High-level specification of business rules and their crosscutting connections

María Agustina Cibrán
Vrije Universiteit Brussel
System and Software Engineering Lab
Pleinlaan 2, 1050 Brussels, Belgium
mcibran@vub.ac.be

Maja D'Hondt
Université des Sciences et Technologies de Lille
Laboratoire d'Informatique Fondamentale de Lille
59655 Villeneuve d'Ascq Cédex, France
Maja.D-Hondt@lifl.fr

ABSTRACT

Many object-oriented software applications are driven by rule-based knowledge, which typically appears implicitly represented, with negative effects on maintainability and reusability. Nowadays, business rules appear as a solution for separating rule-based knowledge from the core application functionality. Although many approaches exist which focus on expressing the separated business rules at the implementation level, not enough effort has been made towards expressing those implementation solutions at the domain level, which would facilitate their definition by domain experts. Moreover, as identified in previous work, the rule connections crosscut the core application and thus AOP is needed, which is also a solution purely at implementation level. In this paper we focus on abstracting business rules and rule connection aspects to the level of the domain. A high-level business rule language and connection language are proposed. The link to the implementation is invisible to domain experts and encapsulated in a mapping which is used to automatically translate the high-level specifications into the low-level AOP-based solutions.

Keywords

Object-Oriented Programming, Aspect-Oriented Programming, Business Rules

1. INTRODUCTION

Many object-oriented software applications are driven by rule-based knowledge about business policies, recommendations and decisions. Examples can be found in e-commerce applications, where rule-based knowledge guide customer preferences, personalized discounts, return and refund policies for instance. Typically, rule-based knowledge is represented implicitly and thus tangled in the application's code with negative effects on reusability and maintainability.

Nowadays, there is an increasing awareness for separating rule-based knowledge as explicit *business rules* [4] [11]. This separation is pursued throughout the whole software development cycle, from the discovery phase, to analysis, design and finally implementation. At the implementation phase, business rules are typically implemented in a rule-based programming language. Although this approach improves the more traditional object-oriented software development, it presents limitations: the rule-based languages are programming languages, implying a technological overhead and forcing the domain experts to have low-level programming skills to be able to write the rules.

Moreover, as we observed in previous work, current approaches fail to decouple the code that connects the business rules with the core application (i.e. applying the rules at certain events and gathering the necessary information for their application), which results crosscutting in the core application. Therefore, *Aspect-Oriented Programming* (AOP) is a good technique for encapsulating it [6] [8] [9] [10]. However, rule connection aspects represent again a solution purely at implementation level and thus not understandable by the domain experts.

The objective of this work is to express decoupled business rules and their (crosscutting) connections at a higher-level of abstraction, simplifying their specification by domain experts. In order to achieve this, we propose a *high-level domain model* consisting of domain concepts, business rules about these domain concepts, and connections of business rules to the core application in terms of the domain concepts. The link to the implementation is invisible to domain experts and encapsulated in a mapping. This mapping is used to automatically translate the high-level specifications into the (AOP-based) low-level implementation.

Two scenarios are possible for the creation of the domain model: 1) the domain model is built on top of an existing implementation, extracting and making explicit the domain information encoded in the application, 2) the domain entities are constructed first, reflecting an initial conceptual view of the system, which is then refined in order to obtain a concrete implementation model. Even though the two scenarios are possible with our approach, in this paper we focus on scenario 1.

This paper is organized as follows: section 2 presents the domain vocabulary that can be referred to in the high-level

business rules. Section 3 presents our high-level business rule language which allows expressing rules at the domain level. Section 4 introduces the high-level business rule connection language for expressing the connection of the rules also at the domain level. In every section, the mapping from the high-level specifications to an implementation based on OOP and/or AOP is explained. Section 5 presents related work and finally section 6 presents the conclusions and future work.

2. 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 connection with the core application. Three kinds of domain entities are defined based on the typical modelling elements: *domain concept*, *domain property* and *domain operation*. A *domain concept* is characterized by a set of domain properties and a set of domain operations and can have many instances. A *domain property* represents an attribute typically associated to the instances of a domain concept whereas a *domain operation* represents a behavior that can be invoked on instances of the domain concept. Examples of typical domain concepts found in the e-commerce domain are *Customer*, *Product*, *ShoppingCart*, *Account* and *Shop*. A *Customer* typically defines domain properties such as the *name*, *age* and *account* and is able to perform the operations to *add a product to his/her shopping cart* and *check out*.

A domain concept can be implemented in many ways, depending on many factors, such as the concrete application requirements, the technology adopted and even the particular programmer that is in charge of coding the application. In our approach, the link from high-level domain entities to implementation is encapsulated in a *mapping*. A full categorization of the mappings from domain entities to implementation can be found in [7]. In this section we briefly introduce these mapping categories and illustrate them in the ecommerce domain.

If we consider for instance the property *age* characteristic of the concept *customer*, the following two realizations are possible in an OO application: a) the domain property is implemented as an attribute in a class; b) a domain property is implemented as a method in a class. In either case, we say that there is a one-to-one mapping between the domain entity and the implementation entity, since there is exactly one implementation entity that realizes the domain entity. We refer to this kind of mapping as *anticipated*.

We observe that, when building the domain model on top of an existing application, the domain entities can be discrepant from the existing implementation entities as there might be no one-to-one correspondence between them. In this case we say that the mapping is *unanticipated*. For instance, a domain property instead of being mapped to a single attribute or method in a class, it can be implemented as the sum of two attributes existing in different classes (i.e. an OO expression involving other existing implementation entities). A concrete example in the ecommerce domain is illustrated in figure 1. Here, the domain property *discount* defined in *Customer* is mapped to an OO expression that calculates the 10 percent of the amount spent by the customer. As the elements involved in the calculation of the discount (i.e. account and amount) are also expressed at the domain level, the expression that represents the mapping can be defined completely at the domain level, without referring to implementation details (see grey box in figure 1). The low level OO expression involving the concrete implementation entities can be automatically generated by means of traversing the mappings of the domain entities involved in the high-level expression.

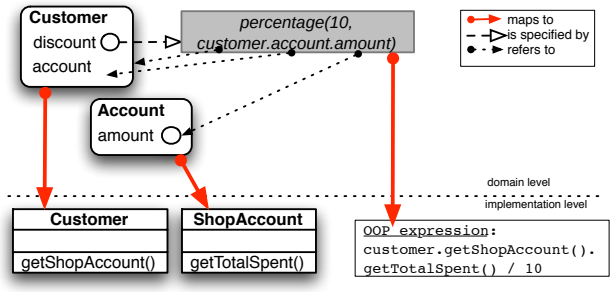

Figure 1: Linking an unanticipated domain entity to implementation using OOP

Another possibility is that a domain entity appears crosscutting in the existing application. For instance, a domain property can appear implemented as a calculation based on information that is scattered or tangled in different places of the core application. Thus, AOP is needed to realize the mapping of that domain entity at the implementation level. Still, we want AOP to be used in a transparent way for the domain expert. Thus, the mapping is still expressed completely at the domain level, in terms of special domain operators and other existing domain entities. We propose patterns for translating these domain operators to aspects. Figure 2 shows an example in the ecommerce domain. The mapping from the *averageTimeBetweenOrderAndPayment* domain property to implementation is specified (at a high-level) as an expression describing the calculation of the time between the invocation of the domain operations *orderProduct* and *checkout*. The *timeBetween* domain operator is translated into an aspect that hooks on the invocation of the concrete methods *addProductToBasket()* and *checkoutBasket()* (mapped to the domain operations *orderProduct* and *checkout* respectively) and calculates the elapsed time between the two invocations. This translation occurs transparently for the domain expert, only requiring the high-level mapping description as input.

A third possibility is that a domain entity is unanticipated in the existing implementation. In this case, instead of linking the domain model with an existing implementation, the mappings define how the domain entity needs to be implemented. More details about the mappings from domain entities to implementation can be found in [7]. The mappings specified completely at the domain level, can be fully automated. Others, the one-to-one mappings, cannot be completely automated since knowledge about the concrete implementation is required.

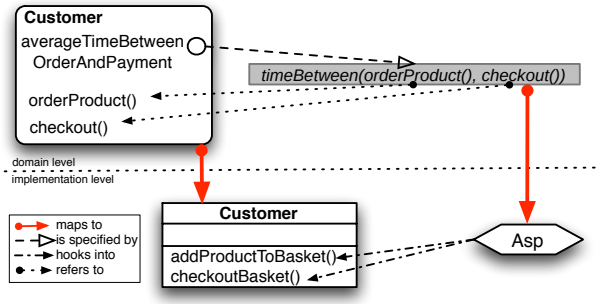

Figure 2: Linking an unanticipated domain entity to implementation using AOP

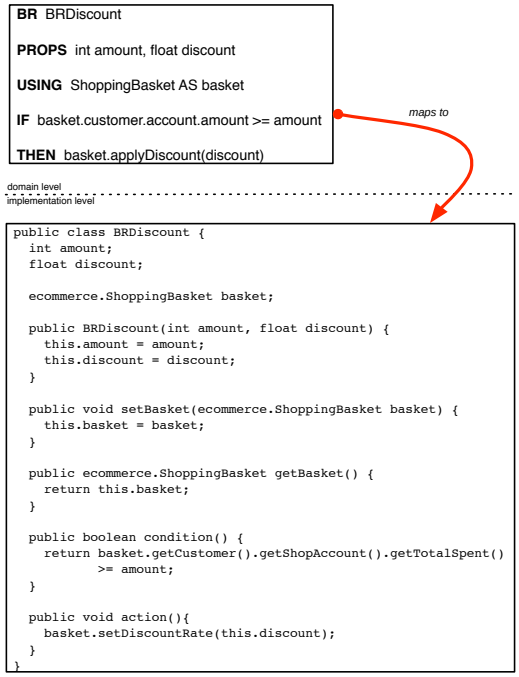

Figure 3: High-level BRDiscount rule and its translation to a rule class

3. RULES AT THE DOMAIN LEVEL

Our approach proposes abstracting the specification of the business rules at the level of the domain, in terms of concepts inherent to the domain and independent of a concrete implementation. The idea of a high-level rule language is not new and therefore present in some existing approaches [14] [17] [20] [12]. Although these approaches allow specifying rules at a higher-level, they require domain entities to have one-to-one mappings with existing implementation entities (see section 5).

We propose a high-level business rule language that allows the definition of **high-level rules** which: i) are declarative; ii) defined in terms of the domain (domain entities taken from the domain model); iii) are independent of any implementation detail. An example rule is shown in Figure 3. *BRDiscount* applies a discount on a shopping basket if the customer that is the owner of that basket has already spent more than a certain amount of money. Several parts are required in the rule definition: the IF clause defines the condition of the rule and must evaluate to true in order to trigger the action; the THEN clause defines the action to be triggered when the condition is met. Optionally, rule templates can be defined using the PROPS clause which specifies information provided at rule creation time. The USING clause is used to specify information passed to the rule at rule connection time.

Following the rule-object pattern [2], the high-level rule maps to a class that defines methods for their condition and action. Figure 3 illustrates the mapping of a high-level rule to a java class. This translation is done automatically, using the mappings (anticipated or unanticipated) of the domain entities involved in the definition of the high-level rule.

4. RULE CONNECTIONS AT THE DOMAIN LEVEL

At the implementation level, the business rule connection identifies the code that connects the rules with the core application. Typically this code is in charge of pointing out the moment in the execution of the core application at which the rule needs to be triggered and making available the information needed for the application of the rule. In previous work [5] [6] [8] [9] we observe that current approaches fail at separating this connection code, which appears crosscutting in the core application. We then show how AOP can successfully be used to encapsulate the rule connections by presenting rule connection patterns (in particular using AspectJ [3] and JAsCo [19]). Even though the AOP-based patterns achieve the successful decoupling of the rule connection code, they represent a solution purely at the implementation level. Our objective is then to abstract these AOP-based patterns and express them at the domain level, facilitating their definition by domain experts.

At the domain level, a **rule connection** specifies the moment at which a rule should be applied which can be before, after or in replacement of an *event*. An *event* refers to the invocation of a domain operation and reifies its context. The information available at the context of the event is available to be passed to the rule, enabling its application. Figure 4 shows an example of a rule connection specification that connects the rule *BRDiscount* before the occurrence of the *Checkout* event (which encapsulates the invocation of the domain operation *checkout*). The MAPPING clause links the information available in the context of the *Checkout* event with the information required by the *BRDiscount* rule. The information passed to the rule can be modified in two ways (possibly having impacts on the core application execution): 1) by invoking domain methods that modify the state of the core application (like in our example by invoking the method *setDiscountRate* on the *basket* business object), 2) by assigning (using the = operator) new values to the passed business objects. The PROPS clause defines the information needed at rule creation time (in this example, the amount spent threshold).

The rule connection also includes the specification of the rule activation time, i.e. the period of time in which the rule application should be considered. The activation is also specified at a high-level in terms of events. The example in figure 4 specifies that the *BRDiscount* rule should only be considered in the context of an express checkout (imagine that an express checkout is a special kind of checkout which uses payment information already stored in the shop and that does not require validation by the client). *CheckoutExpress* is a high-level event denoting the invocation of a domain operation mapped to the *checkoutExpress(Customer)* method defined in the class *Shop*. Other examples of activation time are: *between the customer logs in and the moment he/she adds a product to the shopping cart* and *not while the customer is browsing the products*. The specification of the activation time is optional and when not specified it is assumed that the rule is always active.

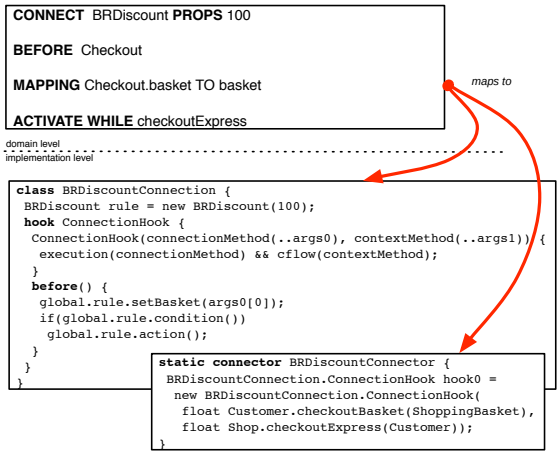

Figure 4: High-level connection of BRDiscount rule at Checkout event and its translation to JAsCo aspects

As we adopt JAsCo [19] as the AOP language to implement the aspect patterns [8], for each rule connection specification, an aspect bean and a connector are generated as follows: if the rule is connected *before* a certain event, then two cases are possible: *a)* if information available in the context of the connection event is passed to the rule and modified by it (assigned by the rule), then an *around advice* is created which first invokes the rule action and then invokes the original event behavior taking into account the modifications performed by the rule; *b)* otherwise, if no contextual information is modified by the rule, a *before advice* is created which triggers the rule action. If the rule is connected *after* an event, then two cases are possible as well: *a)* if the result of invoking the event is passed to the rule, then an *around advice* is created which first invokes the original behavior and passes the result to the rule. In the case the rule modifies that result, then the around advice returns that modified result; otherwise the original result is returned. *b)* if no result is passed to the rule, then an *after advice* is created which triggers the rule action. If a rule is connected replacing a certain event, then an *around* advice is created which only invokes the rule action.

On the other hand, the activation time is translated as follows: *a)* if it is defined as WHILE ⟨event⟩(or NOT WHILE ⟨event⟩), then a *cflow* joinpoint (or a !cflow joinpoint respectively) is added to the connection hook (see Hook0 in figure 4); *b)* if it is defined as BETWEEN ⟨event1⟩ AND ⟨event2⟩, then an extra hook is added to the aspect bean which defines a before and an after advice which set a flag (global variable in the aspect bean) and unset it respectively, indicating whether the rule is active or not. A check on this flag needs to be added in the *isApplicable()* method of the connection hook.

As JAsCo separates the aspect logic from the deployment information, the concrete core application details corresponding to the mappings of the events (e.g. the signatures of the methods where to hook) are specified in the connectors. Following these guidelines, the mapping of the high-level connection specification to the aspect patterns is done completely automatically.

5. RELATED WORK

Several state-of-the-art systems that support business rules were studied and analyzed, among them JRules [14], OPSJ [1], NéOpus [16], HaleyRules [12], QuickRules [17], RuleML [18], IRules [13], OptimalJ [15] and VisualRules [20]. Some approaches, such as JRules, OPSJ, NéOpus, HaleyRules 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 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. The difference with our language is that their business model exposes the OO classes and methods to which the business rules are applied, mapping the natural language 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. On the contrary, our approach enables the definition of domain entities that can have complex mappings to the implementation level. Moreover, we propose decoupling the rule connections and making them explicit at the domain level, which current approaches fail to do. The use of AOP to realize rule connections at the implementation level is also a contribution of our approach.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented an approach for expressing decoupled business rules and their crosscutting connections at the level of the domain. The domain entities that constitute the ingredients of the high-level rules, a high-level business rule language and connection language were presented.

Their translation to implementation, which is achieved completely automatically, was described and exemplified. The use of AOP in our approach has the following benefits:

1) the dynamic addition and removal of business rules becomes possible: new high-level rules can be defined at run-time for which code is generated without having to (i) modify the core implementation nor (ii) regenerate the code for the already existing rules. The code implementing a new rule is localized in a separate module. Analogously, rules can be removed at run-time, without having to change existing code. 2) rules traceability becomes possible: every high-level entity has a corresponding localized implementation. As the mapping is made explicit between the high-level descriptions (for both, rules and connections) and their corresponding implementation, traceability becomes possible. 3) raising level of abstraction of aspect constructs: high-level entities are defined for aspect constructs. However, the use of AOP is transparent at the domain level, and thus domain experts do not need to be aware of the use of AOP as part of the translation.

The main future work is in relation with the definition of the mappings. Currently, a metamodel that describes all the types of mappings is proposed. Thus, in order to define concrete mappings, knowledge about this metamodel is needed. Whereas some mappings can be defined completely at the domain level, in terms of domain entities, the low-level mappings require knowledge about the concrete implementation, and thus their definition becomes more complex. Also, a metamodel is provided describing domain concepts, domain properties and domain operations, which needs to be instantiated in order to create concrete domain entities. We observe that all these metamodels add extra complexity and make the adoption of the approach less straightforward. In future work we will investigate how the use of UML models can simplify the creation of domain entities and mappings: two UML models would be defined at different levels of abstraction, one more conceptual representing the view of the domain expert, and a more detailed one reflecting the design of the concrete implementation. The mapping would then be defined at the level of the models, linking the more conceptual UML model to the more detailed one. OCL could be used for defining the more structural mappings whereas sequence diagrams could be employed to describe the more behavioral ones.

7. REFERENCES

[1] *OPSJ 4.1*, 2001. Manual by Charles L. Forgy from Production Systems Technologies Inc.

[2] A. A. Rule object 2001: A Pattern Language for Adaptive and Scalable Business Rule Construction. 2001.

[3] AspectJ. http://eclipse.org/aspectj/.

[4] V. H. B. *Business Rules Applied*. Wiley, 2001.

[5] 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.

[6] M. A. Cibrán, M. D'Hondt, and V. Jonckers. Aspect-Oriented Programming for Connecting Business Rules. In *Proceedings of BIS International Conference*, Colorado Springs, USA, June 2003.

[7] M. A. Cibrán, M. D'Hondt, and V. Jonckers. Mapping high-level business rules to and through aspects. In *2me Journe Francophone sur le Dveloppement de Logiciels Par Aspects (JFDLPA 2005)*, Lille, France, 2005.

[8] 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 CSITeA International Conference*, Rio de Janeiro, Brazil, June 2003.

[9] 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 ASSE'04, Argentine Conference on Computer Science and Operational Research*, Córdoba, Argentina, Sept. 2004.

[10] M. D'Hondt and V. Jonckers. Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge. Lancaster, UK, Mar. 2004.

[11] R. R. G. *Principles of the Business Rule Approach*. Addison-Wesley Publishing Company, 2003.

[12] HaleyRules. http://www.haley.com/products/HaleyRules.html.

[13] IRules. http://www.eas.asu.edu/ irules/.

[14] JRules. http://www.ilog.com/products/jrules/.

[15] OptimalJ Business Rules. http://www.compuware.com/products/optimalj/.

[16] F. Pachet. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, 8(4):19–24, 1995.

[17] QuickRules. http://www.yasutech.com/.

[18] RuleML. http://www.ruleml.org/.

[19] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. Boston, USA, Mar. 2003.

[20] Visual Rules. http://www.visual-rules.de.