Integration of Application Business Logic and Business Rules with DSL and AOP

Bogumiła Hnatkowska and Krzysztof Kasprzyk

Wroclaw University of Technology, Wyb. Wyspianskiego 27
50-370 Wroclaw, Poland
Bogumila.Hnatkowska@pwr.wroc.pl
Krzysiek.Kasprzyk@gmail.com

Abstract. Business processes and business rules are implemented in almost all enterprise systems. Approaches used today to their implementation are very sensitive to changes. In the paper authors propose to separate business logic layer from business rule layer by introducing an integration layer. The connections between both parts are expressed in a dedicated for that purpose domain specific language (DSL). The definitions in DSL are further translated into working source code. The proof-of-concept implementation of the integration layer was done in the aspect oriented language (AOP) – AspectJ. The AOP was selected because it fits well to encapsulate scattered and tangled source code implementing the connections between business logic and business rules with the source code implementing core business logic.

Keywords: business modeling, business rules, domain specific languages, aspect oriented programming.

1 Introduction

Software systems of enterprise class usually support business processes and business rules existing in a given domain. Because both (business processes and business rules) are often subject of change, they should be defined within a software system in such a way that is easy to maintain.

Approaches used today to business rules implementation are very sensitive to changes, i.e. each modification of: (a) business rule set (b) when (within a business process) to fire specific business rules (c) which business rules to fire – can result in the necessity of application source code modification. Even if the business rule engine is used for business rules management, the source code implementing the connections between business logic and business rules is scattered and tangled with the source code implementing core business logic. That allows to treat the problem of integration between business logic layer and business rules (considered as a separate layer) as a cross-cutting concern. A mechanism usually used for separation of cross-cutting concerns within software systems is Aspect Oriented Programing (AOP) [7], and one of the most popular AOP programming languages is AspectJ [7].

In the paper [4] authors presented an idea of an intermediate layer between business logic layer and business rules layer that eliminates the problem of source code scattering and tangling. The layer is implemented in AspectJ. Unfortunately, aspect-oriented

T. Szmuc, M. Szpyrka, and J. Zendulka (Eds.): CEE-SET 2009, LNCS 7054, pp. 30–39, 2012.

languages are rather difficult, so the source code of intermediate layer is complex and hard to understand. Therefore there is a need for more abstract language (rather declarative one) which can be used for describing how to integrate business logic with business rules. In this paper authors present a domain specific language (DSL) serving that purpose. Models written in the DSL are automatically translated to AspectJ source code. The DSL editor with syntactic checks as well as transformations were implemented in the oAW framework [8].

The structure of the paper is as follows. In Section 2 main features of integration layer are presented. In Section 3 the DSL syntax shortly is described. A short but complete case study is shown in Section 4. Section 5 presents related works while Section 6 contains some concluding remarks.

2 Features of Integration Layer

Business model defines basic notions from a given domain, the relationships between the notions and the way in which they are constrained. Business rules constitutes an important part of a business model. There are many types of business rules, for example von Halle distinguishes [10]: (a) terms – nouns which meaning is commonly accepted, (b) facts – statements defining relationships among terms, (c) rules – declarations of a policy or a condition that must be satisfied. Rules are defined upon terms, and facts, and they are further divided into constraints, action enablers, inferences, and computations. Terms and facts usually are expressed directly in the source code of application. If they are changed also the source code is modified. Rules can be implemented either directly in the application source code or outside it. Using today approaches to rules realizations try to separate them into some components (modules, etc.) to minimize the influence of their changes on the other parts of application. The advantages of rules modularization are as follows:

- Rules are directly expressed
- Rules can be shared between different business processes
- It is easier to define who and when can modify rules
- Rules can be maintained (update, create, delete) not only by programmers but also by business experts

A typical solution to rules modularization employs business rule engines or business rule management systems like JBoss Rules [6], JRules [5], or Jess [3]. However, even in such a case, source code responsible for communication with the engine is scattered and tangled with application source code responsible for business logic. Additionally, every time you decide to use (or not to use) a rule in a given place, you need to modify the application business source code. To eliminate above mentioned problem we have decided to introduce separate layer in the application architecture, between business logic layer and rules representation – see Fig. 1. The main aim of this layer is to isolate the business logic layer from rules. So, this should prevent the business logic layer from being influenced by rules evolving or changing.

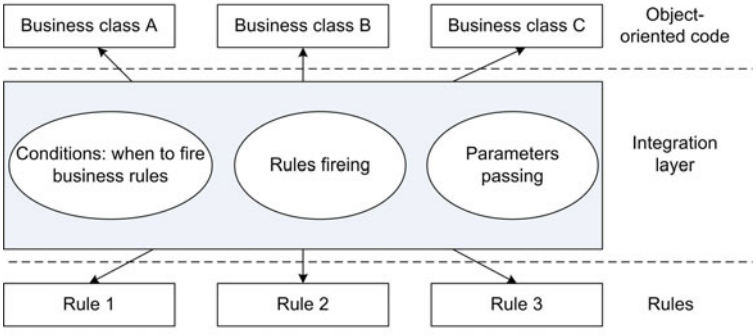

Fig. 1. Integration layer in application architecture

The desired features of integration layer are presented below:

- Support for invocations of all rule kinds
- Definition when to invoke what rules
- Passing parameters and context dependent information to rules

The are two kinds of activation events that can result in rules invocation: (a) invocation of an object business method (b) assigning a new value to an object attribute. The rule can be fired – similarly to aspects – before, after or instead of the method being the source of the event. The activation event can occur in different contexts. There should be a possibility to express different scenarios describing such a context [5], e.g. activation event was raised inside or outside the flow of specific business method.

3 DSL Definition

To hide the complexity of integration layer a textual domain specific language was defined. It allows to define how to integrate business logic with business rules in a declarative way. The full meta-model of the DSL consists of 29 classes and 42 relationships (17 generalizations, 25 associations). The concrete syntax of the language was defined in the form supported by the oAW framework. Models written in DSL are validated against a set of rules expressed in Check language which is a part of the Xtext framework. Transformation between models and AspectJ source code was implemented in Xpand template language. The general structure of text files with DSL definitions is presented in Fig. 2.

```
Package declaration
[Import section]
[Global object declaration]
Event definitions
Business logic to business rules link definitions
```

Fig. 2. General structure of DSL file content

The presentation of the DSL is constrained to mandatory elements.

3.1 Package Declaration

Package declaration has the same form as in java program. It specifies where (to what package) generate AspectJ code.

3.2 Event Definition

Business rules are fired in strictly defined moments during program execution. It is assumed that there are two kinds of activation events: (a) method invocation event (b) attribute change event. Definition of an event activated by method invocation has a form presented in Fig. 3. It defines the unique name for the event and the signature of the method (optionally types of parameters and type of returned value).

```
event <event name> isExecutionOf method
  <method name> in <type name>
  [withParams ( <parameter list> )]
  [returns <type name>]
end
```

Fig. 3. Syntax for method invocation event

As activation event is also responsible for preparing the context to business rules evaluation there is a possibility to exhibit (using `asFact` keyword) some data associated with method execution:

- The object realizing the method
  (`in <type name> asFact <object name>`)
- The value returned by the method
  (`returns <type name> asFact <object name>`)
- The values passed as parameters
  (`withparams(<type name_1>,...,<type name_k>) asFact <object name_1>,...,<object name_k>`)

The data will be used further in link definition (see Subsection 3.3). Definition of an event activated by attribute change has a form presented in Fig. 4. It defines the unique name for the event, the localization (class) and the name of the attribute. Similarly to the activation event there is a possibility to exhibit some data:

- The new value of the attribute
  (`newValue <type name> asFact <object name>`)
- The object which attribute is modified
  (`in <type name> asFact <object name>`)

```
event <event name> isUpdateOf field
  <attribute name> in <type name>
end
```

Fig. 4. Syntax for attribute change event

3.3 Business Logic to Business Rules Link Definition

Business logic to business rules links are the main elements of DSL. They are built upon events and values exhibited by them. Definition of the link determines what business rules when to fire, and optionally the data necessary for business rules evaluation, context of execution etc. – see Fig 5.

```
link <link name>
  [configuredBy <path to configuration file>]
  fires <rule names> <when clause> <event name>
  [requires <object name_1>,...,<object name_k>]
  [active <context definition>]
end
```

Fig. 5. Syntax for business logic to business rules link definition

The most important part of the definition is `fires` clause. It defines the names of business rules that should be fired in a reaction to a specific event. The `when` clause specifies exactly when to run business rules. There are three possibilities to choose from: (a) `before` (rules are called before event activation); (b) `after` (rules are called after event activation); (c) `insteadOf` (rules are called instead of event activation). The `requires` clause is used for passing necessary data identified by names to a rule engine. The order of object names is important because it determines the order of events that are responsible for preparing the objects. The `active` clause defines the context (additional constraints) in which the activation event (defined in `fires` clause) results in business rules invocation. There are many possibilities for context definition, below are presented two of them:

- `while <event name>` – activation event must occur within flow of method defined by an event
- `except <event name>` – activation event must occur outside flow of method defined by an event

4 Case Study

Let consider a simple business process (called *Order Processing*) that aims at processing an order of a given customer. The process consists of four basic operations performed in a sequence: (1) order validation (2) order's total cost calculation (3) order's shipping cost calculation (4) sending an order for further realization. If an order is not validated (result of operation 1), status of the order is set to rejected and the whole business process is stopped; otherwise status of the order is set to accepted, and the process is continued. The business process is constrained with the following set of business rules:

- Rule 1: "Gold" customer pays 40
- Rule 2: "Platinum" customer pays nothing for shipping

The data model and the business layer model (limited to the considered example) of the application supporting *Order Processing* business process is presented in Fig. 6. An order contains a collection of order items, each of which has a price defined. An order is placed by a customer. A customer belongs to one of customer categories (regular, gold, platinum). The main class realizing the business logic is *OrderProcessingService* with *processOrder* operation. The operation implements all four operations defined for the business process – see Fig. 7.

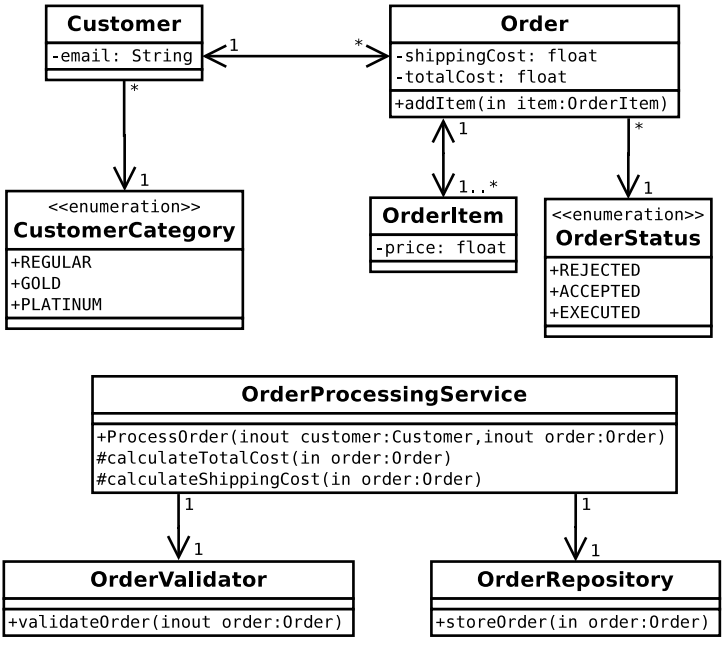

Fig. 6. Data model and business logic layer for considered example

The business rules were defined in DRL language and stored in JBoss engine. Each business rule was given a unique name: Rule 1 – Reduce shipping cost for gold customers, Rule 2 – Reduce shipping cost for platinum customers. An example of rule definition in DRL language is shown in Fig 8.

Business rules should be fired in strictly defined moments of application execution. Rule 1 and Rule 2 should be fired after execution of *calculateShippingCost* method; but only if the method was invoked inside *processOrder* flow. Both rules modify the value returned by the *calculateShippingCost* method basing on specific customer information. Following examples present how to define activation event (Fig. 9), and a link between application business logic and business rules (Fig. 10) in proposed DSL (see Section 3).

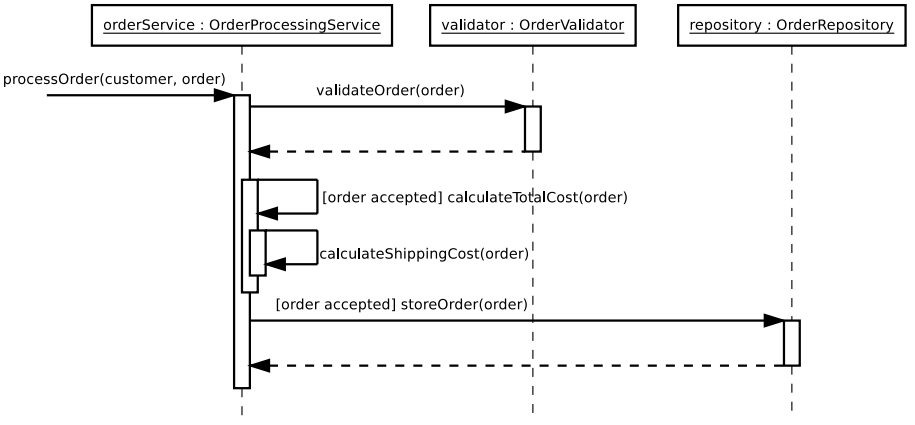

Fig. 7. Order processing realization

```
rule "Reduce shipping cost for gold customers"
when
  order: Order( shippingCost > 0 )
  customer: Customer(category == CustomerCategory.GOLD)
then
  order.setShippingCost(order.getShippingCost()*0.6f);
end
```

Fig. 8. Rule 1 definition in DRL language

```
event ShippingCostCalculation isExecutionOf method
   calculateShippingCost
   in OrderProcessingService withParams (Order)
end
```

Fig. 9. DSL definition of activation event for Rule 1 and Rule 2

```
link CustomizeShippingCost
  fires "*shipping cost*" after ShippingCostCalculation
  requires customer newOrder
  active while OrderProcessing
end
```

Fig. 10. DSL definition of a link between application logic and business rules

```
package pl.wroc.pwr.casestudy.aspects;
import pl.wroc.pwr.casestudy.domain.Customer;
...
import org.drools.StatelessSession;
...
public aspect CustomizeShippingCostAspect
  percflow(execution(
  void OrderProcessingService.processOrder(Customer, Order))) {
   private Customer customer;
   private Order newOrder;
   private int capturedFacts = 0;
   private static int getCapturedFacts() {
    if (CustomizeShippingCostAspect.hasAspect()) {
      return CustomizeShippingCostAspect.aspectOf().capturedFacts;
    } else {
      return -1;
}}
before(Customer customer, Order newOrder) :
  execution(void OrderProcessingService.processOrder
  (Customer, Order))&& args (customer, newOrder)
    && if (getCapturedFacts() == 0) {
      this.customer = customer;
      this.newOrder = newOrder;
      this.capturedFacts++;
  }
pointcut shippingCostCalculationPointcut() :
  execution(void OrderProcessingService.calculateShippingCost(Order))
  && cflow(execution(
  void OrderProcessingService.processOrder
  (Customer, Order))) && if (getCapturedFacts() == 1);
after() : shippingCostCalculationPointcut () {
  RuleAgent agent = RuleAgent.newRuleAgent("config.properties");
  RuleBase ruleBase = agent.getRuleBase();
  StatelessSession session = ruleBase.newStatelessSession();
  session.setAgendaFilter(
    new RuleNameMatchesAgendaFilter("*shipping cost*"));
  try {
    session.execute(new Object[]{customer, newOrder});
  } catch (ConsequenceException exc) {
    Throwable originalException = exc.getCause();
    if (originalException instanceof RuntimeException){
       throw (RuntimeException) originalException;
    } else {
       throw exc;
}}}}
```

Fig. 11. Aspect (a part) generated for *CustomizeShippingCost* link

Business rules (Rule 1, Rule 2) are identified based on part of their names ("*shipping cost*" regular expression). The DSL definition is automatically transformed to AspectJ code. Fig. 11 presents result of such transformation.

5 Related Works

Other researchers have also noticed a relationship between crosscutting nature of business rules and aspect-oriented paradigm. In [2] authors analyze if domain knowledge can be treated as one of the system's aspects that should be developed and maintained independently of other concerns. In [7] author investigates how to move implementation of business rules from core business classes to aspects. Two business rules for a system that supports transferring funds between bank accounts were defined and implemented in AspectJ language. Conducted analysis is neither thorough nor systematic but shows that aspect-oriented programming language can support implementation of business rules in object-oriented systems. In [1] an approach for expressing business rules at a high level of abstraction is presented. A high-level business rule language and high-level business rule connection language were defined. Proposed solution is similar to ours because it uses aspect-oriented paradigm to encapsulate source code that connects business rules with the core application. Each high-level rule is automatically mapped to a java class, but only inferences and computations are supported. JAsCo [9] aspect-oriented language is used for low-level implementation of connections between rules and application. For each rule connection specification an aspect bean and a connector are generated. Our approach differs from that one presented in [1] mainly because of different technologies (JBoss instead of pure Java) and languages (AspectJ instead of JAsCo) used in a proof-of-concept implementation of the integration layer. Moreover, our DSL for integration layer is more flexible and expressive than the high-level business rule connection language proposed in [1]. It supports all kinds of business rules, allows to more precise activation event's context definition and offers better support for capturing business objects within business processes realizations.

6 Conclusions

Authors have proposed to use a specific DSL for describing dependencies between application business layer and business rules. At that moment only two types of events that result in a business rule invocation are identified (method call and attribute change). Introduction of a new event kind must be followed with extension of both, DSL syntax and DSL-to-code transformations.

Applying proposed DSL for the integration layer has many advantages. It allows to define connections between rules and business process at higher abstraction level in a declarative way. The syntax is easy and very flexible. The proof-of-concept implementation proved that the reduction above 70% in source code line numbers is possible. The solution is platform independent, so – if something changes at implementation level it will only have influence on model-to-code transformations. The transformations are complete in the sense that obtained aspect definitions need not to be changed by programmers.

The main disadvantage of DSL is that to apply it successfully you need to know the business classes, relationships among them, the semantics of their methods and the interactions among instances. Therefore, the obvious direction of further research is a formalization of business rules and business processes, that allow to abstract from their concrete implementations.

References

1. Cibran, M.A., D'Hondt, M.: High-level Specification of Business Rules and Their Crosscutting Connections. In: Proc. of the 8th International Workshop on Aspect-Oriented Modeling at the International Conference on Aspect-Oriented Programming, AOSD 2006 (2006), http://dawis2.icb.uni-due.de/events/AOM_AOSD2006/Cibran.pdf
2. D'Hondt, M., D'Hondt, T.: Is Domain Knowledge an Aspect? LNCS, vol. 1743, pp. 293–294. Springer, Heidelberg (1999)
3. Friedman-Hill, E.: Jess in Action: Rule-Based Systems in Java. Manning Publications (2003)
4. Hnatkowska, B., Kasprzyk, K.: Business Rules Modularization with AOP (in Polish). In: Proc. of the 11th Software Engineering Conference KKIO 2009, WKŁ, Warsaw (2009)
5. ILOG JRules, http://www.ilog.com/products/jrules
6. JBoss Rules, http://www.jboss.com/products/rules
7. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications (2003)
8. OpenArchitectureWare User Guide, http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents
9. JAsCo language documentation, http://ssel.vub.ac.be/jasco
10. von Halle, B.: Business Rules Applied – Building Better Systems Using the Business Rules Approach. Wiley (2002)