissertation is motivated by a lack of support for the separation of concerns observed in current state-of-the-art approaches on developing object-oriented software applications with rule-based knowledge. Existing approaches advocate making rules explicit and separate from the object-oriented core functionality [vH01; Ros03; Dat00] in order to trace them to business policies and decisions, externalize them for a business audience, and evolve them. The need for a separation is increased by the fact that business rules do not necessarily change at the same pace as the core application functionality [Ars01]. However, we observe that existing approaches fail at separating the code that connects business rules with the core application. This connection code is essentially in charge of denoting the events at which rules are applied, capturing the required data and making it available for rule manipulation. Moreover, the connection code crosscuts the core application and is therefore tangled with code tackling other concerns and scattered among several modules of the core application. This occurs independently of the concrete approach used to represent the business rules (e.g. object-oriented patterns, rule-based languages, etc.).

With the aim of tackling the limitation previously described, we set out to decouple the crosscutting rule connection code. We propose encapsulating it in separate modules, decoupled from both the core application's functionality and the business rules. We consider the core application to be developed using standard object-oriented programming and the business rules to be decoupled and implemented using the *Rule-Object Pattern*. Our first three contributions are structured around the steps taken in decoupling business rule connections:

> **Contribution 1:** *Identification of technological requirements for achieving a highly flexible connection of business rules*
> General requirements are identified, described and motivated that are essential for any technology to be suitable to cleanly encapsulate the rule connection code

> and achieve high flexibility in the integration of the rules. These requirements are described independently of concrete implementation languages and/or technologies. Those requirements are also discussed in [CDJ03; CDS$^+$03; CDS$^+$05; CSD$^+$04]

A second natural step in the process of pursuing the decoupling of business rule connections consists of finding a suitable technology that accomplishes these identified requirements. Because AOP appears as a promising technique to cleanly encapsulate crosscutting code from object-oriented applications, we set out to tackle the decoupling of crosscutting rule connection code with AOP. This leads us to our second contribution:

> **Contribution 2:** *Analysis of the suitability of AOP for accomplishing the technological requirements identified in Contribution 1*
> On the one hand, we point out the fundamental AOP characteristics that are instrumental to successfully decoupling and modularizing crosscutting rule connections while accomplishing the identified technological requirements. We conclude that in general AOP is a suitable technology for achieving the decoupling of rule connections. On the other hand, a more concrete contribution is to show how the features supported by two representative AOP approaches — namely AspectJ and JAsCo — can be used to actually achieve the decoupling and modularization of crosscutting rule connections. The results of these concrete experiments are also reported in [CDJ03; CDS$^+$03; CDS$^+$05; CSD$^+$04]

These experiments let us observe that both languages succeed in achieving our goal, although for some of the requirements their solutions differ quite radically. This is because each approach provides different AOP mechanisms and features. However, we can observe that, independently of the concrete mechanisms and features, an AOP solution for decoupling rule connections follows a certain structure, in which we identify commonalities and variabilities that we abstract in aspect patterns. This constitutes a third contribution of this dissertation:

> **Contribution 3:** *Identification of aspect patterns and their variabilities in the implementation of rule connections*
> We identify several connection elements that recur in the implementation of rule connection aspects. Moreover, we observe how and under which specific circumstances these elements vary. Solutions for each of the connection elements and their variations are proposed, in particular implemented in JAsCo. Moreover, we observe that dependencies exist between these connection elements, which challenge their implementation. Thus, these dependencies need to be taken into account when combining solutions for each of these connection elements in order to obtain a complete and valid rule connection aspect. Note that although in this dissertation we opted for illustrating the proposed transformations using JAsCo as a target language, their implementation is not bound to the specific features of this particular technology. The identified patterns build on top of the common AOP characteristics (identified in Contribution 2) and therefore can be realized in any AOP approach that adheres to those characteristics. Thus, the proposed aspect patterns are generic since they do not depend on the concrete mechanisms and features of a specific approach. The connection elements and their variations are also discussed in [Cib02; CD06a; CDJ06b]. The identified dependencies are also presented in [CD06a].

A second dimension of our work is focused on facilitating the task of understanding and defining business rules by the domain experts. We observe that, when using state-of-the-art approaches, in order for rules to be executable they ultimately need to be implemented in a rule-based programming language, or expressed using design patterns or XML, etc. No matter which technical solution is followed, the need for having technical skills is implied, which excludes the domain expert. A second problem observed in this dissertation is that executable rules are expressed in terms of concrete implementation elements from the existing core application, which makes rules fragile and not reusable among applications of the same domain. To overcome these two problems, existing approaches propose the idea of "pulling business rules up" to a higher level of abstraction (as it can be found in JRules, QuickRules, VisualRules and HaleyRules). Following this idea, we set out to build a layer of abstraction that we call *domain model*, which allows expressing business rules in terms of domain concepts and enhances the current support found in existing approaches. We realize that in order for high-level rules to be executable and flexibly integrated in existing applications, their connections also needed to be expressed in terms of the domain. It is then observed that, whereas aspects are a good solution to the problem of decoupling rule connection code, they completely reside at the implementation level and therefore also exclude the domain expert. Several contributions are identified as part of this dimension, which we summarize as follows (contributions 4, 5, 6 and 7):

First of all, the innovation of the *domain model* approach presented in this dissertation in comparison to existing approaches is that not only business rules can be expressed at a higher level of abstraction but also their connections with the existing core application. This leads us to the next contribution:

> **Contribution 4:** *High-level dedicated languages are designed for expressing business rules and their connections in terms of domain concepts*
> The choice of constructs of our high-level languages is the result of extensive previous work [CDJ03; CDS$^+$03; CDS$^+$05; CSD$^+$04]. The presented high-level rule language raises the expressive power offered by existing approaches that support high-level business rules since it allows expressing rules in terms of domain entities that can in turn have very complex realizations at the implementation level. The idea of pulling the business rule connections up to the domain level has not been proposed before. The features of the high-level rule connection language presented in this dissertation are abstractions of the recurrent connection elements that conform the aspect patterns identified in Contribution 3.

Moving to a higher level of abstraction improves understandability as it becomes possible to express the rule and connection concerns in terms of the domain. Moreover, it becomes possible for the domain expert to add, modify and remove rules and rule connections.

We pursue high-level rules to be executable meaning that they can be directly integrated with the existing application. To this end, executable implementations are automatically obtained from the high-level specifications of rules and connections. Automatically generating these implementations allows the domain experts to remain oblivious to the low-level details. This is achieved by incorporating ideas from MDE, leading to the next contribution:

> **Contribution 5:** *Automatic and transparent transformations from the high-level specifications — of rules and connections — to implementation*
> Rules and rule connections expressed in the dedicated languages are transformed

into OOP programs (rule objects) and AOP programs (rule connection aspects). A detailed conceptual analysis of the challenges posed by these transformations is provided. These transformations are carried out automatically in the prototype implementation.

The use of AOP as target of the transformations allows us to keep the implementation of the rules and their connections well modularized and localized, without invasively changing the existing core application. Moreover, as the mapping from high-level specifications to their implementation is made explicit, rule and rule connection traceability becomes possible.

During this dissertation we were concerned with providing a flexible mechanism to deal with the inherent volatility of business rules. Because the underlying dynamic AOP language adopted is dynamic, the variability of existing rule connections, the definition of new connections for existing rules, the addition or removal of rules and connections can all occur at run time. Thus, it becomes possible to dynamically adapt the behavior of the core application by simply plugging in different sets of business rules, this way creating different versions of the same application.

It is also important to stress that, besides the clear advantages that these automatic transformations pose for domain experts, our approach also facilitates the task of the application engineer in charge of writing rule connection aspects. As mentioned before, manually implementing a rule connection aspect is tedious as several recurrent issues and dependencies need to be taken into account. Obtaining the rule connection code automatically from the high-level description of a rule connection removes the need for having to reason about these issues and dependencies every time such an aspect needs to be written from scratch.

As mentioned before, the idea of expressing rules in terms of domain concepts is already supported by existing state-of-the-art approaches. However, these approaches only support the definition of domain concepts that are simple aliases for implementation entities, which requires the existence of a one-to-one mapping between them. We observe that these one-to-one mappings are not enough to: i) represent domain concepts that have a more complex realization in terms of existing implementation entities, and ii) represent domain concepts that are not present whatsoever in the existing implementation. Thus, in order to tackle this limitation, we improve on the existing support for linking domain concepts to implementation by means of allowing the definition of more complex mappings. This lead us to our next three contributions:

**Contribution 6:** *A dedicated language for the definition of domain entities and their mappings*
This language allows defining mappings to complex navigational, arithmetical and logical expressions in terms of many implementation entities. Moreover, mappings can be completely defined at the domain level, in terms of existing domain entities of a domain model. High-level mappings can be defined by domain experts, as they do not require knowledge about the existing implementation. Moreover, special domain operators are provided as part of this mapping language which allow capturing crosscutting domain knowledge or introduce new knowledge that is unanticipated in the existing application. The implementation of these operators is based on AOP: aspects and/or connectors

are automatically generated as a result of using these operators in mapping specifications. The use of AOP is transparent to the domain experts. The automatic translation from mapping specifications to their internal representations in the domain model infrastructure is also supported. These internal mapping representations are consulted during the process of transforming the high-level rules and connections to code.

Our approach allows defining expressions as mappings for domain properties or operations. Moreover, it also allows writing those expressions directly in the conditions and actions of rules. This last option is advisable when those expressions are needed only in few rules. As a consequence, the domain model is not polluted with the definition of domain entities that are hardly used.

**Contribution 7:** *Identification of several use cases that show the power of the proposed mapping language*
Five use cases are presented which illustrate how the mapping language can be successfully used to realize the expression of domain knowledge. We show how a complete OO class can be pulled up to the domain level, how a single domain class can map to many OO classes following two different semantics, i.e. union and intersection, how the mapping for anticipated as well as unanticipated knowledge can be realized and how derived knowledge can be captured at different execution points.

A framework supporting the entire domain model presented in this dissertation has been implemented. Moreover, this framework has been validated in a non-trivial case study application, the Web Services Management Layer. We observe that, although the WSML enhances the overall service management, its customization is still inflexible since only anticipated and hardcoded business rules are considered and programming skills are still needed to add unanticipated business rules. We then show how our approach can enhance the customization of the WSML by focusing on two validation scenarios:

i We *evolve* the existing WSML by adding new configuration business rules at the high level.

ii We *refactor* existing selection policies from the core WSML implementation and express them and their connections at the high level.

We then conclude that both scenarios can be supported using the approach presented in this dissertation. Existing rules can be externalized and new rules can be added, enhancing the adaptability of service-oriented applications.

To conclude, we list these two last contributions of our dissertation:

**Contribution 8:** *The implementation of a prototype supporting the entire domain model infrastructure*

**Contribution 9:** *The evaluation of the presented domain model infrastructure in a non-trivial case study: the WSML*

## 10.2 Trade-offs and Future Work

In this section we present a discussion on trade-offs of our approach and possible directions of future work.

### 10.2.1 Modularity

A first issue in our approach is the importance of maintained modularity in the generated code and, related to that, the overhead of using AO technology in order to achieve this modularity. One could argue that in MDE, the generated code does not have to be modular since it is typically not regarded by humans. However, in our particular context, the generated code is integrated with existing code, which *is* most likely regarded by developers. Therefore, it is of utmost importance that the generated code that pertains to rules and their connections does not affect the existing source code in numerous places. The more mature AO approaches provide excellent tool support for showing the impact of aspects in a base application.

### 10.2.2 Scalability

One of the concerns with respect to scalability is the size of a particular domain model with respect to the number of domain entities. We have found that a set of business rules typically considers the same domain classes, even if the reference to some properties or operations on these classes may vary between the rules. Therefore, an initial effort is required for building the domain model, whereas a much smaller effort is required for adapting the domain model as new rules are added. In chapter 6 we discussed how mappings for domain entities can be defined. Our approach supports pulling up implementation entities to the domain level automatically. However, this can only be achieved in the case of perfect one-to-one mappings. In the case of more complex mappings, adaptations might be required for which input from someone knowledgeable about the implementation is needed. In these complex cases, the complete mapping automation is not possible. In any case, the mapping language presented in chapter 6 facilitates the task of defining the initial domain model and extending it accordingly.

A second important issue is the scalability of our approach with respect to the number of business rules. The domain model infrastructure allows adding new business rules and connections with no extra penalty. Of course, the more business rules the more overhead that is incurred in the system. This overhead is in relation to the number of aspects that are generated and need to be put in place for the application of the rules (section 10.2.3).

A last important issue that appears with numerous business rules is rule interference. This is a well-known problem in the field of rule-based knowledge and therefore it is an issue in all lightweight approaches to business rules and even in AI expert systems. The latter systems are more advanced than the former ones as they typically build on heuristics (e.g. recency, positioning) that determine the order of rule applicability. However, these systems are still fragile since adding a new rule can completely break consistency. Therefore these systems are not very scalable as in the worst case, adding a rule requires checking the relation between that new rule and the existing ones. Because our solution builds on AOP, we could recuperate ideas from the feature interaction research carried out in the AOP community and attempt to pull these ideas up to the domain level. In previous work we investigated current AO approaches and their support for combining rule connection aspects at the implementation level [CDS+03; CDJ03; CDS+05; CSD+04]. For example, JAsCo combination and precedence strategies appear useful to control the order and the way rule objects need to be invoked. We can envision extending the high-level languages with explicit constructs for specifying rule precedence and combination strategies and so on. However, with large amounts of rules, manually detecting and resolving dependencies is not scalable and thus alternative techniques might be required. This is subject of future work.

### 10.2.3 Aspect Issues

#### 10.2.3.1 Overhead

In our approach, a different aspect and connector exist per high-level business rule connection. Moreover, extra aspects and connectors are also generated for the realization of unanticipated domain entities that result crosscutting. Thus, the number of aspects grows proportionally to the number of rules and the number of unanticipated domain entities that are defined in the domain model. Of course managing these aspects implies an extra overhead. Dealing with this overhead largely depends on the efficiency of the underlying aspect technology. Most mature AO approaches have an acceptable performance overhead, especially if one knows which are the costly features to avoid.

#### 10.2.3.2 Interference

Another issue in relation to the use of aspects in our approach is aspect interference. Aspect interference can occur between different connection aspects generated by our approach. For instance, the execution of one connection aspect can undesirably prevent another connection aspect from triggering. This is difficult to tackle and control at the high-level as it involves having perfect understanding of the execution flow and dealing with other low-level issues. Moreover, when considering the existence of aspects in the core application, aspect interference can occur between those aspects and the connection aspects that are generated for high-level rule connections. Aspect interference is a well-known issue referred to as feature interaction problem [PSC+02; BMV02; NBA04; DFS02; KPRS01] which attracts the attention of the whole AOP community.

### 10.2.4 Transformations

Another direction of work which is in relation to the implementation of the transformations is the use of a dedicated state-of-the-art transformation language. By expressing our transformations in dedicated languages, a more modular implementation of those transformations could be obtained, and eventually, formal verification could also be an added value depending on the chosen transformation language. However, when using contemporary language development techniques, implementing complex transformations (such as the ones we encounter in this dissertation which are of the kind *local-to-global*, *global-to-local* and *global-to-global* transformations) is not a straightforward task. This is identified and tackled in the PhD work of Cleenewerck [Cle07]. In that work, the author observes that the transformation process is typically decomposed into a set of implicitly co-operating transformation modules which are hard to manage. These implicit dependencies between the transformations complicates the implementation of the transformation process as a whole. Thus, support for better separation of concerns in the implementation of the transformations is required.

### 10.2.5 Expressivity of High-level Languages

The choice of constructs of our high-level languages is the result of extensive work which has been published and presented in several conferences and articles [Cib02; CDJ03; CDS$^+$03; CDS$^+$05; CSD$^+$04]. As such we are able to express the business rules that we find in the applications of our industrial partners as well as the ones presented in books on business rules [vH01; Ros03]. Still, improvements can be envisioned in order to enhance the expressivity of the high-level languages. In this section, possible directions of work in this area are discussed.

#### 10.2.5.1 Temporal Rules

Temporal business rules, rules that have time-dependent conditions, cannot be expressed in the current high-level languages. However, initial research on temporal business rules has been carried out. Stateful aspects in JAsCo [VSCF05] have been investigated as an implementation solution, as they appear to be useful to capture the time dependencies between the different events referred to by the rules. In that work, temporal rules that capture behavioral patterns have been implemented as stateful aspects. In [CV05], a first categorization of temporal business rules is presented, focusing in particular on the domain of web service compositions, as well as proposing solutions in JAsCo. Furthermore, some research was conducted in order to sketch a classification of temporal business rules and temporal events and an analysis of the required technological features which are inspired in temporal logics was carried out. A possible continuation of this initial work consists of refining this categorization and incorporating these findings in the dedicated high-level languages. This would allow the expression of temporal business rules at the domain level. A required step in this direction implies identifying new aspect patterns for temporal rules and writing new transformations that would translate a high-level temporal rule specification into a stateful aspect.

#### 10.2.5.2 Collections

We envision extending the high-level business rule language with support for collections. Similarly to the way collections are supported in OCL, we can imagine extending the high-level rule language with special operators that would enable, for instance, checking whether a given entity is included in a collection, iterating through the elements of a collection, invoking a domain operation on all the elements of a collection, etc.

#### 10.2.5.3 Events

The current mapping language allows defining events by means of pointing to specific domain operations. However, having to explicitly refer to domain operations can be tedious in some cases. For instance, imagine a different event needs to be defined for each of the domain operations of a given domain class. With the current support, a different event definition is needed per domain operation in the domain class. To facilitate these definitions, a shortcut mechanism can be envisioned that would take a domain class as input and automatically define a different event per domain operation encountered in the domain class definition. Moreover, a pattern language can be envisioned — for instance, similar to the one supported by pointcut languages of existing AOP approaches [AOS05] — which would allow a more declarative way of specifying the set of domain operations on which events need to be defined. For example, we can envision support for wildcards or the specification patterns that need to be checked on the name or parameters of the domain operation, etc.

#### 10.2.5.4 Rule-Based Languages

Some applications have knowledge-intensive subtasks, such as (semi-)automatic scheduling, intelligent help desks and advanced support for configuring products and services, which require not only the specification of rule-based knowledge in an *if...then...* format, but also a rule engine that supports ordering and chaining of rules. This category of rule-based knowledge is considered in [D'H04], but not at the domain level. Following the idea of building a domain model on top of rules expressed in a rule-based language, a first initial analysis has been carried out in [CDJ06b] where we adapt the aspect patterns for the case of rules expressed in a rule-based language, in particular the Jess language [FH03]. Although the general set-up of the patterns is the same as the one presented here (for instance, with respect to the recurrent connection elements), other cases need to be considered which are inherent of the way rules are executed in a rule engine. Moreover, most likely the proposed transformations to aspect patterns would need to be adapted accordingly.

#### 10.2.5.5 Predefined Operators

As shown in chapter 6, aspects are generated for certain mappings that require capturing values at several points in the execution of the core application or adding new implementation. These mappings are expressed at the domain level, using special operators predefined in the domain model infrastructure. One could wonder at this stage whether these special operators could also be used in expressions that are directly included in the specification of a high-level rule. Using special operators directly in the rules would require having some kind of preprocessor in place which can iterate over all the rules defined in the system and deploy the necessary aspects for the realization of the operators used in the rules. This preprocessing step would ensure having all the necessary aspect infrastructure set up in order to make the needed information available at rule application time. This kind of support would contribute to avoid the pollution of the domain model with domain entities that are only needed in a few rules.

Another direction of work with respect to the special operators based on AOP is to extend the predefined library by means of adding operators for specific domains.

### 10.2.6 Raising the Level of Abstraction of AOP

The experiments carried out using AOP for the implementation of rule connection aspects (contribution 3) let us observe that the general purpose nature of the features supported by all analyzed AOP approaches are sometimes too low-level for the specific kinds of problems we need to address. This motivates the need for having higher-level abstractions on top of these general purpose AOP features. In particular, the proposed high-level connection language can be seen as a first attempt to raise the level of abstraction of common AOP constructs while keeping domain experts oblivious to the use of AOP. This idea could be extended in order to express other kinds of aspects at the higher level.

### 10.2.7 Mapping

Two directions of future work are identified and described with respect to the mappings for domain entities.

#### 10.2.7.1 MDE for Mapping Specification

While some mappings are defined completely at the domain level — in terms of other existing domain entities — some mappings still require knowledge about the concrete implementation, which complicates their definition. Although tool support has been provided as part of this dissertation with the aim of facilitating these mapping definitions, this support can still be improved. For example, a possible direction of future work can involve applying MDE ideas to the definition of the mapping. More concretely, the use of models could be investigated to simplify the creation of domain entities and their mappings. One possible way is to manipulate UML-style models defined at different levels of abstraction. Imagine a more conceptual model is defined which represents the application domain from the point of view of the domain expert, and a more detailed one reflecting a more concrete design of the existing implementation. Depending on the modelling formalism used, it might become possible to express the mappings completely at the modelling level. This way, mappings could then be described as model transformations.

#### 10.2.7.2 Mapping to AOP

In this dissertation we consider a core application implemented in OOP and therefore, the mappings presented in chapter 6 consider the case in which a domain entity is realized by one or more OO entities — i.e. a class, attribute, method, expression or interface. We can imagine the situation in which the core application is implemented using AOP. In this case, the domain knowledge can also be realized by aspects. A possible continuation of the work presented in this dissertation is to extend the analysis of the anticipated mappings by considering AOP entities that are existing in the core application. A first analysis has been carried out which is presented in [CDJ05; CDJ06a]. Moreover, the mapping language can be extended to support the new anticipated mappings to AOP.

A new issue appears when considering mappings to AOP (besides the issues described in section 10.2.3) which is in relation to the manipulation of aspect instances. The way aspect instances are manipulated in AOP is different to the way objects are manipulated in OOP. This difference has an impact on the interaction between a high-level rule and its corresponding high-level rule connection. In the current approach, a rule can define expected domain entities — specified in the *USING* clause — that are meant to be provided at rule connection time. This implies that, at rule connection time, the required entities are obtained from the context of the high-level events — involved in the rule connection definition — and made available to the rule. When domain entities that are realized by aspect entities are considered, the process of making the required information available to the rules might require retrieving and manipulating aspect instances. Because aspect instances are created and manipulated differently than objects, the current mechanism to obtain and pass the required objects to the rules needs to be adapted.

### 10.2.8 Quantification

In the current rule connection language, a rule connection specification involves one connection event only, at which the rule is actually triggered, and one or more capturing events, at which the extra information required for the application of the rule is obtained. We could imagine the case in which a rule needs to be applied at more than one connection event. In the current approach, different connection specifications are needed for each connection event. An improvement of the current connection language would consist of adding support for referring to more than one connection event in the same connection specification and modify the translation process accordingly. This extension would realize quantification in the true AOP sense. Supporting this extension however is not straightforward, as new issues need to be taken into account:

- *expressivity of the rule connection language*: if more than one connection event can be specified in the same connection specification, for each of those events, a specification of how the available information maps to the required information is needed which hampers the understandability of the connection specification as a whole. Imagine the situation in which the same rule is applied before the occurrence of N different heterogeneous events, as follows:

```
CONNECT BR1
BEFORE event1, event2
MAPPING event1.available_entity1 TO required_entity1
      event2.available_entity2 TO required_entity1
      ...
      eventN.available_entityN TO required_entity1
AFTER ...
INSTEAD OF ...
```

  The problem with this specification is that the mapping clause gets cluttered with the specification of how the required rule information has to be mapped to the available information for each of the events involved in the respective *BEFORE/AFTER* or *INSTEAD OF* connection.

  A possible solution that would improve understandability consists of splitting the definition of connection events as follows:

```
CONNECT BR1

BEFORE event1
MAPPING event1.available_entity1 TO required_entity1

BEFORE event2
MAPPING event2.available_entity2 TO required_entity1
....
BEFORE eventN
MAPPING eventN.available_entityN TO required_entity1
...
AFTER ...
INSTEAD OF ...
```

  These enhancements to the rule connection language are subject of future work.

- *translations to rule connection aspects*: in the previous example, consider the case where the `available_entity1` corresponds to `event1`'s target object whereas `available_entity2` corresponds to `event2`'s parameter (assuming there is one). The different ways in which the required information needs to be obtained from the context of the events makes it impossible to generate aspect advice code that is valid for all the connections of the same kind (i.e. *before*, *after* or *instead of*). Thus, for each *before* connection for instance, a different hook and advice needs to be generated; the same for the cases of *after* and *instead of* connections. We can however envision improving this translation process by performing local optimizations, e.g. if more than one *before* connection involves entities that need to be obtained in the same way, then a single advice and therefore a single hook can be generated for them. This is a possible future work direction.

An evaluation of the approach presented in this dissertation and the implemented tool support is currently undertaken using an industrial application from inno.com, one of our industrial partners, where numerous business rules need to be integrated in a Customer Information Tracking System of the Belgian Post. This system is mainly involved with processing customer requests to open accounts in the Bank of the Post. These requests need to follow a complex workflow of tasks. AOP is already used in this system to tackle crosscutting concerns like audit logging, performance logging, security (Spring/AOP). However, rule integration is carried out in a very naive way. The idea is to use our domain model infrastructure for expressing the connection of the business rules that are inherent to this workflow. Some example rules are for instance: task priority rules, rules that act on historical information about the customers and statistical rules based on logged information that might even be unanticipated in the current system.

# Appendix A

# High-Level Business Rule Language

## A.1 Non-Terminals

$$
\begin{array}{rcl}
< BR > & ::= & < BRIdentifier > \\
 & & [< Properties >] \\
 & & [< BODeclarations > \\
 & & [< BOProperties >]] \\
 & & < Condition > \\
 & & < Actions > \\
< BRIdentifier > & ::= & < BR >< Name > \\
< Properties > & ::= & < PROPS >< AliasList > \\
< BODeclarations > & ::= & < USING >< FilteredAliasList > \\
< BOProperties > & ::= & < WHERE >< PropDefList > \\
< Condition > & ::= & < IF >< ORCondition > \\
< Actions > & ::= & < THEN >< ActionBlock > \\
< AliasList > & ::= & < Alias > (< COMMA > Alias >)^{*} \\
< FilteredAliasList > & ::= & < FilteredAlias > (< COMMA >< FilteredAlias >)^{*} \\
< PropDefList > & ::= & < PropertyDefinition > (< COMMA > \\
 & & < PropertyDefinition >)^{*} \\
< ORCondition > & ::= & < ANDCondition > (< OR >< ANDCondition >)^{*} \\
< ActionBlock > & ::= & < ActionStatement > (< AND >< ActionStatement >)^{*} \\
< ActionStatement > & ::= & (< Invocation > \mid < Assignment >) \\
< Assignment > & ::= & (< Reference > \mid < Name >) < IS >< Value > \\
< FilteredAlias > & ::= & < Reference >< AS >< Name > [< Filter >] \\
< Filter > & ::= & < MATCHING >< Name > \\
< Alias > & ::= & < Reference >< AS >< Name >
\end{array}
$$

$$\begin{array}{rcl}
<PropertyDefinition> & ::= & <Name><IS>(<Invocation>\,|<Reference>) \\
<ANDCondition> & ::= & <XORCondition>(<AND><XORCondition>)^{*} \\
<Invocation> & ::= & <Reference><LPAR>[<ParList>]<RPAR> \\
<Reference> & ::= & <Name>(<DOT><Name>)^{*} \\
<XORCondition> & ::= & <NOTCondition> \\
 & & |<RelationalExpression> \\
 & & (<XOR><XORCondition>)^{*} \\
<ParList> & ::= & <Expression>(<COMMA><Expression>)^{*} \\
<Expression> & ::= & <ORCondition> \\
<RelationalExpression> & ::= & <Comparand> \\
 & & (<COMPARATOR><Comparand>)* \\
<NOTCondition> & ::= & <NOT><ORCondition> \\
<Comparand> & ::= & <Term>((<PLUS>\,|<MINUS>)<Term>)^{*} \\
<Term> & ::= & <Factor>(<MULTIPLIER><Term>)^{*} \\
<Factor> & ::= & <Value>(<POWER><Value>)^{*} \\
<Value> & ::= & <LPAR><ORCondition><RPAR> \\
 & & |<Invocation> \\
 & & |<Reference> \\
 & & |<Boolean> \\
 & & |<Name> \\
 & & |<Integer> \\
 & & |<Real> \\
 & & |<Literal> \\
<Boolean> & ::= & (<TRUE>\,|<FALSE>) \\
<Name> & ::= & <NAME> \\
<Integer> & ::= & (<INTEGER>\,|<MINUS><INTEGER>) \\
<Real> & ::= & (<REAL>\,|<MINUS><REAL>) \\
<Literal> & ::= & <LITERAL>
\end{array}$$

## A.2 Terminals

$$
\begin{aligned}
< BR > &::= \text{“}BR\text{”} \\
< PROPS > &::= \text{“}PROPS\text{”} \\
< USING > &::= \text{“}USING\text{”} \\
< WHERE > &::= \text{“}WHERE\text{”} \\
< AS &::= \text{“}AS\text{”} \\
< MATCHING > &::= \text{“}MATCHING\text{”} \\
< IS > &::= \text{“}IS\text{”} \\
< COMMA > &::= \text{“}, \text{”} \\
< QUOTE > &::= \text{“”}\,\text{”} \\
< LPAR > &::= \text{“}(\text{”} \\
< RPAR > &::= \text{“})\text{”} \\
< DOT > &::= \text{“}.\text{”} \\
< IF > &::= \text{“}IF\text{”} \\
< THEN > &::= \text{“}THEN\text{”} \\
< COMPARATOR > &::= \text{“} = \text{”}|\text{“} < \text{”}|\text{“} <= \text{”}|\text{“} > \text{”}|\text{“} >= \text{”} \\
< PLUS > &::= \text{“} + \text{”} \\
< MINUS > &::= \text{“} - \text{”} \\
< MULTIPLIER > &::= \text{“} * \text{”}|\text{“}/\text{”} \\
< POWER > &::= \text{“”} \\
< NOT > &::= \text{“}NOT\text{”} \\
< AND > &::= \text{“}AND\text{”} \\
< OR > &::= \text{“}OR\text{”} \\
< XOR > &::= \text{“}XOR\text{”} \\
< TRUE > &::= \text{“}true\text{”} \\
< FALSE > &::= \text{“}false\text{”} \\
< INTEGER > &:= [\text{“1”} - \text{“9”}](< DIGIT >)^{*} \\
< REAL > &::= (< DIGIT >) * \text{“.”}(< DIGIT >)^{*} \\
< NAME > &::= < LETTER > (< LETTER > | < DIGIT >)^{*} \\
< LITERAL > &::= < QUOTE > (< LETTER > | < DIGIT > |\text{“”})^{*} \\
&\quad\;\; < QUOTE > \\
< LETTER > &::= [\text{“„}, \text{“}a\text{”} - \text{“}z\text{”}, \text{“}A\text{”} - \text{“}Z\text{”}] \\
< DIGIT > &::= [\text{“0”} - \text{“9”}]
\end{aligned}
$$

## Appendix B

# High-Level Business Rule Connection Language

### B.1 Non-Terminals

$$
\begin{array}{rcl}
< BRL > & ::= & < BRConnection > \\
 & & [< CapturePoints >] \\
 & & [< Mappings >] \\
 & & [< Activation >] \\
< BRConnection > & ::= & < CONNECT >< NAME > \\
 & & [< PropSpecifications >] \\
 & & < ConnectionSpecification > \\
< PropSpecifications > & ::= & < PROPS >< PropSpecification > \\
 & & (< COMMA >< PropSpecification >)^* \\
< PropSpecification > & ::= & < Boolean > \\
 & & | < Literal > \\
 & & | < Integer > \\
 & & | < Real > \\
< Boolean > & ::= & (< TRUE > | < FALSE >) \\
< Literal > & ::= & < LITERAL > \\
< Integer > & ::= & (< INTEGER > | < MINUS >< INTEGER >) \\
< Real > & ::= & (< REAL > | < MINUS >< REAL >) \\
< ConnectionSpecification > & ::= & < Before > \\
 & & | < After > \\
 & & | < InsteadOf > \\
< Before > & ::= & < BEFORE >< NAME > \\
< After > & ::= & < AFTER >< NAME > \\
< InsteadOf > & ::= & < INSTEADOF >< NAME >
\end{array}
$$

$$\begin{array}{rcl}
< CapturePoints > & ::= & < CAPTURE >< AT >< CapturePoint > \\
 & & (< COMMA >< CapturePoint >)^{*} \\
< CapturePoint > & ::= & < NAME > \\
< Mappings > & ::= & < MAPPING >< MappedName > \\
 & & (< COMMA >< MappedName >) * \\
< MappedName > & ::= & < NAME >< DOT >< NAME > \\
 & & < TO >< NAME > \\
< Activation > & ::= & < ActivateWhile > \\
 & & | < ActivateNotWhile > \\
 & & | < ActivateBetween > \\
< ActivateWhile > & ::= & < ACTIVATE >< WHILE >< NAME > \\
< ActivateNotWhile > & ::= & < ACTIVATE >< NOT >< WHILE >< NAME > \\
< ActivateBetween > & ::= & < ACTIVATE >< BETWEEN >< NAME > \\
 & & < AND >< NAME >
\end{array}$$

## B.2 Terminals

$$
\begin{aligned}
<CONNECT> &::= \text{“}CONNECT\text{”}\\
<BEFORE> &::= \text{“}BEFORE\text{”}\\
<AFTER> &::= \text{“}AFTER\text{”}\\
<INSTEADOF> &::= \text{“}INSTEADOF\text{”}\\
<IS> &::= \text{“}IS\text{”}\\
<CAPTURE> &::= \text{“}CAPTURE\text{”}\\
<AT> &::= \text{“}AT\text{”}\\
<MAPPING> &::= \text{“}MAPPING\text{”}\\
<TO> &::= \text{“}TO\text{”}\\
<ACTIVATE> &::= \text{“}ACTIVATE\text{”}\\
<WHILE> &::= \text{“}WHILE\text{”}\\
<NOT> &::= \text{“}NOT\text{”}\\
<BETWEEN> &::= \text{“}BETWEEN\text{”}\\
<COMMA> &::= \text{“,”}\\
<QUOTE> &::= \text{“””}\\
<LPAR> &::= \text{“(”}\\
<RPAR> &::= \text{“)”}\\
<DOT> &::= \text{“.”}\\
<IF> &::= \text{“}IF\text{”}\\
<THEN> &::= \text{“}THEN\text{”}\\
<COMPARATOR> &::= \text{“}=\text{”}|\text{“}<\text{”}|\text{“}<=\text{”}|\text{“}>\text{”}|\text{“}>=\text{”}\\
<PLUS> &::= \text{“}+\text{”}\\
<MINUS> &::= \text{“}-\text{”}\\
<MULTIPLIER> &::= \text{“}*\text{”}|\text{“/”}\\
<POWER> &::= \text{“”}\\
<NOT> &::= \text{“}NOT\text{”}\\
<AND> &::= \text{“}AND\text{”}\\
<OR> &::= \text{“}OR\text{”}\\
<XOR> &::= \text{“}XOR\text{”}\\
<TRUE> &::= \text{“}true\text{”}\\
<FALSE> &::= \text{“}false\text{”}\\
<INTEGER> &:= [\text{“1”} - \text{“9”}](<DIGIT>)^*\\
<REAL> &::= (<DIGIT>) * \text{“.”}(<DIGIT>)^*\\
<NAME> &::= <LETTER> (<LETTER> \mid <DIGIT>)^*\\
<LITERAL> &::= <QUOTE> (<LETTER> \mid <DIGIT> \mid \text{“”})^*\\
&\phantom{::=} <QUOTE>\\
<LETTER> &::= [\text{“„}, \text{“}a\text{”} - \text{“}z\text{”}, \text{“}A\text{”} - \text{“}Z\text{”}]\\
<DIGIT> &::= [\text{“0”} - \text{“9”}]
\end{aligned}
$$

# Appendix C

# High-Level Mapping Language

## C.1 Non-terminals

$$
\begin{array}{rcl}
< MapSpec > & ::= & (< MapForContainedDomainEntity > \\
 & & \mid < DCInheritanceSpecification > \\
 & & \mid < MapForDC > \\
 & & \mid < EventDefinition >)^{*} \\
< MapForContainedDomainEntity > & ::= & < DC >< LBRACE > \\
 & & (< MapForContainedDM > \\
 & & \mid < MapForContainedDP >)^{*} \\
 & & < RBRACE > \\
< MapForContainedDM > & ::= & < DM >< MapForDM > \\
< MapForContainedDP > & ::= & < DP >< MapForDP > \\
< DCInheritanceSpecification > & ::= & < DC >< INHERITS >< DC > \\
< MapForDC > & ::= & (< MapForDC_{N}OTAlias > \\
 & & \mid < MapForDC_{A}lias >) \\
< MapForDC_{N}OTAlias > & ::= & < DC >< MAPTOIMPL > \\
 & & < MapFromDCToImpl > \\
< MapForDC_{A}lias > & ::= & < DC >< ALIASFOR > \\
 & & < MapFromDCToImpl > \\
< Reference > & ::= & < Name > (< DOT >< Name >)^{*} \\
< Name > & ::= & < NAME > \\
< Literal > & ::= & < LITERAL > \\
< DC > & ::= & < Name > \\
< DM > & ::= & < Name > (< DOT >< Name >) < LPAR > \\
 & & (< DMParamDeclaration > \\
 & & [< COMMA >< DMParamDeclaration >)^{*}] \\
 & & < RPAR >
\end{array}
$$

$$
\begin{array}{rcl}
< DP > & ::= & < Name > (< DOT >< Name >) \\
< DMParamDeclaration > & ::= & < Name > \\
< MapForDM > & ::= & (< MAPTOIMPL >< MapFromDMToImpl > \\
 & & \mid < MAPTODOMAIN >< MapFromDMToDomain >) \\
< MapForDP > & ::= & (< MAPTOIMPL >< MapFromDPToImpl > \\
 & & \mid < MAPTODOMAIN >< MapFromDPToDomain > \\
 & & \mid < MAPTOVALUE >< MapFromDPToValue >) \\
< MapFromDMToImpl > & ::= & < NavigationInImplModel > \\
< NavigationInImplModel > & ::= & (< Name > \mid < LPAR >< Reference >< RPAR >) \\
 & & < DOT >< TailReferencePathToImpl > \\
< NavigationInDomainModel > & ::= & < Name > \\
 & & < DOT > \\
 & & < TailReferencePathToDomain > \\
< TailReferencePathToImpl > & ::= & (< ReferenceToIM > \\
 & & \mid < ReferenceToAttribute >) \\
 & & (< DOT > \\
 & & (< ReferenceToIM > \\
 & & \mid < ReferenceToAttribute >))^{*} \\
< TailReferencePathToDomain > & ::= & (< ReferenceToDM > \\
 & & \mid < ReferenceToAttribute >) \\
 & & (< DOT > \\
 & & (< ReferenceToDM > \\
 & & \mid < ReferenceToAttribute >))^{*} \\
< ReferenceToIM > & ::= & < Name >< LPAR > \\
 & & [< IMParamList >] < RPAR > \\
< ReferenceToDM > & ::= & < Name >< LPAR > \\
 & & [< DMParamList >] < RPAR > \\
< ReferenceToAttribute > & ::= & < Name > \\
< IMParamList > & ::= & < IMParam > \\
 & & (< COMMA >< IMParam >)^{*} \\
< DMParamList > & ::= & < DMParam > (< COMMA >< DMParam >)^{*} \\
< IMParam > & ::= & < ParamType >< COLON >< ImplExpr > \\
< ParamType > & ::= & < Reference > [< LBRACKET >< RBRACKET >] \\
< DMParam > & ::= & < DomainExpr > \\
< ImplExpr > & ::= & < SimpleIMParamMap > \\
 & & [< Operator >< ImplExpr >] \\
< DomainExpr > & ::= & < SimpleDMParamMap > \\
 & & [< Operator >< DomainExpr >]
\end{array}
$$

$$\begin{aligned}
< SimpleIMParamMap > &::= (< NavigationInImplModel >\\
&\quad | < Variable >\\
&\quad | < FixedValue >\\
&\quad | < CompoundIMParamMap >)\\
< SimpleDMParamMap > &::= (< NavigationInDomainModel >\\
&\quad | < Variable >\\
&\quad | < FixedValue >\\
&\quad | < CompoundDMParamMap >)\\
< Variable > &::= < Name >\\
< FixedValue > &::= (< Literal > | < Integer > | < Boolean >)\\
< CompoundDMParamMap > &::= < LBRACE >< SimpleDMParamMap >\\
&\quad (< COMMA >< SimpleDMParamMap >)^*\\
&\quad < RBRACE >\\
< CompoundIMParamMap > &::= < LBRACE >< SimpleIMParamMap >\\
&\quad (< COMMA >< SimpleIMParamMap >)^*\\
&\quad < RBRACE >\\
< NonTerminalDomainExpr > &::= (< DomainExpr >\\
&\quad | < NavigationInDomainModel >)\\
< Boolean > &::= (< TRUE > | < FALSE >)\\
< Operator > &::= (< PLUS > | < COMPARATOR >)\\
< Integer > &::= (< INTEGER > | < MINUS >< INTEGER >)\\
< MapFromDCToImpl > &::= < Reference >\\
< MapFromDMToDomain > &::= < NonTerminalDomainExpr >\\
< MapFromDPToImpl > &::= < NavigationInImplModel >\\
< MapFromDPToDomain > &::= < NonTerminalDomainExpr >\\
< MapFromDPToValue > &::= < DC >< COLON >< Value >\\
< Value > &::= (< Literal > | < Integer > | < Boolean >)\\
< EventDefinition > &::= < EVENT >< Name >< AT >< DMForEvent >\\
&\quad < EXPOSING >\\
&\quad (< Parameter > | < Target > | < ReturnValue >)^*\\
< Parameter > &::= < PARAMETER >< ParamNumber >\\
&\quad < AS >< Name >\\
< Target > &::= < TARGET >< AS >< Name >\\
< ReturnValue > &::= < RETURNVALUE >< AS >< Name >\\
< DMForEvent > &::= < Name > (< DOT >< Name >)\\
&\quad < LPAR > [< DMParamDeclaration >\\
&\quad (< COMMA >< DMParamDeclaration >)^*]\\
&\quad < RPAR >\\
< ParamNumber > &::= < Integer >
\end{aligned}$$

## C.2 Terminals

$$\begin{aligned}
< MAPTOIMPL &::= \text{“}MAP-TO-IMPL\text{”}\\
< MAPTODOMAIN &::= \text{“}MAP-TO-DOMAIN\text{”}\\
< MAPTOVALUE &::= \text{“}MAP-TO-VALUE\text{”}\\
< EVENT &::= \text{“}EVENT\text{”}\\
< AS &::= \text{“}AS\text{”}\\
< AT &::= \text{“}AT\text{”}\\
< EXPOSING &::= \text{“}EXPOSING\text{”}\\
< PARAMETER &::= \text{“}PARAMETER\text{”}\\
< TARGET &::= \text{“}TARGET\text{”}\\
< RETURNVALUE &::= \text{“}RETURNVALUE\text{”}\\
< SEPARATOR &::= \text{“} >> \text{”}\\
< ARROW &::= \text{“} => \text{”}\\
< INHERITS &::= \text{“}INHERITS-FROM\text{”}\\
< ALIASFOR > &::= \text{“}ALIAS-FOR\text{”}\\
< AS &::= \text{“}AS\text{”}\\
< COMMA > &::= \text{“,”}\\
< QUOTE > &::= \text{“””}\\
< LPAR > &::= \text{“(”}\\
< RPAR > &::= \text{“)”}\\
< DOT > &::= \text{“.”}\\
< IF > &::= \text{“}IF\text{”}\\
< THEN > &::= \text{“}THEN\text{”}\\
< COMPARATOR > &::= \text{“} = \text{”}|\text{“} < \text{”}|\text{“} <= \text{”}|\text{“} > \text{”}|\text{“} >= \text{”}\\
< PLUS > &::= \text{“} + \text{”}\\
< MINUS > &::= \text{“} - \text{”}\\
< MULTIPLIER > &::= \text{“} * \text{”}|\text{“}/\text{”}\\
< POWER > &::= \text{“”}\\
< NOT > &::= \text{“}NOT\text{”}\\
< AND > &::= \text{“}AND\text{”}\\
< OR > &::= \text{“}OR\text{”}\\
< XOR > &::= \text{“}XOR\text{”}\\
< TRUE > &::= \text{“}true\text{”}\\
< FALSE > &::= \text{“}false\text{”}\\
< INTEGER > &:= [\text{“1”} - \text{“9”}](< DIGIT >)^*\\
< REAL > &::= (< DIGIT >) * \text{“.”}(< DIGIT >)^*\\
< NAME > &::= < LETTER > (< LETTER > | < DIGIT >)^*\\
< LITERAL > &::= < QUOTE > (< LETTER > | < DIGIT > | \text{“”})^*\\
&\phantom{::=} < QUOTE >\\
< LETTER > &::= [\text{“„}, \text{“}a\text{”} - \text{“}z\text{”}, \text{“}A\text{”} - \text{“}Z\text{”}]\\
< DIGIT > &::= [\text{“0”} - \text{“9”}]
\end{aligned}$$

# Bibliography

[AA01] A. Arsanjani and J. Alpigini. Using grammar-oriented object design to seamlessly map business models to component-based software architectures. In *Proceedings of the International Symposium of Modelling and Simulation*, pages 186–191, 2001. 24

[ACD$^+$03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, et al. *Business Process Execution Language for Web Services (BPEL4WS), version 1.1.* IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems, 2003. Available at: http://www-128.ibm.com/developerworks/library/specification/ws-bpel/. 171

[AOS05] AOSD-europe: European Network of Excellence on Aspect-Oriented Software Development. *Survey of Aspect-oriented Languages and Execution Models*, 2005. Available at: http://www.aosd-europe.net/. 39, 234

[Ars01] A. Arsanjani. Rule pattern language 2001: A pattern language for adaptive manners and scalable business rule design and construction. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS '01)*, page 370, Washington, DC, USA, 2001. IEEE Computer Society. 2, 7, 26, 103, 218, 227

[BA01] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001. 9, 38

[Bal05] M. Ball. *OO jDREW: Design and Implementation of a Reasoning Engine for the Semantic Web*, 2005. Honours Thesis Project Report. 212

[BC$^+$04] T. Bellwood, S. Capell., et al. *Universal Discovery, Discovery, and Integration (UDDI)*, 2004. Specification version 3.02. 171

[Bel03] C. Belderrain. Message-driven beans and encapsulated business rules. Article at The Server Side, Available at: http://www.theserverside.com/tt/articles/article.tss?l=RuleBasedMDB, 2003. 218

[BHW97] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: simplifying the development of numerical software. pages 353–372, 1997. 102

[BIA03] BEA, IBM, and Microsoft S. AG. *Web Services policy framework*, 2003. 223

[BK05] M. Bajec and M. Krisper. A methodology and tool support for managing business rules in organisations. *Information Systems*, 30(6):423–443, 2005. 224

[BMV02] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE '02)*, pages 110–127, London, UK, 2002. Springer-Verlag. 55, 233

[Bol03] H. Boley. Object-Oriented RuleML: User-level roles, URI-grounded clauses, and order-sorted terms. In *Second International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML'03)*, pages 1–16, 2003. 212, 213

[BR00] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)*, pages 73–87, New York, NY, USA, 2000. ACM Press. 121

[BRG01] *Defining Business Rules: What Are They Really?*, 2001. Business Rule Group, http://www.businessrulesgroup.org/. 1, 22

[BV] J. Bonér and A. Vasseur. AspectWerkz: a dynamic, lightweight and high-performant AOP/AOSD framework for Java. Available at: http://aspectwerkz.codehaus.org. 9, 38, 39, 44

[cac02] The Adaptive Web (special issue). *Communications of the ACM*, 45(5), 2002. 23

[CC$^+$03] E. Christensen, F. Curbera, et al. *Web Services Description Language (WSDL)*. W3C Web Services Activity, 2003. Specification version 1.2., W3C Technical Document. 171, 174

[CD03] M. A. Cibrán and M. D'Hondt. Composable and reusable business rules using AspectJ. In *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT) at the International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, United States, March 2003. 43

[CD05] M. A. Cibrán and M. D'Hondt. Towards automatic integration of high-level business rules using aspect-oriented programming. In *Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD at the European Conference on Object-Oriented Programming (ECOOP'05)*, Glasgow, United Kingdom, July 2005. 12

[CD06a] M. A. Cibrán and M. D'Hondt. A Slice of MDE with AOP: Transforming High-Level Business Rules to Aspects. In *International Conference on Model Driven Engineering Languages and Systems (MODELS'06)*, Genoa, Italy, October 2006. LNCS Springer. 11, 12, 16, 65, 228

[CD06b] M. A. Cibrán and M. D'Hondt. Explicit high-level rules for the customization of web services management. In *International Conference on Objects, Aspects, Services, the Web (NODe'06)*, Erfurt, Germany, September 2006. 18

[CD06c] M. A. Cibrán and M. D'Hondt. High-level specification of business rules and their crosscutting connections. In *8th International Workshop on Aspect-Oriented Modeling at the 5th International Conference on Aspect-Oriented Programming (AOSD'06)*, Bonn, Germany, 2006. 12, 16

[CDJ03] M. A. Cibrán, M. D'Hondt, and V. Jonckers. Aspect-oriented programming for connecting business rules. In *Proceedings of the 6th International Conference on Business Information Systems (BIS'03)*, 2003. 7, 9, 10, 11, 29, 43, 228, 229, 233, 234

[CDJ05] M. A. Cibrán, M. D'Hondt, and V. Jonckers. Mapping high-level business rules to and through aspects. In *2ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*, Lille, France, September 2005. 12, 15, 16, 18, 236

[CDJ06a] M. A. Cibrán, M. D'Hondt, and V. Jonckers. Mapping high-level business rules to and through aspects. *L'Objet*, 12(2-3), 2006. 12, 15, 16, 18, 236

[CDJ06b] M. A. Cibrán, M. D'Hondt, and Viviane Jonckers. Design of the hybrid aspect languages model for integrating concerns implemented in languages of different paradigms. Technical Report Project IWT 040116, Workpackage 1 - Deliverable 1.3b, System and Software Engineering Lab, May 2006. 65, 228, 235

[CDS$^+$03] M. A. Cibrán, M. D'Hondt, D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo for linking business rules to object-oriented software. In *Proceedings of International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'03)*, 2003. 7, 9, 10, 11, 29, 43, 228, 229, 233, 234

[CDS$^+$05] M. A. Cibrán, M. D'Hondt, D. Suvée, W. Vanderperren, and V. Jonckers. Linking business rules to object-oriented software using JAsCo. *Journal of Computational Methods in Sciences and Engineering (JCMSE)*, 5(1):13–27, 2005. 7, 9, 10, 11, 29, 43, 228, 229, 233, 234

[Che76] P. Pin-Shan Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976. 86

[Cib02] M. A. Cibrán. Using aspect-oriented programming for connecting and configuring decoupled business rules in object-oriented applications. Master's thesis, Vrije Universiteit Brussel, Belgium, 2002. 7, 9, 10, 11, 29, 43, 65, 228, 234

[Cle07] T. Cleenewerck. *Modularizing Language Constructs: A Reflective Approach.* PhD thesis, Vrije Universiteit Brussel, Belgium, 2007. 101, 102, 233

[CM04] A. Charfi and M. Mezini. Hybrid web service composition: business processes meet business rules. In *Proceedings of the 2nd international conference on Service oriented computing (ICSOC'04)*, pages 30–38. ACM Press, November 2004. 224

[CSD$^+$04] M. A. Cibrán, D. Suvée, M. D'Hondt, W. Vanderperren, and V. Jonckers. Integrating Rules with Object-Oriented Software Applications using Aspect-Oriented Programming. In *Proceedings of the Argentine Conference on Computer Science and Operational Research (ASSE'04)*, Córdoba, Argentina, 2004. 7, 9, 10, 43, 228, 229, 233, 234

[CTHB05] P-A. Champin, P. Thiran, G-J. Houben, and J. Broekstra. Exposing relational data on the semantic web with CROSS. Technical Report RR-LIRIS-2005-016, LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2, Ecole Centrale de Lyon, December 2005. 223

[CV03] M. A. Cibrán and B. Verheecke. Modularizing web services management with AOP. In *1st European Workshop on Object-Orientation and Web Services at the European Conference on Object-Oriented Programming (ECOOP'03)*, Darmstadt, Germany, July 2003. 170

[CV05] M. A. Cibrán and B. Verheecke. Dynamic business rules for web service composition. In *Proceedings of Workshop on Dynamic Aspects at the 4rd International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, United States, March 2005. 234

[CVJ03] M. A. Cibrán, B. Verheecke, and V. Jonckers. Modularizing client-side web service management aspects. In *Proceedings of Second Conference on Web Services*, volume 8, pages 1–12, Växjo, Sweden, November 2003. Växjo University Press, Series: Mathematical Modelling in Physics, Engineering and Cognitive Sciences. 169

[CVV$^+$07] M. A. Cibrán, B. Verheecke, W. Vanderperren, D. Suvée, and V. Jonckers. Aspect-oriented programming for dynamic web service selection, configuration and management. *World Wide Web Journal (WWWJ)*, 9, 2007. viii, 17, 169, 172, 175, 177

[CW01] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering (ICSE'01)*, pages 5–14, 2001. 220

[Dat00] C. Date. *What not How: The Business Rules Approach to Application Development*. Addison-Wesley, 2000. 2, 218, 227

[DC02] M. D'Hondt and M. Cibrán. Domain knowledge as an aspect in object-oriented software applications. In *Workshop on Knowledge-Based Object-Oriented Software Engineering (KBOOSE) at the European Conference on Object-Oriented Programming (ECOOP'02)*, 2002. 4

[Der06] D. Deridder. *A Concept-Centric Environment for Software Evolution in an Agile Context*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2006. 129

[DFS02] R. Douence, P. Fradet, and M. Sudholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE'02)*, pages 173–188, London, UK, 2002. Springer-Verlag. 55, 233

[D'H04] M. D'Hondt. *Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2004. 1, 3, 219, 235

[Die03] J. Dietrich. *The Mandarax Manual*, 2003. Available at: http://mandarax.sourceforge.net/. 214

[Dij76a] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 2

[Dij76b] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 8, 37

[DJ04] M. D'Hondt and V. Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, 2004. 219

[DMW99] M. D'Hondt, W. De Meuter, and R. Wuyts. Using reflective logic programming to describe domain knowledge as an aspect. In *First Symposium on Generative and Component-Based Software Engineering*, 1999. 4

[EFB01] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001. 7, 37

[ES04] M. Ehrig and Y. Sure. Ontology mapping - an integrated approach. In *Proceedings of the First European Semantic Web Symposium*, volume 3053 of *Lecture Notes in Computer Science*, pages 76–91, Heraklion, Greece, May 2004. Springer Verlag. 222

[Fab05] J. Fabry. *Modularizing Advanced Transaction Management.* PhD thesis, Vrije Universiteit Brussel, Belgium, 2005. 65, 87

[FC05] J. Fabry and T. Cleenewerck. Aspect-oriented domain specific languages for advanced transaction management. In *International Conference on Enterprise Information Systems (ICEIS'05)*, pages 428–432, 2005. 87

[FH03] E. J. Friedman-Hill. *Jess 6.1, The Rule Engine for the Java Platform.* Sandia National Laboratories, 2003. User Guide. 2, 235

[FHK$^+$05] N. E. Fuchs, S. Höfler, K. Kaljurand, F. Rinaldi, and G. Schneider. Attempto controlled english: A knowledge representation language readable by humans and machines. In *Reasoning Web*, pages 213–250, 2005. 213

[FR03] M. Fleury and F. Reverbel. The JBoss extensible server. In Markus Endler and Douglas Schmidt, editors, *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 344–373. Springer-Verlag, 2003. 9, 38, 39, 44

[FS03] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, 2003. 222

[Fuc97] M. Fuchs. Domain specific languages for ad hoc distributed applications. In *Proceedings USENIX Conference on Domain-Specific Languages*, pages 27–36, 1997. 87

[GBN99] J. Gray, T. Bapty, and S. Neema. Aspectifying constraints in model integrated computing. In *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, Denver, Colorado, USA, 1999. 220

[GBNT01] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, 2001. 220

[GGBH05] I. Garrigos, J. Gomez, P. Barna, and G-J. Houben. A reusable personalization model in Web application design. In *International Workshop on Web Information Systems Modeling (WISM'05) at the International Conference on Web Engineering*, 2005. 223

[GH$^+$03] M. Gudgin, M. Hadley, et al. *SOAP Version 1.2 Part 1: Messaging Framework*, 2003. W3C Recommendation. 171

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 26, 29, 149, 218

[GLC99] B. N. Grosof, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: courteous logic programs in xml. In *ACM Conference on Electronic Commerce*, pages 68–77, 1999. 24

[GLZ06] J. Gray, Y. Lin, and J. Zhang. Automating change evolution in Model-Driven Engineering. *Computer*, 39(2):51, 2006. 221

[Gmb06] Innovations Softwaretechnologie GmbH. VisualRules 3.5.1. Manual, 2006. Available at: http://www.visual-rules.de/en/pdf_en/vr_handbuch_en.pdf. viii, ix, 208, 209

[Gru93] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. 86

[Hala] The Haley Enterprise Inc. *Café Rete*. Available at: http://www.haley.com. 204

[Halb] The Haley Enterprise Inc. Haleyrules. http://www.haley.com/products/HaleyRules.html. 2, 15, 91, 94, 204

[Hal01] T. Halpin. *Information modeling and relational databases: from conceptual analysis to logical design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. 86

[Hal05] The Haley Enterprise Inc. *Methods for Managing Business Rules with HaleyAuthority*, 2005. Technical White Paper. viii, 205, 206

[HBR00] W. Harrison, C. Barton, and M. Raghavachari. Mapping UML designs to Java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'00)*, pages 178–187, New York, NY, USA, 2000. ACM Press. 222

[Hir06] D. Hirtle. TRANSLATOR: A TRANSlator from LAnguage TO Rules. In *Proceedings of the Canadian Symposium on Text Analysis (CaSTA'06)*, New Brunswick, Canada, 2006. 213

[HJPP02] W. Ho, J. Jézéquel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 99–105, New York, NY, USA, 2002. ACM Press. 221

[HL95] W. L. Hürsch and C. Lopes. Separation of concerns. Technical report, North Eastern University, 1995. 2

[HPSB$^{+}$04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3c member submission, World Wide Web Consortium, 2004. 212, 222

[Hud96] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996. 86

[HUS03] S. Hanenberg, R. Unland, and A. Schmidmeier. AspectJ idioms for aspect-oriented software construction. In *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPLoP'03)*, 2003. 63

[IBM] IBM. *CommonRules*. Web Site of IBM Research, http://www.research.ibm.com/rules/commonrules-overview.html. 21, 223

[ILO] ILOG. *JRules*. http://www.ilog.com/products/jrules/. 2, 14, 15, 91, 94, 201

[ILO04] ILOG. JRules 4.6. Technical White Paper, 2004. viii, 203, 204

[ILO06] ILOG. JRules 6: Deploying rule applications. Technical White Paper, 2006. Available at: http://www.ilog.com/products/jrules/. 202

[Inn] Innovations Softwaretechnologie GmbH. Visual rules. http://www.visual-rules.de. 15, 91, 207

[J$^{+}$] R. Johnson et al. *Spring/AOP*. The Spring Framework - Reference Documentation (chapter 6). Available at: http://www.springframework.org/docs/reference/aop.html. 9, 38

[Jav] Java Community Process. *JSR 94: JavaTM Rule Engine API*. Specification available at: http://jcp.org/en/jsr/detail?id=94. 211, 215, 224

[JBoa] JBoss. *Drools 3.0.6 Documentation*. Available at: http://labs.jboss.com/portal/jbossrules/docs. ix, 211, 212

[JBob] JBoss. *JBoss Rules*. http://www.jboss.com/products/rules. 2, 15, 91, 94, 209

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 222

[Ker05] M. Kersten. AOP@Work: AOP tools comparison. *developerWorks*, 2005. Available at: http://www-128.ibm.com/developerworks/library/j-aopwork1/. 44

[KHH$^{+}$01] G. Kiczales, E. Hilsdale, JJ. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, 2001. 8, 9, 10, 37, 38, 43

[KLM$^{+}$97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 220–242. Springer-Verlag, 1997. 4, 8, 30, 38

[KPRS01] H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE '00)*, pages 57–69. Springer-Verlag, 2001. 55, 233

[KR03] V. Kulkarni and S. Reddy. Separation of concerns in Model-Driven Development. *IEEE Software*, 20(5):64–69, 2003. 100

[KRS00] G. Kappel, W. Retschitzegger, and W. Schwinger. Modeling customizable Web applications. In *Proceedings of the Kyoto International Conference on Digital Libraries*, page 387, 2000. 21, 24, 223

[KSP04] J. Koskinen, A. Salminen, and J. Paakki. Hypertext support for the information needs of software maintainers. *Journal of Software Maintenance and Evolution*, 16(3):187–215, 2004. 121

[Lad03] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003. 219

[Leh96] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996. 139

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical report, Northeastern University, 1999. NU-CCS-99-01 Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html. 40

[LLO03] K. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations: Combining modules and aspects. *British Computer Society Journal (Special issue on AOP)*, 45(5):542–565, 2003. 40

[LOO01] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001. 9, 38

[MB02] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Ivar Jacoboson. 222

[MCF03] S. J. Mellor, A. N. Clark, and T. Futagami. Guest editors' introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18, 2003. 5, 13, 84

[MCG05] T. Mens, K. Czarnecki, and P. Van Gorp. A taxonomy of model transformations. In *Language Engineering for Model-Driven Software Development*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. 13, 87

[Min81] M. Minsky. A framework for representing knowledge. In J. Haugeland, editor, *Mind Design: Philosophy, Psychology, Artificial Intelligence*, pages 95–128. MIT Press, Cambridge, MA, 1981. 85

[MO03] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03).* ACM Press, 2003. 40

[MR97] N. Medvidovic and D. S. Rosenblum. Domains of concern in software architectures and architecture description languages. In *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages*, 1997. 87

[NBA04] I. Nagy, L. Bergmans, and M. Aksit. Declarative aspect composition. In *2nd Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT) at the International Conference on Aspect-Oriented Software Development (AOSD'04)*, 2004. 55, 233

[Nei84] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, 1984. 11

[New82] A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982. 85

[NI01] M. E. Nordberg III. Aspect-oriented dependency inversion. In *Workshop on Advanced Separation of Concerns, Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2001. 40

[OMG03] OMG. UML 2.0 OCL specification, 2003. Available at: www.omg.org. 91, 93

[OMG05] OMG. Semantics of business vocabulary and business rules (SBVR). Adopted specification, 2005. Available at http://www.omg.org/docs/dtc/06-08-05.pdf. 224

[OT01] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, October 2001. 8, 9, 37, 38, 39, 219

[Pae06] E. Van Paesschen. *Advanced Round-Trip Engineering: An Agile Analysis-driven Approach for Dynamic Languages.* PhD thesis, Vrije Universiteit Brussel, Belgium, 2006. 222

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. 2

[PE00] M. Perkowitz and O. Etzioni. Towards adaptive Web sites: conceptual framework and case study. *Artificial Intelligence*, 118(1-2):245–275, 2000. 24

[PG03] M. P. Papazoglou and D. Georgakopoulos. Introduction. *Communications of the ACM*, 46(10):24–28, 2003. 1

[PSC$^+$02] E. Pulvermueller, A. Speck, J. Coplien, M. D'Hondt, and W. De Meuter. Feature interaction in composed systems. In *Proceedings of the Workshops on Object-Oriented Technology (ECOOP'01)*, pages 86–97, London, UK, 2002. Springer-Verlag. 55, 233

[PSD$^+$02] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, L. Martelli, and F. Legond-Aubry. A UML notation for aspect-oriented software design. 1st AOSD Workshop on Aspect-Oriented Modelling with UML at the 1st International Conference on Aspect-Oriented Software Development (AOSD'02), April 2002. 221

[PSDF01] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION'01)*, pages 1–24. Springer-Verlag, 2001. 39, 44

[RBG02] N. Routledge, L. Bird, and A. Goodchild. UML and XML schema. In *ADC '02: Proceedings of the 13th Australasian database conference*, pages 157–166, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc. 222

[RDR$^+$00] I. Rouvellou, L. Degenaro, K. Rasmus, D. Ehnebuske, and B. McKee. Extending business objects with business rules. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'00)*, pages 238–249. IEEE Computer Society, 2000. 2, 21, 219

[RFCS01] G. Rossi, A. Fortier, J. Cappi, and D. Schwabe. Seamless personalization of e-commerce applications. In *Proceedings of the 2nd International Workshop on Conceptual Modeling Approaches for e-Business at the 20th International Conference on Conceptual Modeling (ER'01)*, Yokohama, Japan, 2001. Springer-Verlag. 21, 29, 218, 223

[RGI75] D. T. Ross, J. B. Goodenough, and C. A. Irvine. Software engineering: Process, principles, and goals. *Computer*, May 1975. 12

[RL01] R. G. Ross and G. S. W. Lam. RuleSpeak sentence templates - developing rule statements using sentence patterns. Technical report, version 1.0, Business Rule Solutions, LLC, 2001. 224

[Ros03] R. G. Ross. *Principles of the Business Rule Approach.* Addison-Wesley, 2003. 2, 7, 227, 234

[RSG01a] G. Rossi, D. Schwabe, and R. Guimaraes. Designing personalized Web applications. In *World Wide Web*, pages 275–284, 2001. 21, 29, 223

[RSG01b] G. Rossi, D. Schwabe, and R. Guimaraes. Designing personalized web applications. In *Proceedings of the 10th international conference on World Wide Web (WWW '01)*, pages 275–284, New York, NY, USA, 2001. ACM Press. 24

[Rul] The Rule Markup Initiative. *RuleML.* http://www.ruleml.org/. 212

[SAA$^+$00] A. Schreiber, J. M. Akkermans, A. A. Anjewierden, R. de Hoog, N. R. Shadbolt, W. Van de Velde, and B. J. Wielinga. *Knowledge Engineering and Management: The CommonKADS Methodology.* MIT Press, 2000. 1, 6, 85

[Sch06] D. C. Schmidt. Guest editors' introduction: Model-driven engineering. *IEEE Computer*, 39(2), February 2006. 85, 87

[SK03] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. 13, 84

[SKR99] J. B. Schafer, J. A. Konstan, and J. Riedi. Recommender systems in e-commerce. In *ACM Conference on Electronic Commerce*, pages 158–166, 1999. 24

[SRF$^+$05] D. Simmonds, R. Reddy, R. B. France, S. Ghosh, and A. Solberg. An aspect oriented model driven framework. In *EDOC*, pages 119–130. IEEE Computer Society, 2005. 221

[SSR$^+$05] A. Solberg, D. Simmonds, R. Reddy, S. Ghosh, and R. B. France. Using aspect oriented techniques to support separation of concerns in model driven development. In *COMPSAC (1)*, pages 121–126. IEEE Computer Society, 2005. 221

[Str00] R. Van Der Straeten. Using and enforcing constraints in object-oriented application development, 2000. 220

[SVJ03] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of 2nd International Conference on Aspect-Oriented Software Development*. ACM Press, 2003. 9, 10, 38, 39, 40, 43

[Szy05] C. Szyperski. *Components and Web Services*, 2005. Software Development, Beyond Objects column, Vol 9, Issue 8. 171

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 107–119. IEEE Computer Society Press / ACM Press, 1999. 8, 38, 39

[TPE$^+$02] V. Tosic, B. Pagurek, B. Esfandiari, K. Patel, and W. Ma. *WSOL - Web Service Offerings Language*, pages 57–67. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002. 174

[VC04] B. Verheecke and M. A. Cibrán. Dynamic aspects for web service management. In *Dynamic Aspect Workshop (DAW) at the 3rd International Conference on Aspect Oriented Software Development (AOSD'04)*, Lancaster, UK, March 2004. 169

[VC05] B. Verheecke and M. A. Cibrán. Dynamic aspects in large scale distributed applications: An experience report. In *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT) at the International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, United States, March 2005. 169

[VCJ03] B. Verheecke, M. A. Cibrán, and V. Jonckers. AOP for dynamic configuration and management of web services in client-applications. In *Proceedings of the International Conference on Web Services - Europe (ICWS'03-Europe)*, Erfurt, Germany, September 2003. Springer-Verlag. 169

[VCJ04] B. Verheecke, M. A. Cibrán, and V. Jonckers. Aspect-oriented programming for dynamic web service monitoring and selection. In *European Conference on Web Services 2004 (ECOWS'04)*, Erfurt, Germany, September 2004. 17, 169, 172

[VCS$^+$04] B. Verheecke, M. A. Cibrán, D. Suvée, W. Vanderperren, and V. Jonckers. Automatic service discovery and integration using semantic descriptions in the web services management layer. In *Proceedings of 3rd Nordic Conference on*

*Web Services*, volume 11, pages 79–89, Växjo, Sweden, November 2004. Växjo University Press, Series: Mathematical Modelling in Physics, Engineering and Cognitive Sciences. 169

[VCV$^+$04] B. Verheecke, M. A. Cibrán, W. Vanderperren, D. Suvée, and V. Jonckers. AOP for dynamic configuration and management of web services in client-applications. *International Journal on Web Services Research (JWSR)*, 1(3), 2004. 17, 169, 172

[vD97] A. van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Proceedings of Smalltalk and Java in Industry and Academia, STJA'97, pages 35–39, Erfurt, September 1997. Ilmenau Technical University.*, 1997. 87

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000. 86

[vdSH06] K. van der Sluijs and G-J. Houben. A generic component for exchanging user models between Web-based systems. *International Journal of Continuing Engineering Education and Life-Long Learning (IJCEELL'06)*, 16, 2006. 222

[Ver06] B. Verheecke. *Dynamic Integration, Composition, Selection and Management of Web Services in Service-Oriented Applications: An Approach using Aspect-Oriented Programming.* PhD thesis, Vrije Universiteit Brussel, Belgium, 2006. viii, 173

[vH01] B. von Halle. *Business Rules Applied.* Wiley, 2001. 2, 7, 22, 217, 227, 234

[Völ05] M. Völter. Patterns for handling cross-cutting concerns in model-driven software development. Version 2.3, 2005. Version 2.3, Dec 26, 2005. Available at http://www.voelter.de/data/pub/ModelsAndAspects.pdf. 64, 221

[VSCF05] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. In *workshop on Software Composition (SC 2005), co-located with ETAPS 2005, LNCS ISSN: 0302-9743*, Edinburgh, Scotland, April 2005. 43, 71, 73, 234

[VSV$^+$05] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in jasco. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 75–86, New York, NY, USA, 2005. ACM Press. 43

[VTH06] R. De Virgilio, R. Torlone, and G-J. Houben. A rule-based approach to content delivery adaptation in Web information systems. In *Proceedings of the 7th International Conference on Mobile Data Management (MDM'06)*, page 21, Washington, DC, USA, 2006. IEEE Computer Society. 223

[vWV03] J. van Wijngaarden and E. Visser. Program Transformation Mechanics. Technical Report UU-CS-2003-048, Universiteit Utrecht, 2003. 101, 102

[WPD92] S. Wartik and R. Prieto-Diaz. Criteria for comparing reuse-oriented domain analysis approaches. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):403–432, 1992. 84

[WTM+04] E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu. GlueQoS: Middleware to sweeten quality-of-service policy interactions. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 189–199, Washington, DC, USA, 2004. IEEE Computer Society. 223

[YAS] YASU Technologies Inc. QuickRules BRMS. http://www.yasutech.com/. 2, 14, 15, 91, 94, 215

[YAS03] YASU Technologies Inc. QuickRules: Application developer manual (version 2.5), 2003. ix, 216

[YJ02] J. Yoder and R. Johnson. The adaptive object-model architectural style. In *Working IEEE/IFIP Conference on Software Architecture (WICSA'02)*, 2002. 218