Aspect-Oriented Programming for Connecting Business Rules

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

Viviane Jonckers
System and Software Engineering Lab
Vrije Universiteit Brussel
Belgium
vejoncke@info.vub.ac.be

Abstract

In object-oriented business support applications, separating business rules from core application functionality is crucial. Essential to business rules are their connectors *to the core application, which existing approaches fail to support explicitly. Our paper presents requirements for encapsulating business rule connectors and shows that* Aspect-Oriented Programming *is necessary to fulfil them. All is illustrated with a small e-commerce example in* Java™ *and* AspectJ.

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 Approach* [23] states that it is crucial to implement them adhering to four objectives: *separate* business rules from the core application, *trace* business rules to business policies and decisions, *externalise* business rules for a business audience, and *position* business rules for change.

In this paper, we consider business rules in the context of business support applications developed using object-oriented software engineering techniques. Although we agree with [23] that there are discrepancies between an object-oriented approach and a business rules approach, objects are the state of the art in software engineering and hence our starting point. These kinds of applications have a considerable application layer and are usually implemented in an object-oriented programming language with extensive libraries, such as Java™. They also depend on domain and business knowledge manifest as business rules, which would be lost in the implementation of the core application functionality without a business rules approach.

Pure database approaches for supporting business rules and rule-based systems for developing entirely knowledge-intensive applications are not suitable when dealing with inherently object-oriented applications. Fortunately, there are some approaches that advocate and support the four objectives of a business rules approach in object-oriented software applications. However, they all fail to apply these objectives to the part that *connects* the business rules to the core application: one has to adapt the source code of the core application manually in different places each time the *connector* changes due to evolving business rules. Moreover, the connector should be reusable in other business rule configurations. Current business rules approaches have no support for separating and encapsulating code with these characteristics, thus cannot fully separate, trace, or externalise business rules, nor position them for change.

This phenomenon is known as *crosscutting* code in the area of *Aspect-Oriented Programming* (AOP). Crosscutting code cannot be encapsulated in the decomposition mechanisms of a programming language, such as functions, objects or any other constructs. Approaches that support AOP provide additional constructs to encapsulate this code and make it reusable. *AspectJ* is an aspect-oriented extension of Java™ proposing *aspects* to achieve this. Although AOP is usually used for encapsulating logging, synchronisation, and other implementation-level issues, [11] and [10] show that the novel idea of domain knowledge as an aspect is also accepted.

The focus of this paper is the problem of connecting the business rules to the core application functionality in object-oriented software applications. We present requirements for encapsulating business rule connectors (Sec. 5), after discussing business rules (Sec. 2), the problem with connecting them to the core application (Sec. 3), and an example e-commerce application with price personalisation (Sec. 4), which will be used throughout the paper. AOP, and in particular *AspectJ*, is employed to complement the object-oriented implementation of the e-commerce example with aspects to encapsulate the business rule connectors (Sec. 6). Finally, we discuss related work (Sec. 7) and conclude

Witold Abramowicz, Gary Klein (eds.), Business Information Systems, Proceedings of BIS 2003, Colorado Springs, USA

(Sec. 8). Note that this paper reports on (part of) the work done in [6].

2. 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* [4]. A significant characteristic of business rules is that they tend to change whenever the business policies they embody change, which is more often than the core application functionality does [23][15][1].

Business rules are present in business support systems, software applications that are driven by business or domain knowledge. Examples of such businesses or domains are the financial industry, television and radio broadcasting, hospital management, rental businesses and electronic commerce. A business rule is applied at an event which is a well-defined point in the execution of the core application functionality. There are different kinds of business rules [23]:

- **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. For example, an inference is *if a customer has purchased more than 20 products then he or she is a frequent customer*.

3. The missing link

Changes to software applications can occur in several ways. First, real-world domains change and businesses have to cope with this in order to stay competitive. On top of this, business policies tend to be very time-sensitive which means that they might be valid in a certain period of time, sometimes even changing from day to day. Finally, there are variations within a family of applications in a certain domain where different stakeholders require different business policies. Business rules are reconfigured and reused in a number of ways in order to accommodate these changes: they are plugged in and out of the application; conditions, actions, and events are reused in other business rules; and business rules are reused across applications.

In order to optimise reuse of business rules, they should be fully separated from the core application functionality. Separated business rules facilitate traceability and reduce propagation of changes to the core application, which in turn simplifies impact assessment. Finally, separated business rules allow externalisation for business audiences. [23]

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 [13]; *Business Rule Beans* using Java™ Enterprise Beans also from IBM [20]; von Halle's *Business Rules Approach* [23]; and object-oriented patterns for business rules such as the *Rule Object Pattern* [1] (related to *Adaptive Object Models* [25]), *Patterns for Personalisation* [18] and Rule Patterns [15].

We do not discuss all of these different approaches here, but observe that they have one thing in common: they all focus on separating the business rules from the core application, but do not offer support for encapsulating the connection of the business rules to the core application. In order to have maximum configurability of business rules, it is also necessary to separate the connection from the core application and encapsulate it. Indeed, whenever a business rule is plugged in or out of the core application, the corresponding connection has to be updated. If this connection is separated and encapsulated, then this can be achieved without having to touch the core application, which should not be affected by changes in the ever-evolving business rules. Moreover, the connection can be reused in different configurations.

Essentially, this connection denotes the events that correspond to precise points in the execution of the core application, and captures business objects needed for the application of the business rule in the context of those events. Using the existing business rules approaches, however, one has to adapt the source code of the core application manually in different places. Depending on the approach this is done differently, but essentially the results are the same: the code that enables the connection of a business rule is scattered in the core application, or - to use terminology from AOP - the connection *crosscuts* the core application. In current business rule approaches, it is impossible to separate and encapsulate the connection of business rules to the core application because they have no support for encapsulating crosscutting code, as opposed to aspect-oriented approaches.

This problem is thoroughly discussed in Sec. 5, as a result of which we distil general requirements for encapsulating the business rule connectors. First, we introduce the running example of this paper in the next section.

4. An e-commerce example

In this section, we illustrate the domain of e-commerce, which is used by most of the existing business rule approaches as a case. We introduce three business rules and events to be used in the remainder of this paper to identify problems and explain solutions. Next, we present the design and implementation of a very simple online store, constructed using object-oriented patterns for business rules since the implementation language is Java™. This approach to business rules is essentially different from other approaches in the way the business rules are separated and represented, but the problems arising with connecting them to the core application are essentially the same. The use of this approach has the added benefit of being "light-weight": no other system has to be explained and the focus of this paper can remain on connecting business rules to core application.

4.1 The domain of e-commerce

Many of the existing business rules approaches use the domain of e-commerce for illustrating and validating their ideas. We list some policies commonly found in e-commerce applications that are sources of business rules:

- policies for price discounting [14]:
  - refunds
  - returns
  - usage restrictions
  - lead time to place an order
  - cancelling orders
  - creditworthiness, trustworthiness, and authorisation
- policies for personalising links, structure, content and behaviour of web pages [19]
- policies that determine the control flow of online purchases [2]
- policies typically found in online stores (e.g. Amazon web site at amazon.co.uk):
  - recommendations
  - availabilities of products
  - returns
  - delivery rates
  - delivery restrictions

In this paper, we use business rules for price personalisation because this is an increasingly important and pertinent issue, as the special issue of the *Communications of the ACM* [8] shows. Some examples of business rules for price personalisation are:

BR1 If today is Christmas then a customer gets a 5% discount.
BR2 If a customer has purchased more than 20 books then he or she is a frequent customer.
BR3 If a customer is a frequent customer then he or she gets a 10% discount.

Examples of events for these business rules are, respectively:
E1 Before the price of a product is retrieved.
E2 After the customer has checked out.
E3 Before the price of a product is retrieved, and if the customer is checking out.

The last event might need clarification: the price of a product is retrieved in two situations. The first is when the price of the order is calculated at checkout. The second is when a customer is browsing the available products. In the latter case, the standard price of a product is shown, without applying any discounts.

4.2 Design and implementation of an online store

The Patterns for Personalisation [18] and the Rule Object Pattern [1] provide object-oriented patterns for e-commerce applications and business rules in general. We applied these techniques for designing and implementing an online store with price personalisation. The implementation language is Java™. A class diagram of the online store is shown below.

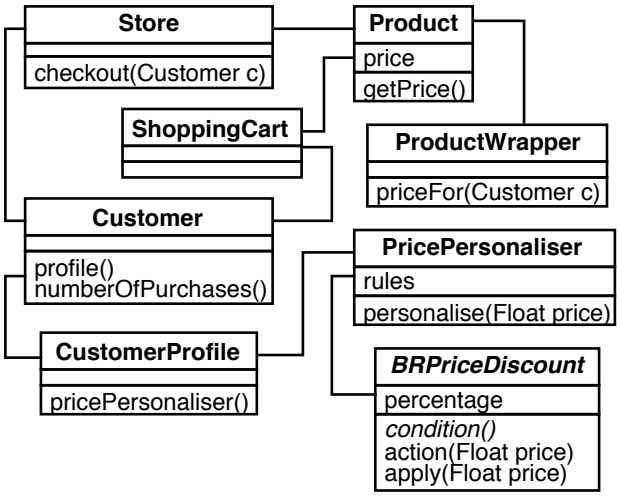

Figure 1: A class diagram of an online store.

This design is conceived to accommodate price personalisation. It assumes that all price personalisation will occur at the event E1 before the price of a product is retrieved which is any call to `getPrice()`. The product wrapper intercepts these calls and delegates them to a `PricePersonaliser`. Each `Customer` has a `Cus-`

tomerProfile which holds the different personalisers needed in the system, such as a PricePersonaliser and a ContentPersonaliser (not shown in the figure) for personalising the representation of product information to the customer. The use of explicit business rules is facilitated because they are collected in the corresponding personalisers. Different customer profiles may require different sets of business rules, which can be customised in the personalisers.

4.2.1 Price personalisation

In this model, a price personalisation business rule is implemented in Java™ by a single class implementing the methods condition() and action(Float price), shown below. Note that this is just a simple manifestation of the Rule Object Pattern. The abstract class BRPriceDiscount 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}; } } }
```

4.2.2 A simple discount

The first concrete price personalisation business rule is BR1 *if today is Christmas then a customer gets a 5% discount*. The implementation in Java™ of this business rule, BRChristmasDiscount, is shown below. It implements condition() by testing if today is Christmas. This business rule fits into this design naturally because it only needs globally available information to evaluate its condition and it is applied at the anticipated event.

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

However, adding the next example business rules and events to this model is more problematic. We elaborate in the next section and derive general requirements for encapsulating the business rule link from those examples.

5. Requirements for encapsulating the connector

This section lists the requirements that should be fulfilled by potential approaches that want to support encapsulated connectors between business rules and core application. For each requirement (safe the last one) we first provide an example that builds on the examples introduced in the previous section to illustrate the problem, and then describe the support required to solve this problem.

The examples are not coded but described, and clearly show that statements need to be added in different places of the core application implementation. Moverover, the new code cannot be encapsulated in the decomposition mechanisms of the language, in this case objects.

5.1 Dynamic events

5.1.1 Example: an event depending on control flow

In order to express the more sophisticated event E3 *before the price of a product is retrieved, and if the customer is checking out*, code needs to be added that crosscuts the other business objects. Typically this event is expressed by adding a flag which is set to true if a checkout process is started, in other words when checkout() is called in Store. An extra conditional evaluates this checkout flag in ProductWrapper before personalising the price of a product.

5.1.2 Requirement: dynamic events

The events denote well-defined points in the execution of the core application where a business rule should be applied. They can be distributed in the core application, for example in objects with different types. These dynamic points depend on properties only available at run-time, such as control flow in the example. Since business rules change often and new ones are added regularly, it is generally not possible to anticipate their events. In current approaches, explicit hooks for the events have to be foreseen in the core application. Hence, a mechanism is needed that allows specification of dynamic events that may depend on properties available at run-time, without having to change the source code manually.

5.2 Introduction of unanticipated business objects

5.2.1 Example: unanticipated attribute and event

Consider the business rule BR2 *if a customer has purchased more than 20 books then he or she is a frequent customer*. Although this business rule expects the class `Customer` to have a boolean attribute `frequent`, this is not anticipated in the original design. Moreover, this business rule requires a new and unanticipated event E2 *after the customer has checked out*. Each time a customer's purchase is confirmed, more specifically after each call to `checkout()` in `Store`, this business rule should be applied in order to establish if the customer can be labelled as frequent or not. Usually, these changes have to be applied manually in the original source code. They are unanticipated and they result in crosscutting code.

5.2.2 Requirement: introduction of unanticipated business objects

When the need for unanticipated state and behaviour arises, a mechanism should be available for introducing new business objects, attributes and operations in the core application, without having to manually alter the original source code. The new code should be encapsulated so that it can be reused or removed easily.

5.3 Exposing business objects

5.3.1 Example: unavailable business objects

The business rule BR3 *if a customer is a frequent customer then he or she gets a 10% discount* is introduced. As is usually the case, this business rule depends on properties of certain business objects and not just on a global property such as the date. Unfortunately, these business objects are mostly not in the scope of the event, which is where they are needed for the business rule application. Generally, capturing this unavailable data at one point and retrieving it at the event involves introducing a global variable or outfitting all methods in the control flow between those points with an extra parameter. Either way this again results in crosscutting code.

5.3.2 Requirement: exposing business objects

The problem of business objects needed in, but only available outside the current scope arises in two situations: specifying an event and applying a business rule. In the first case this occurs when the event depends on run-time properties not accessible in its scope, such as control flow information. The second case occurs when properties from business objects available outside the scope of the event are needed for the application of a business rule. In order to support both cases a mechanism is needed that captures data at the point when it is available and exposes it to the event.

5.4 Configurability and reusability of the connector

Descriptions of events, introduced data structures or captured data should be configurable and reusable. It is possible, for example, that the event is reused with other business rules, which need different data for their application than the previous ones. Inversely, the same business rule might be applied at a different event.

6. Aspect-oriented programming for business rules

A software application involves many and heterogeneous concerns. By concerns we refer to properties or areas of interest in the system. Typically, concerns can range from high-level notions like security and quality of service and low-level notions such as caching and buffering. They can be functional, like business rules or non-functional such as synchronisation and transaction management [12]. To deal with all these heterogeneous concerns in a software application *Separation of Concerns* is fundamental. It refers to the ability to identify, encapsulate and manipulate only those parts of the software that are relevant to a particular concept, goal or purpose [17].

In many situations, separation of concerns is not easy to achieve using the modularisation constructs provided by current languages and environments. Current software engineering techniques generally provide a dominant decomposition mechanism that is not suitable to capture and represent all kinds of concerns that can be found in a software application. This problem is identified in as the [22] "*tyranny of the dominant decomposition*". Therefore, it is impossible to encapsulate and manipulate all the diverse and heterogeneous concerns in only one of the decomposition mechanisms.

In particular, the "system-wide" concerns, i.e. concerns that *cut across* the natural modularity of the rest of the implementation [16], often do not fit nicely into the modularisation mechanism of the technique used. Common examples of crosscutting concerns are design or architectural constraints, systemic properties or behaviours (e.g., logging and error recovery, synchronisation). As a consequence, the code that addresses crosscutting concerns is not localised but

scattered across many places in the entire application and usually they appear even *tangled* with code that addresses other concerns.

Aspect-Oriented Software Development (AOSD) [3] appears as a new software development approach that focuses on the separation of crosscutting concerns. AOSD allows us to better programme software applications by separately specifying the various concerns of a system and some description of their relationships, and then providing mechanisms to weave or compose them together into a coherent programme [12]. AOSD helps to avoid tangling of concerns by explicitly capturing, representing and manipulating them as first entities. In this way, the target application code is easier to develop, understand, reason about, maintain and reuse.

AOP focuses on mechanisms for simplifying the separation of crosscutting concerns in software applications at the implementation level. AOP addresses the problem of the dominant decomposition by implementing the *base* programme (addressing the dominant concern) and several aspect programmes (each addressing a different cross-cutting concern) separately and then *weaving* them all together automatically into a single executable programme.

The requirements for separating the business rules connectors indicate that an aspect-oriented approach is needed, since aspects are ideal for encapsulating the crosscutting connector code. Therefore, our goal is to demonstrate the added benefit of AOP in this context, using the concrete online store example. Of all the existing aspect-oriented approaches, the most important ones for separating general-purpose aspects are discussed in a recent issue of the *Communications of the ACM* [7]. Although these approaches all support AOP, the provided mechanisms can be so fundamentally different that it is impossible to come up with a general and approach-independent aspect-oriented solution for separating business rules connectors [21]. Therefore, we select one specific approach to show that the use of AOP improves the separation of the business rule connection to the core application. *AspectJ,* an aspect-oriented extension to Java™, is not only the most mature of the available aspect-oriented approaches, but also the one with features which match closest with the requirements we discovered.

In the following, we briefly introduce AspectJ and its suitability after which we show how it complements or replaces the objects and patterns in the implementation of the online store example. The example business rules and events are considered again one by one, this time implemented using aspect-oriented programming.

6.1 AspectJ

AspectJ is a simple general-purpose extension to Java™ that provides, through the definition of new constructors, support for modular implementation of crosscutting concerns. It enables plug-and-play implementations of crosscutting concerns [16]. AspectJ has been successfully used for cleanly modularize implementations of crosscutting concerns such as tracing, contract enforcement, display updating, synchronisation, consistency checking, protocol management and others.

The identified requirements reveal that in order to encapsulate the business rule connectors we should be able to identify events, i.e. well-defined points in the execution of the core application. An event can be modelled as a *join point* in AspectJ and a *pointcut* is used to denote a set of join points. *Advice* is used to specify code that should be executed at each of the join points of a pointcut and as a result can be used to trigger the application of a business rule. Moreover, AspectJ's pointcut parameter mechanism allows us to expose business objects available at the execution context of a join point to advice which can be used for passing unavailable business objects to an event. An aspect combines dynamic crosscutting (pointcuts and advices) and static crosscutting, which introduces Java™-like declarations of new attributes and operations in existing classes. AspectJ achieves the above in a non-invasive way, i.e. the programmer does not have to change the source code manually in different places.

For an overview of AspectJ which discusses these constructs thoroughly, we refer the reader to [16] and the web site aspectj.org.

6.2 Design decisions

As before, the business rules themselves will be implemented with objects. This allows for more elaborate Rule Object Patterns to be applied when the number of business rules increases and they become more complex. Aspects will be used for encapsulating the code that connects the business rules to the core application. However, because parts of this connector should be reusable in other situations, several aspects are created for each connector. Typically, there will be an aspect expressing the event, an optional aspect exposing unavailable or introducing unanticipated business objects to the event, and a last aspect that puts the previous ones together and actually applies the business rule. Each of these three 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 implemented in the core application). The implementation of the last aspect will differ depending on the second aspect.

6.3 The online store revisited

Reflecting the design decisions outlined above, the naming conventions used below are:

- names of business rule classes start with BR (as before)
- 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

6.3.1 A non-invasive simple discount

As a first demonstration of how AspectJ is used, a new and non-invasive implementation of the event and the application of `BRChristmasDiscount` is presented.

The first aspect denotes the event. To achieve this it specifies a pointcut `priceCalculation(Product p)` capturing each call to `getPrice()` and exposing the target of this method, an instance of `Product`.

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

Since no data is needed besides the current date, which is available at application time, no aspect for exposing or introducing business objects is defined.

The only other aspect needed holds the business rule to be applied and defines an around advice on the join points designated by the pointcut `priceCalculation(Product p)` from the first aspect `EpricePersonalisation`. This pointcut exposes the instance of `Product` to the body of the advice where it is used to obtain the standard price (by calling `proceed(p)`) after which the resulting price is passed to the business rule when it is applied (by calling `apply(price)`).

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

The application aspect is initialized with the business rule: `ApplyChristmasDiscount.setBR(new BRChristmasDiscount())`.

6.3.2 Dynamic events

The more sophisticated event that depends on the control flow is presented next. In addition to the first pointcut `priceCalculation(Product p)` a second pointcut needs to be specified, `priceCalcInCheckout(Product p)`, that matches join points designated by the first pointcut and that are in the dynamic scope of join points matching the pointcut designated by the execution of `checkout(Customer c)`. This is done using the primitive pointcut designator `cflow`.

```
aspect EPricePersonalisation{
 pointcut priceCalculation(Product p):
  target(p) && call(Float Product.getPrice());
 pointcut priceCalcInCheckout(Product p):
  cflow(execution(Float
    Store.checkout(Customer))) &&
    priceCalculation(p); }
```

6.3.3 Introduction of unanticipated business objects

When adding the next business rule, BR2 *if a customer has purchased more than 20 books then he or she is a frequent customer*, unanticipated business objects are needed and an unanticipated event, E2 *after the customer has checked out*, has to be considered. It implements the three typical aspects: one for the event, one for extending the core application with unanticipated business objects and one for applying the business rule.

The business rule itself is implemented by the class `BRFrequentCustomer` which has a condition that checks if the number of purchases of a customer is bigger than 20, and an action which marks the customer as being frequent. Note that this business rule does not personalise the price, hence it does not extend `BRPriceDiscount` and implements its own `apply(Customer c)`.

```
public class BRFrequentCustomer
 public boolean condition(Customer c){
  return(c.numberOfPurchases() > 20); }
 public void action(Customer c){
  c.becomeFrequent(); }
 public void apply(Customer c){
  if (condition(c)){
   action(c); } }
```

The first aspect `ECheckout` shown below designates calls to `checkout(Customer c)` as the events and exposes the instance of `Customer` at those points.

```
aspect ECheckout{
 pointcut checkout(Customer c):
  args(c) &&
  call(Float Store.checkout(Customer)); }
```

The second aspect `ExtendCustomer` introduces the unanticipated attribute frequent in `Customer` and the operations `isFrequent()` and `becomeFrequent()` for getting and setting the attribute respectively.

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

The third aspect `ApplyFrequentCustomer` again

holds the business rule and applies it after the events designated by the pointcut `checkout(Customer c)` in the first aspect `ECheckout`. Note that the instance of `Customer` exposed by this pointcut is passed to the advice and used there in the application of the business rule.

```
aspect ApplyFrequentCustomer{
 static BRFrequentCustomer businessRule;
 static public void
    setBR(BRFrequentCustomer br){
  businessRule = br; }
 after(Customer c):ECheckout.checkout(c){
  return businessRule.apply(c); } }
```

6.3.4 Exposing business objects

Finally, the third business rule BR3 *if a customer is a frequent customer then he or she gets a 10% discount* needs the customer, a business object anticipated in the core application, but which is unavailable at the application time.

The business rule `BRFrequentDiscount` is again a price personaliser thus inherits from `BRPriceDiscount`, reusing `apply(Float price)` and `action(Float price)` and providing a concrete implementation for `condition()` which tests if the `Customer` is frequent. Since it needs an instance of `Customer` at that time, it foresees an attribute `customer`.

```
public class BRFrequentDiscount
    extends BRPriceDiscount{
 Customer customer;
 public void setCustomer(Customer c){
  customer = c; }
 public boolean condition(){
  return customer.isFrequent(); } }
```

The first aspect, for applying this business rule, is `EPricePersonalisation` introduced in Sec. 6.3.1.

The second aspect `CaptureCustomer` exposes the instance of `Customer` available at every call of `checkout(Customer c)` to `Store`. This aspect sets customer with the captured instance of `Customer` in `BRFrequentDiscount`. Note that `ApplyFrequentDiscount.aspectOf()` picks the correct aspect instance to be associated with the correct business rule instance.

```
aspect CaptureCustomer{
 pointcut checkout(Customer c):
  call(public Float Store.checkout(Customer))
   && args(c); }
 before(Customer c):checkout(c){
  BRFrequentDiscount br =
    new BRFrequentDiscount();
  br.setCustomer(c);
  ApplyFrequentDiscount.aspectOf().setBR(br);}}
```

The third aspect `ApplyFrequentDiscount` increases in complexity because the business rule it applies has state. For each join point denoted by the pointcut `checkout(Customer)` in the aspect `CaptureCustomer`, a new `BRFrequentCustomer` has to be instantiated. Each of these business rule instances holds an instance of `Customer` that was exposed at that join point. As a result, for each of these join points (and also `BRFrequentCustomer` and `Customer` instances) a new instance of the aspect `ApplyFrequentDiscount` has to be created. This is achieved by the `percflow` construct which instantiates an aspect for each join point designated by the pointcut it takes as parameter.

```
aspect ApplyFrequentDiscount
  percflow(CaptureCustomer.checkout(Customer)){
 BRFrequentDiscount businessRule;
 public void setBR(BRFrequentDiscount br){
  businessRule = br; }
 Float around(Product p):
  EPricePersonalisation.priceCalcInCheckout(p){
  Float price = proceed(p);
  return businessRule.apply(price); } }
```

7. Related work

We refer to related work with respect to business rules approaches that can be used in the context of object-oriented software applications in the paper. 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.

In previous work, we separated business rules for ensuring the quality of geographic data from the core application [24][5].

We do not discuss related work concerning AOP, since the purpose of this paper is to show that AspectJ as a representative approach is able to meet the requirements we presented adequately, not to compare aspect-oriented approaches in their ability to fulfil the requirements.

8. Conclusions

This paper identifies a fundamental problem with current state of the art in business rules approaches in the context of object-oriented software engineering. We observe that although the business rules are separated from the core application functionality, their connectors are not. This impedes the business rules objectives: separate, trace, externalise business rules and position them for change [23].

This paper does not take in to account all the complex issues involved in business rules, such as for example business rule intereference. However, what we report here is based on our work described in [6], which provides a more complete coverage of those issues.

The most important contribution of this paper is the

identification of a set of general requirements that any support for business rules in an object-oriented software engineering context must meet. 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.

Additionally, we point out that some of the required features are supported by AOP since they provide constructs for encapsulating crosscutting concerns. Hence, another contribution of this work is to show that a suitable and mature approach for AOP, AspectJ, indeed fulfils the requirements. Although it is impossible to generalise the results to all aspect-oriented approaches because some of them are too fundamentally different [21], it is possible to distil at least patterns of AspectJ usage. Due to lack of space, these patterns are not included here, but extensively described in [6].

Although AspectJ fulfils the requirements adequately in the sense that one achieves encapsulation of the connectors, it is not perfect. The last requirement concerning reuse of the connectors is not entirely met. Parts of connectors can be reused in other connectors because they can be referred to, but specialisation through inheritance is not sufficiently supported. Additionally, instantiation, initialization and referencing of aspects could be improved. In general, we can conclude that the existing AOP approaches that are suitable candidates for supporting business rules connectors, offer support that is too low-level because they are general-purpose. This results in a proliferation of aspects (in the case of AspectJ) which are hard to manage. Higher-level abstractions are needed to synchronise the business rules with the core application. This is food for future work.

9. References

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

[2] Arsanjani and J. Alpigini. Using grammar-oriented object design to seamlessly map business models to component-based software architectures. In International Symposium of Modelling and Simulation, Pittsburgh, USA, pages 186--191, 2001.

[3] Aspect-Oriented Software Development.
http://www.aosd.net/

L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. Communications of the ACM, 44(10):51--57, 2001.

[4] The Business Rules Group. Defining Business Rules: What Are They Really?, July 2000.
http://www.businessrulesgroup.org/

[5] M. Casanova, M. D'Hondt, and T. Wallet. Explicit domain knowledge in geographic information systems. In 14th Conference on Software Engineering and Knowledge Engineering (SEKE), Buenos Aires, Argentina. Knowledge Systems Institute, 2001.

[6] 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.

[7] Communications of the ACM. Aspect-Oriented Software Development, October 2001.

[8] Communications of the ACM. The Adaptive Web, June 2002.

[9] C. Date. What not How: The Business Rules Approach to Application Development. Addison-Wesley Publishing Company, 2000.

[10] M. D'Hondt, W. De Meuter and R. Wuyts. Using Reflective Logic Programming to Describe Domain Knowledge as an Aspect. In Proceedings of the First Symposium on Generative and Component-Based Software Engineering. Erfurt, Germany, 1999.

[11] M. D'Hondt and T. D'Hondt. Is Domain Knowledge an Aspect? ECOOP '99, Workshop on Aspect-Oriented Programming. Lisbon, Portugal, 1999.

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

[13] B. N. Grosof, Y. Kabbaj, T. C. Poon, M. Ghande, and et al. Semantic Web Enabling Technology (SWEET).
http://ebusiness.mit.edu/bgrosof/

[14] B. N. Grosof, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: courteous logic programs in xml. In Proceedings of the first ACM conference on Electronic commerce, pages 68--77. ACM Press, 1999.

[15] 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.

[16] 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.

[17] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. Communications of the ACM, 44(10):43--50, Oct. 2001.

[18] 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.

[19] G. Rossi, D. Schwabe, and R. Guimaraes. Designing personalized web applications. In World Wide Web, pages 275--284, 2001.

[20] 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.

[21] P. Tarr, M. D'Hondt, L. Bergmans, and C. V. Lopes. Report from the ecoop2000 workshop on aspects and dimensions of concern: Requirements on, and challenge problems for, advanced separation of concerns. In Workshop Reader of the 14th European Conference on Object-Oriented Programming (ECOOP `00), pages 203--240. Springer-Verlag, 2000.

[22] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In Proceedings of the 1999 International Conference on Software Engineering, pages 107-119. IEEE Computer Society Press / ACM Press, 1999.

[23] B. von Halle. Business Rules Applied. Wiley, 2001.

[24] T. Wallet, M. Casanova, and M. D'Hondt. Ensuring quality of geographic data with uml and ocl. In Third International Conference on the Unified Modeling Language, York, UK, pages 225--239. Springer-Verlag, 2000.

[25] J. Yoder and R. Johnson. The adaptive object-model architectural style. In Working IEEE/IFIP Conference on Software Architecture (WICSA), Montreal, Canada, 2002.