Composable and Reusable Business Rules Using AspectJ[*]

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

Maja D'Hondt
System and Software Engineering Lab
Vrije Universiteit Brussel, Belgium
mjdhondt@vub.ac.be

1 Introduction

The complexity of business domains is steadily increasing and it is becoming more important to explicitly capture business processes and policies as business rules. The Business Rules Group defines a business rule as *a statement that defines or constraints some aspect of the business. It is intended to assert business structure or to control the behaviour of the business* [3]. Business rules tend to evolve more frequently than the core application functionality, and therefore should be decoupled from the rest of the application [11][7][1]. However, in order to achieve highly flexible, configurable and reusable business rules, it is also necessary to fully separate the code that connects the business rules with the core application and encapsulate it. We observe that this code crosscuts the core application and that Aspect-Oriented Programming [2][5] is necessary for encapsulating it. In particular this code

- connects business rules to core application events which depend on run-time properties and
- provides necessary business objects to make business rules applicable at those events.

But in addition to the need for encapsulating crosscutting aspects, we also pursue good software engineering practices. In this paper, we focus on achieving *composability* and *reusability*. As will be discussed later, business rule connectors typically consist of several parts. Therefore we want to *modularize* each of these parts and achieve:

- composability of all parts involved in business rule connectors and
- reusability of business rule connectors and their parts.

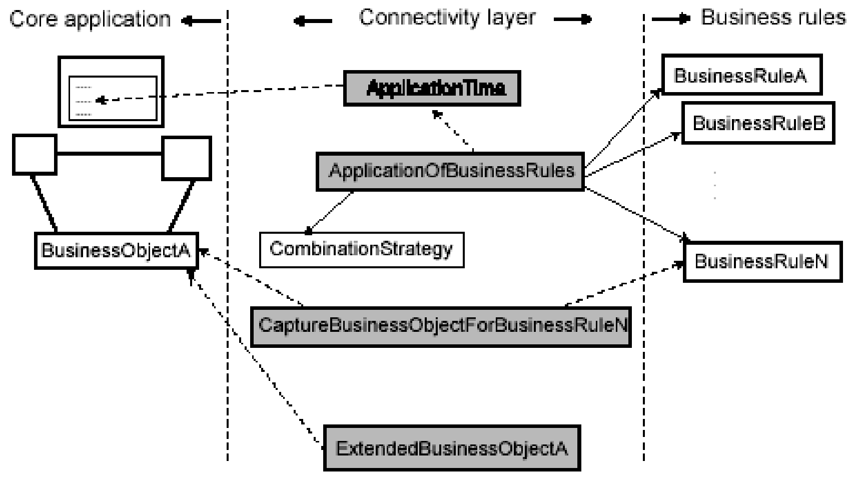
With its dynamic join point model and its join point parameter mechanism, AspectJ [8] is a suitable candidate for achieving the aforementioned encapsulation of the business rule connectors. Moreover, AspectJ offers relatively mature tool support, which allows us to investigate closely its effect on software engineering. Aspects are also a suitable vehicle for modularising business rule connectors in their identified parts. However, composing and reusing those parts is not fully supported. This paper shows AspectJ's limitations with respect to this, illustrating them with business rules for price personalisation in an online store.

Section 2 describes design decisions we have taken to achieve the decoupling of business rule connectors. Section 3 introduces the example application. Section 4 and 5 show examples of situations where AspectJ features fall short to achieve composability and reusability. We discuss related work in Section 6 and conclude in Section 7.

2 Design Decisions

As a first attempt it seems natural to model the core application with objects and the business logic with aspects. However, we found that it is only the connectivity between the core application and the business logic that requires the weaving power of AspectJ whereas the conditions and most actions of the business rules can be expressed using objects (typically classes with methods for the business rules conditions and actions) as it is shown in [9][1].

Moreover it is desirable to keep the business logic as reusable as possible. Because of that and due to the complexity of business logic we decided to use the power of the object-oriented paradigm to model business logic. Thus, in our approach both the core application and the business rules are expressed as objects and we use aspects for encapsulating the code that connects the business rules to the core application. However we want parts of the business rules connectors to be reusable in other situations. Thus, in order to enhance separation of concerns on the connectivity layer, several aspects are created to define the connectors (shown below).



Typically, there will be an aspect expressing the event that determines the application time of the business rule, an optional aspect exposing unavailable or introducing unanticipated business objects to the event, and a last aspect that puts the

[*] Funded by the *Flemish Institute for the Improvement of the Scientific-Technological Research in Flanders* (Belgium).

previous ones together and actually triggers the application of the business rule. Each of these aspects can have different manifestations. The second aspect may expose unavailable business objects that were anticipated (i.e. it is implemented in the core application but not available at the event) or unanticipated (i.e. not foreseen and implemented in the core application). The implementation of the last aspect will differ depending on the second aspect. All these aspects combined together form the connectivity layer in between the core application and the business rules. This way, splitting the connectivity layer in several aspects allows each part to be independently reused, achieving high flexibility and configurability of business rules connectors.

Notice that it is not the objective of this paper to exhaustively discuss the different design solutions that can be proposed. A more complete analysis of the suggested design solution as well as the requirements that motivated it can be found in [4].

3 E-Store Example

We introduce an example application that models an electronic store. The customer can chose the products he or she wants to buy and add them to the shopping cart, indicating the desired number of units. When the customer wants to confirm the purchase an order is generated and the check-out process is started, including information about paying mechanism, shipping address, delivery options and wrapping details. Each customer of the e-store has an account containing the buying history of that customer.

In this simple model, a price personalization business rule is implemented in Java™ by a single class implementing the methods `condition()` and `action(Float price)`, shown below. The abstract class `BRPrice-Discount` implements the method `apply()` that tests the condition, and performs the action if the condition evaluates to true or returns the standard price if it does not. It also provides a default implementation for the method `action()` (not shown here) by subtracting the percentage attribute from the standard price.

```
abstract public class BRPriceDiscount{
 private Float percentage;
 abstract public boolean condition();
 public Float action(Float price){...}
 public Float apply(Float price){
  if (condition()){
   return action(price);
  else{return price}; } } }
```

Suppose we consider two concrete price personalization business rules: BR1 *if today is Christmas then a customer gets a 5% discount* and BR2 *if the customer has bought more than 10 products then he or she gets a 10% discount on the current purchase.* The implementation in Java™ of these rules, `BRChristmasDiscount` and `BRPurchasedItems-Discount` respectively is shown below. They define concrete implementations for the inherited method `condition()`.

```
public class BRChristmasDiscount
    extends BRPriceDiscount{
 private boolean isChristmas(){...}
 public boolean condition(){
  return isChristmas(); } }

public class BRPurchasedItemsDiscount
    extends BRPriceDiscount{
Customer customer;
public void setCustomer(Customer c){
    customer = c; }
public boolean condition(){
  // returns if customer purchased more than 10 products } }
```

As part of the connectivity layer we need to define the following aspects: An aspect `EPricePer-sonalisation` identifies the event in the core application that defines the application time of both business rules introduced before.

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

`ApplyChristmasDiscount` and `ApplyBRPurchasedItemsDiscount` apply the rules at the given application time. Notice that we make use of AspectJ's feature `dominates` to control the order in which the discounts are applied: whenever both rules are applicable, the discount specified in `BRPurchasedItemsDiscount` is applied before the discount in `BRChristmasDiscount`.

```
aspect ApplyChristmasDiscount dominates ApplyBRPurchasedItemsDiscount {
 BRChristmasDiscount businessRule;
 public void setBR(BRChristmasDiscount br){
  businessRule = br; }
 Float around(Product p):
  EPricePersonalisation.priceCalculation(p){
  Float price = proceed(p);
  return businessRule.apply(price); } }

aspect ApplyBRPurchasedItemsDiscount
                       percflow(CaptureCustomer.checkOut(Customer)){
 BRPurchasedItemsDiscount businessRule;
 public void setBR(BRPurchasedItemsDiscount br){
  businessRule = br; }
 Float around(Product p):
  EPricePersonalisation.priceCalculation(p){
  Float price = proceed(p);
  return businessRule.apply(price); } }
```

As we can observe, `BRChristmasDiscount` only needs globally available information to evaluate its condition. On the contrary, `BRPurchasedItemsDiscount` needs to have the information about the customer that is currently buying to be able to check its condition. To be able to capture the customer for the business rule to be applied we need to define the following aspect `CaptureCustomer`:

```
aspect CaptureCustomer{
 pointcut checkOut(Customer c):
  call(public Float CheckOut.checkOut(Customer)) && args(c);

 before(Customer c): checkOut(c){
  BRPurchasedItemsDiscount chBR = new BRPurchasedItemsDiscount();
  chBR.setCustomer(c);
  ApplyBRPurchasedItemsDiscount.aspectOf().setBusinessRule(chBR); } }
```

Also note that the application aspects of the business rules are no longer atomic and independent of other business rules because the `dominates` statement needs to refer to another business rule application aspect.

4 Aspect Composability and Coordination

As we mentioned before, we pursue maximum configurability by separating each of the parts that form the connectivity layer in different aspects. However, to achieve the desired behaviour we need to adequately compose the several aspects for them to interact in the correct way. Thus, significant coordination among the different pieces is needed.

4.1 Example 1

To illustrate this problem, the example introduced in the previous section is considered. Both aspects for the application of the rules, `ApplyChristmasDiscount` and `ApplyBRPurchasedItemsDiscount`, need to refer to the aspect `EPricePersonalisation` that defines the application time to trigger the application of the rules at the occurrence of that event, determining the need for a reference relation between these aspects. Moreover, the type of information the rules need, determine the way the application aspects are instantiated. For example, we need a different instance of `ApplyBRPurchasedItemsDiscount` each time a customer checks out. Thus, the instantiation of the `ApplyBRPurchasedItemsDiscount` aspect will be determined by the pointcut `checkout` defined in the aspect `CaptureCustomer`, introducing a dependency between the two. The problem is that in AspectJ all the relations between aspects are implicitly specified in the same code of the aspects. 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, generally by hard-coding the references and dependencies in the aspect code itself. This not only makes the application to be error prone but affects reusability of aspects (section 5).

Thus, when the objective is to split the functionality in different aspects pursuing reusability, AspectJ provides limited support for composing all the pieces together for the system to behave in the desired way.

4.2 Example 2

We introduce a small variation on the previous example by slightly modifying the `BRPurchasedItems-Discount` business rule. The rule defines in its condition that the required number of items bought by the customer for the rule to be applied must be greater than 10. Now suppose that due to changes in business considerations we would like to define that number depending on situations that are checked at run time. For instance, if the customer is frequent, the rule needs to check that the customer has bought 5 items and if the customer is not frequent, we want to apply the discount only if he or she has bought at least 10 products.

To implement this change, the business rule `BRPurchasedItemsDiscount` must introduce this variation:

```
public class BRPurchasedItemsDiscount extends BRPriceDiscount{
 static int minAmount;
 ... //same implementation as before
 public boolean condition(){
     // returns if customer purchased more than minAmount items} }
```

Now the application aspect becomes abstract, and we define two concrete subaspects to distinguish between the application of the rule for a frequent and non frequent customer:

```
abstract aspect ApplyBRPurchasedItemsDiscount {}

aspect ApplyBRFrequentPurchasedItemsDiscount
```

```
    extends ApplyBRPurchasedItemsDiscount
    percflow(CustomerFrequency.frequentCustomer(Customer)){

     BRPurchasedItemsDiscount.setMinAmount(5); }

aspect ApplyBRNoFrequentPurchasedItemsDiscount
    extends ApplyBRPurchasedItemsDiscount
    percflow(CustomerFrequency.noFrequentCustomer(Customer)){

     BRPurchasedItemsDiscount.setMinAmount(10); }

aspect CustomerFrequency dominates ApplyBR*{

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

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

The problem in this solution appears in the `CaptureCustomer` aspect (section 3). In the before advice defined in that aspect, `ApplyBRPurchasedItemsDiscount.aspectOf()` needs to pick the right instance of the application aspect to be associated with the right instance of the rule created before. But now `ApplyBRPurchasedItemsDiscount` is an abstract aspect and there is no way to refer to the aspect instance of the concrete subaspect that is effectively instantiated. That information is only known at run time, when the current customer that is checking out is evaluated to see if he or she is frequent.

This problem occurs because to access an aspect instance in AspectJ, it is necessary to refer to the aspect class that is instantiated. But in this example situation, that information is not known until run-time. This example illustrates that it is not always possible in AspectJ to refer to particular aspect instances, especially when they are created based on dynamic properties.

This uncovers a fundamental feature, *referencing aspect instances*, which should be well-designed in any aspect-oriented programming language because it affects composability.

5 Reusability

We want to achieve maximum reusability not only by dividing the connectivity layer in many aspects in order to reuse each part independently when rules need to be applied in different contexts, but also stressing the reuse of their implementation through inheritance of aspects, as a well-known good software engineering practice.

In the previous example, both aspects `ApplyChristmasDiscount` and `ApplyBRPurchasedItemsDiscount` for the application of the rules implement the same around advice for the triggering of the rules. Thus, we would like to abstract out that definition in an abstract aspect `ApplyBRPriceDiscount` and reusing the generic implementation as much as possible. Both concrete aspects `ApplyChristmasDiscount` and `ApplyBRPurchasedItemsDiscount` will inherit from the new abstract aspect.

```
abstract aspect ApplicationOfBRPriceDiscount{
 BRPriceDiscount businessRule;
 public void setBusinessRule(BRPriceDiscount br){
  businessRule = br; }

 Float around(Product p):
   EPricePersonalisation.priceCalculation(p){
    Float price = proceed(p);
    return businessRule.apply(price); } }

aspect ApplyChristmasDiscount
        extends ApplicationOfBRPriceDiscount
        dominates ApplyBRPurchasedItemsDiscount {}

aspect ApplyBRPurchasedItemsDiscount
        extends ApplicationOfBRPriceDiscount
        percflow(CaptureCustomer.checkOut(Customer)){}
```

As in the previous example in section 3, we would still like to control the order in which the discounts are applied by using the `dominates` feature between `ApplyChristmasDiscount` and `ApplyBRPurchasedItemsDiscount`. But the difference now is that the around advice is not defined in each of the application aspects but inherited from the abstract aspect. In this case the `dominates` feature would not perform in the desired way because it does not work on inherited advices. This restriction of AspectJ affects reusability of aspects as we cannot abstract out common pieces of advice defined in concrete aspects and still explicitly control their order of execution.

6 Related Work

Several approaches exist that deal with business rules and can be used in the context of object-oriented software engineering. The most notable approaches are the rule-based system implemented as a Java™ library SweetRules (formerly known as CommonRules) from IBM [6]; Business Rule Beans using Java™ Enterprise Beans also from IBM [10]; von Halle's Business Rules Approach [11]; and object-oriented patterns for business rules such as the Rule Object Pattern [1] (related to Adaptive Object Models [12]), Patterns for Personalisation [9] and Rule Patterns [7]. We do not discuss all the distinguishing characteristics of each approach, but observe their common lack of support for separating and encapsulating business rule connectors.

A more complete coverage of all the complex issues involved in business rules, such as for example business rule interference is addressed in our work described in [4]. In that work we identify a list of general requirements that any support for business rules in an object-oriented software engineering context should consider. Although it is impossible to generalize the results to all aspect-oriented approaches because some of them are too fundamentally different, it is possible to distil at least patterns of AspectJ usage. These patterns are not included here, but extensively described in [4].

We do not discuss related work concerning AOP here, since the purpose of this paper is not to compare different aspect-oriented approaches but to report on concrete experiences using AspectJ as a representative AOP approach.

7 Conclusions

We conclude that AspectJ successfully encapsulates the business rules connectors and supports modularization of each part of the connector using aspects. However, it fails to achieve full composability and reusability of the resulting aspects.

Moreover, this results in a proliferation of aspects which are hard to manage. The problem is that there is no higher-level abstraction available to express the composition of aspects. As a consequence, the composition details are specified in the aspects themselves and not as first-class values.

Finally, we observe that AspectJ, and probably each aspect-oriented approach, has some very powerful and low-level features that are used for solving a wide range of problems. In AspectJ, percflow is an example. Since the same features are used for solving semantically different concerns, they impede program understandability and portability.

8 References

[1] A. Arsanjani. Rule object 2001: A pattern language for adaptive and scalable business rule construction.
[2] Aspect-Oriented Software Development. http://www.aosd.net/
[3] The Business Rules Group. Defining Business Rules: What Are They Really?, http://www.businessrulesgroup.org/, July 2000.
[4] M. A. Cibrán. Using Aspect-Oriented Programming for Connecting and Configuring Decoupled Business Rules in Object-Oriented Applications. Master Thesis, Vrije Universiteit Brussel, Belgium, 2002.
[5] Communications of the ACM. Aspect-Oriented Software Development, October 2001.
[6] B. N. Grosof, Y. Kabbaj, T. C. Poon, M. Ghande, and et al. Semantic Web Enabling Technology (SWEET). http://ebusiness.mit.edu/bgrosof/
[7] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen. From rules to rule patterns. In Conference on Advanced Information Systems Engineering, pages 99--115, 1996.
[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. An overview of AspectJ. In Proceedings European Conference on Object-Oriented Programming, volume 2072 of Lecture Notes in Computer Science, pages 327--353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.
[9] G. Rossi, A. Fortier, J. Cappi, and D. Schwabe. Seamless personalization of e-commerce applications. In 2nd International Workshop on Conceptual Modeling Approaches for e-Business at the 20th International Conference on Conceptual Modeling, Yokohama, Japan, 2001.
[10] I. Rouvellou, L. Degenaro, K. Rasmus, D. Ehnebuske, and B. McKee. Extending business objects with business rules. In 33rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe 2000), Mont Saint-Michel - St-Malo, France, pages 238--249, 2000.
[11] B. von Halle. Business Rules Applied. Wiley, 2001.
[12] J. Yoder and R. Johnson. The adaptive object-model architectural style. In Working IEEE/IFIP Conference on Software Architecture (WICSA), Montreal, Canada, 2002.