Integrating Rules with Object-Oriented Software Applications using Aspect-Oriented Programming

María Agustina Cibrán, Davy Suvée, Maja D'Hondt, Wim Vanderperren, Viviane Jonckers

System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel (VUB)
Pleinlaan 2
1050 Brussels
Belgium
{mcibran,dsuvee,mjdhondt,wvdperre}@vub.ac.be, vejoncke@info.vub.ac.be

Abstract. Literature on developing software applications with rule-based knowledge, or business rules, advocates separating rules from the object-oriented core functionality. Many technologies exist that pursue this goal, although they take radically different approaches to physically separating rule definitions from object-oriented programs. However, we observe that the integration code is still tangled in the programs themselves. Separating and encapsulating the integration code is not straightforward: we identify a number of issues that need to be addressed. This paper reviews these issues and determines a number of features, both at the language and the technological level, which – according to us – are required to address these issues. Since the principle of Aspect-Oriented Programming (AOP) advocates the encapsulation of tangled code, we examine several state-of-the-art AOP approaches. This paper discusses, for each of the identified issues, the different kinds of features provided by these approaches.

KEYWORDS: Object-Oriented Software Engineering, Business Rules, Aspect-Oriented Programming

1 Introduction

The real-world domains of many software applications, such as e-commerce, the financial industry, television and radio broadcasting, hospital management and rental business, are inherently knowledge-intensive. Part of this knowledge is rule-based, which typically represents knowledge about policies, preferences, decisions, advice and recommendations, to name just a few. Current software engineering practices result in software applications that contain *implicit* rule-based knowledge, which is *tangled* with the object-oriented core functionality. Nowadays, rule-based knowledge has become a hot topic and is also referred to as *business rules* [8, 21, 26].

When considering literature on developing software applications with rule-based knowledge, or business rules, we find that all advocate making rules explicit and separating them from the object-oriented core functionality [8, 21, 26]. Moreover,

many technologies exist that are targeted towards these goals, although they take radically different approaches.

This abundant choice of approaches notwithstanding, we observe that the actual tangling of rules and object-oriented functionality is not resolved by any of them. Although the rule definitions – in whichever format or language – are physically separated from the object-oriented program, the code that integrates them is still tangled in the programs themselves. In order to truly separate them, the integration code should be *encapsulated*. This enables the development of alternative integrations when either rules or core functionality changes due to maintenance, reuse or evolution.

Encapsulating the integration code is not straightforward: we identify a number of issues that need to be addressed. This paper reviews these issues and determines a number of features, both at the language and the technological level, which – according to us – are required to address these issues. Since the principle of Aspect-Oriented Programming (AOP) [7, 16] advocates the encapsulation of tangled code, we examine several state-of-the-art AOP approaches. This paper discusses, for each of the identified issues, the different kinds of features provided by the approaches.

The next section briefly introduces business rules using some straightforward examples. Section 3 presents the set of requirements which are essential in order to successfully encapsulate the rule integration code. Section four introduces the ideas and concepts behind AOP and demonstrates how its features are ideal to be used in the context of encapsulating rule integration. This is achieved by investigating several state-of-the-art AOP approaches, such as AspectJ [2], HyperJ [19], JAsCo [24], JAC [20], AspectWerkz [3] and JBoss/AOP [11]. Finally, we present some related work and end with the conclusions.

2 Business Rules Example

In order to introduce business rules, consider a simple e-commerce application which allows buying books online. This shop defines a price personalization policy with the following discount rules:

- **Rule1**: If today is Christmas, then a customer gets a 5% discount.
- **Rule2**: If a customer has purchased more than 20 books, then he or she becomes a frequent customer.
- **Rule3**: If a customer is a frequent customer, then he or she gets a 10% discount.

Typically rules are applied at events, which denote well-defined points in the execution of the core functionality. Example events are the following:

- **Event1**: Before the price of a product is retrieved.
- **Event2**: After the customer has checked out.
- **Event3**: Before the price of a product is retrieved while the customer is checking out.

Thus, Rule1, Rule2 and Rule3 can be respectively applied at Event1, Event2 and Event3, for instance.

A rule is applicable when its condition is satisfied. When a rule is applicable, its action can be performed. Moreover, performing the action of a rule might depend on the applicability of other rules. Suppose Rule1 and Rule3 mutually exclude each other and that whenever Rule1 and Rule3 are applicable, only the action specified by Rule1 will be performed, i.e. the Rule1's discount will be applied and not Rule3's discount.

3 Requirements for encapsulating rule integration

In this section, we describe and motivate the set of requirements that are essential for a technology to be suitable to cleanly encapsulate the rule integration code and achieve high flexibility in the integration of rules. These requirements are described independently from concrete implementation languages and/or technologies.

Encapsulation of crosscutting behaviour
Typically, rules are applied at different points in the core functionality. Many times, the concrete integration code is identical at all these points. Using current approaches that separate rules however, the core program needs to be adapted manually each time rules have to be integrated. In addition, extra code is needed to capture the required information for the deployment of the rules. For instance, it might be needed to add additional parameters to existent operations in order to pass and make available the required information for the application of the rules. This causes tangled and scattered code. As a consequence, rule integration results tangled and scattered and thus, crosscuts the core functionality. Consequently, the clean encapsulation of this crosscutting code in separate modules is required.

Run-time identification of dynamic events
The events at which rules need to be deployed usually represent dynamic points in the core functionality such as method invocations and property accesses. These events are scattered among many places in the core functionality and depend on properties only available at run-time, such as the control flow in the case of Event3. Moreover, since rules change often and others are added or removed regularly, it is generally not possible to anticipate the events at which they would need to be applied. Hence, a mechanism is required that allows the run-time specification of dynamic events, without having to change the source code manually.

Capturing and exposing data at dynamic events
Some rules only need global information directly available in the system, such as the current system date. In this case, this information is directly accessible by the rules. Other rules need data from objects that are in the scope of the event that activates the rules. In this case, these objects are available at those points but not directly accessible by the rules. Thus, a mechanism is needed to capture and pass them to the rules. Moreover, there are rules that require information from specific objects that are outside the scope of the event which activates the rules. Rule3 for instance, depends

on properties of the customer that might not be directly accessible at the moment the rule is applied at Event3, when the price of a product is retrieved. Consequently, a mechanism is required, which allows identification of the points in the core functionality where these objects are available, and which captures and exposes these objects to be used by the rules.

Introduction of unanticipated data and behaviour
Like mentioned in the previous requirement, rules need specific information in order to be applied correctly. Some rules however, require information that was not foreseen at the moment the core functionality was designed and implemented. Rule2 for instance, expects the class *Customer* to have a *frequent customer* attribute and a method, e.g. *is frequent*, to consult the value of this attribute. This property and method, however, may not be anticipated in the original core functionality with which this rule is integrated. When the need for unanticipated data and behaviour arises, a mechanism is required for introducing new objects, attributes and operations in the core functionality, without having to alter the original source code manually. In addition, these new additions should be encapsulated so that they can be reused and removed easily.

Sharing of context information
Once the required data for the activation of the rules is obtained, it needs to be passed along to the events at which the rules are applied. This implies the need for a communication/interaction mechanism between the different modules that encapsulate the rule integration such that these can share and pass along the required information to the rule.

Controlling instantiation and initialization
It should be possible to control the instantiation of rules so that initializing a rule with specific properties for a particular rule integration is enabled. Considering the volatility of rules, this is a vital requirement, as it allows customizing application-independent rules to conform to a specific integration.

Specifying precedence, combination and execution strategies
Some rules deployed within a software application may state policies that interfere with other rules. To avoid rule interference, the collaboration among rules needs to be managed. In addition, some rules can have precedence over others or should not be applied when others are deployed. In the example, Rule1 excludes Rule3. As such when both rules are applicable (both conditions are satisfied), only Rule1's discount has to be applied. In order to address these complex interdependencies among rules, combining and prioritizing the modules that encapsulate the rule integrations is required. Moreover, we need to be able to explicitly control the application of the rules.

Dynamic pluggability of crosscutting behaviour
New rules might be considered that were not identified at development time. Others might become obsolete after a particular period and should be removed. At the end of

a season for instance, the sales period starts and shops consider new price discount rules. Rules such as Rule2 might be considered only during the sales period and not during the rest of the year. As a result, it should be possible to deploy a rule at run-time and to remove it when it is not longer desired. To reflect this volatility, a technology that allows the dynamic addition and removal of data and behaviour is required.

Dynamic customization
Rule-based knowledge tends to change independently of the core functionality. The frequent customer discount assigned by Rule3 could for instance be subject to change over time. As a result, it should not only be possible to add and remove rule integration code at run-time, but also to adapt their properties and behaviour dynamically, such that these are able reflect the changes in rule-based knowledge of a company.

4 AOP for encapsulating rule integration

4.1 Introduction to AOP

Aspect-Oriented Programming (AOP) is a new development paradigm that aims at achieving a better separation of concerns than possible using standard object-oriented software engineering methodologies. AOP claims that some concerns of an application cannot be cleanly modularized as they are scattered over or tangled with the different modules of the system [16]. Similar logic is thus repeated in different modules and due to this code duplication, it becomes very hard to add, edit and remove such a crosscutting concern in the system. AOP proposes to capture such a crosscutting concern in a new kind of module construct, called an aspect. An aspect typically consists of a set of points in the base program where the aspect is applicable (called *joinpoints*) and the concrete behaviour that needs to be executed at those points (called *advice*). Aspect weaving consists of merging the aspects with the base implementation of the system. Nowadays, several mature AOP approaches are available and aspect-oriented programming starts getting worldwide industrial acceptance [1]. Examples of AOP approaches are AspectJ [2], JAC [20], JBoss/AOP [11], JAsCo [24], AspectWerkz [3], and HyperJ [19].

4.2 Discussion

The aspect-oriented idea seems to be ideal to encapsulate the inherently crosscutting and extremely volatile rule integration code. We investigate whether the current aspect-oriented approaches offer the required features we identified in section 3. This section assesses state-of-the-art AOP approaches against these requirements.

Encapsulation of crosscutting behaviour
As identified in section, rule integration crosscuts the core functionality. Aspects are meant to encapsulate the implementation of crosscutting concerns and as such appear suited to modularize the crosscutting rule integration. Aspects in the approaches JAsCo, JAC, AspectWerkz and JBoss/AOP are implemented as fully independent modules. They are completely independent and reusable entities. Even at run-time, the aspects remain first-class entities independent from the core functionality. In HyperJ and AspectJ, the aspects are physically woven into the core functionality, embedding the advices in the base behaviour. This makes the aspect again crosscutting at run-time. As such, aspects loose their identity at run-time and it is in principle impossible to refer to the aspect entity directly [17]. In AspectJ, it is even impossible to separately compile an aspect. As such, when developing an aspect library for third-parties, the aspects are necessary open source. In addition, when the aspect logic has to be altered, the complete application has to be rewoven; this gives raise to scalability issues when a multitude of aspects are present in large scale applications. In JAsCo, JAC, AspectWerkz and JBoss/AOP, aspects can change independently and reflect those changes directly in the core functionality, without the need to be reintegrated; it suffices to recompile the aspects.

Identification of dynamic events
The identification of dynamic events in the core functionality where the rules have to be activated is a feature supported by almost all the approaches except for HyperJ in which the weaving of aspects is fundamentally different than the other approaches. All other approaches achieve the identification of dynamic events in a non-invasive way, i.e. the programmer does not have to change the source code manually in different places. AspectJ's joinpoint language is the most expressive one, since it allows capturing very sophisticated dynamic events including control flows and execution of exception handlers.

Capturing and exposing data at dynamic events
Rules often require information which is dependent on the dynamic context. Therefore, it should be possible for a rule to obtain and analyze the context of the event that triggered its execution. All investigated AOP approaches allow to analyze the context of the joinpoint that triggered the execution of the aspect behaviour. For instance, one can query the name of the method invocation, the supplied arguments and the object on which the method was called. In many cases this expressive power suffices for providing the business logic with the necessary information. If the data objects required by the rules are outside the scope of the joinpoint that triggered the rule, other aspects can be employed in order to capture those data objects. In the example, Rule3 requires the customer object that is checking out in order to verify whether he or she is a frequent customer. However, this customer object is not directly accessible when the price of a product is retrieved. Thus another aspect can be defined which defines a joinpoint to capture the moment when the method for checking out is invoked and the customer object is still available. At this joinpoint, the customer object can be retrieved and passed to the rule. Note that two different aspects need to collaborate in order the rule to be successfully applied.

Introduction of unanticipated data
Some rules require the introduction of new business objects and behaviour in order to execute their business logic appropriately. Introducing new methods and attributes is sometimes required when crosscutting concerns need to be encapsulated. Several AOP approaches support the concept of introduction. The open classes feature (previously named "static crosscutting") provided by AspectJ allows the insertion of fields and methods. It also allows extending classes from specific superclasses and interfaces from specific superinterfaces. JBoss/AOP also supports the concept of introductions. New behaviour can be added to the context of an aspect by forcing it to implement an interface. A mix-in class is provided that handles the new interface and is automatically attached to the concerned classes. Dynamic AOP approaches such as JAsCo and JAC however do not provide introductions.

Sharing of context information
Once the required information for the activation of the rules is captured by aspects, it has to be made available at the events where the rules are applied. To this end, support for sharing the aspect context information is needed. JAsCo enables this by specifying aspect beans. An aspect bean is able to contain several modular "aspects" and allows sharing information between the contained aspects. This information can include both structure and behaviour. In AspectJ, AspectWerkz and JBoss/AOP, sharing information between several aspects is not so straightforward to achieve. JBoss/AOP does however allow explicit control over aspect instantiation by employing aspect factories. As such, related aspects can easily receive references to each other. In AspectWerkz and AspectJ, aspect instantiation happens implicitly by the aspect framework, and as a result, it is not possible to influence this.

Controlled instantiation and initialization
As the rules themselves are defined as reusable as possible, it is required to customise the rules towards the specific environment in which they are being applied. Most aspect-oriented technologies however do not allow sophisticated control for initializing aspects applications, as the aspect instantiation is done implicitly when the aspect is woven into the core functionality. JAsCo and JBoss/AOP are the only two reviewed approaches that allow to explicitly instantiate and initialize aspects. JAsCo employs a separate connector construct where aspects are instantiated upon a concrete context. The connector also allows customising the instantiated aspects and supports the full expressiveness of the Java language for this end. In addition, connectors are also able to specify more expressive combination strategies in order to manage the cooperation among several aspects that are applicable onto the same joinpoint. Using combination strategies it is even possible to customise the aspects on a per joinpoint basis. JBoss/AOP introduces the novel concept of aspect factories, allowing fine-grained control over aspect instantiation. Aspect customisation happens through an XML connector that describes the deployment details. This XML file allows specifying a set of properties that are passed as input for aspect initialization. The aspect itself is responsible for parsing the XML tree, which makes it somewhat more

cumbersome. In contrast to JAsCo connectors, no static type checking is possible for these XML property definitions.

Specifying precedence, combination and execution

The rule integration needs to specify the combined behaviour of several rules explicitly if they are applicable at the same joinpoints. If this is not possible, the different rules might interfere and cause incorrect results. This problem is a well-known issue in AOP, and is identified as the *feature interaction problem* [25]. Most approaches support a limited form of managing the combined aspectual behaviour by specifying the aspect sequence. Approaches such as JAC, JBoss/AOP and AspectWerkz allow specifying explicit sequences of aspect deployments by means of *stacks*. Whenever a joinpoint is encountered, the deployed aspects are executed in the order specified by the stack. JAsCo allows arranging the execution of a set of rules by explicitly specifying the desired sequence in the connector. In addition, JAsCo provides combination strategies, which allow, in comparison to mere precedence, a more fine-grained programmatic control over the combined aspectual behaviour.

Dynamic pluggability of crosscutting behaviour

Rules constantly evolve to cope with changes in the business requirements, other rules become obsolete and new ones are added. Thus, the aspects that encapsulate their links should be pluggable at run-time to reflect that volatility. Approaches like JAC and JAsCo allow adding and removing aspects at run-time in a very straightforward way. Aspects can be attached as well as removed from any joinpoint at run-time.

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

Some AOP approaches such as AspectJ and HyperJ only allow static pluggability of aspects, i.e. aspects can only be added at compile-time and it is not possible to plug them in or out at run-time.

Dynamic customization

Likewise, to adding and removing aspects, altering properties of aspects at run-time is also a desired feature when they represent volatile rule integration code. Altering properties is in most approaches as simple as invoking methods defined in the aspects. However, in order to be able to invoke methods, the aspects have to be found! In AspectJ, it is only possible to fetch an aspect by name. AspectWerkz allows fetching aspects on a per joinpoint basis, but however requires obtaining every joinpoint by

name. Fetching all aspects disregarding the concrete joinpoint they are attached to is not possible. JAsCo and JBoss/AOP do allow fetching all aspects in the system.

5 Related work

Our work is an original combination of two areas, more specifically aspect-oriented programming and separating rule-based knowledge in object-oriented software applications. We have contributed previously to this line of research. First of all, in [5] and [6] we discuss similar but more elaborate experiments with AspectJ and JAsCo, respectively. Secondly, we developed aspect-oriented techniques for encapsulating the rule integration code in the context of *hybrid systems*, which combine an object-oriented language and a full-fledged rule-based language for representing rules. These aspect-oriented techniques are based on HyperJ [9] and AspectJ [10]. In these papers we do not consider advanced aspect-oriented features as described in this paper, but we have to deal with the additional challenge of combining two languages of different programming paradigms. To our knowledge, there have been no other efforts that apply aspect-oriented programming to improve the separation of rules from object-oriented software.

Furthermore, we consider related work in the areas separately. First of all, the main body of this paper provides a thorough overview of the relevant work in the field of aspect-oriented programming. Secondly, there exist many technologies that represent rules explicitly and separately from core functionality in object-oriented software applications. We observe that they take radically different approaches:

- Rule-based knowledge can be represented separately in the object-oriented programming language itself. An extension to this approach is representing rule-based knowledge explicitly using object-oriented design patterns, such as the *Rule Object Pattern* [4], *Patterns for Personalisation* [22] and *Rule Patterns* [15].
- Other approaches focus on *externalising* explicit rules, such as *Business Rule Beans*, which store rules as XML fragments [23].
- There are dozens of both commercial and academic hybrid systems, which support explicit and separate representation of rules in a rule-based language. Due to space limitation we do not list them here. The results of a survey of hybrid systems is presented in [9]. A few examples are *OPSJ* [12], *JRules* [14], *SOUL* [18] and *Jinni* [13].

However, since none of these approaches support the encapsulation of tangled or crosscutting code, we find that they are not able to separate the rule integration code fully.

Note that we do not consider information systems, although some database management systems offer support for business rules. The reason is that they implement a *data-change-oriented* approach, activating rules when data changes. However, when rules are not bound to a particular object or data but are “free-floating”, a *service-oriented* approach is warranted [26]. Moreover, even C. J. Date

states that not all rules can be implemented in the database layer, but have to be considered in the application layer [8].

6 Conclusions

State-of-the-art business rules approaches mainly aim at physically separating the rule definitions from object-oriented applications. The integration code for a rule however, still remains tangled in the core functionality itself. In this paper, we identify a set of requirements, which we believe are essential in order to successfully encapsulate the rule integration code. In addition, we show how Aspect-Oriented Programming in particular is ideal for describing this rule integration, as AOP advocates the encapsulation of tangled code. To this end, several state-of-the-art AOP approaches are analyzed and tested towards the requirements that we identified early on.

We conclude that Aspect-Oriented Programming in general is able to satisfy all of the requirements that we identified for cleanly encapsulating the rule integration. However, none of the analyzed approaches provides full support for all of the requirements. Dynamic AOP approaches, such as AspectWerkz, JAsCo and JBoss/AOP allow dynamically adding and removing rules integration code when needed. This is in a lot of cases an essential requirement, as rules tend to change independently of the rest of the application. In addition, these approaches make use of a separate connector concept, which allows separating the identification of an event and the application of rules upon those events. As a result, rules can be instantiated explicitly, customized towards the context upon which they are being applied and their mutual interaction can be managed. Some dynamic AOP approaches however might induce a rather big performance penalty at run-time and their joinpoint model is at the moment less expressive than the ones provided by their static counterparts. Although static AOP approaches, such as AspectJ, do not allow the dynamic pluggability of rule integration code, they provide a more fine-grained description of the events upon which rules can be applied. In addition, these approaches allow introducing unanticipated data required by rules quite easily in the application at hand.

This paper contributes in identifying the requirements that are necessary for encapsulating the rule integration code and that need to be satisfied by any suitable AOP approach. At the moment, even though AOP concepts are suitable to encapsulate the business rule integration code, no ideal AOP approach is available which covers all identified requirements for the clean encapsulating of the rule integration. One needs to make a choice between the expressive joinpoint model offered by static AOP approaches and the dynamic pluggability offered by dynamic AOP approaches. However, the obstacles for achieving the contributions of both static and dynamic AOP approaches are mainly of technical nature. Dynamic approaches are introducing more and more features which are supported by their static counterparts. For example, the latest JAsCo release includes a run-time weaver that even improves AspectJ performance-wise. Hence, as AOP research is evolving, we foresee an increasing number of AOP approaches which combine those static and dynamic characteristics.

References

1. Aspect-Oriented Software Development website: http://www.aosd.net
2. AspectJ Website: http://eclipse.org/aspectj
3. AspectWerkz Website http://aspectwerkz.codehaus.org
4. Arsanjani A.: Rule object 2001: A Pattern Language for Adaptive and Scalable Business Rule Construction (2001)
5. Cibrán M. A., D'Hondt M., Jonckers V.: Aspect-Oriented Programming for Connecting Business Rules. In Proceedings BIS, Colorado Springs, USA (2003)
6. Cibrán M. A., D'Hondt M., Suvée D., Vanderperren W., Jonckers V.: JAsCo for Linking Business Rules to Object-Oriented Software. In Proceedings CSITeA, Rio de Janeiro, Brazil (2003)
7. Communications of the ACM: Aspect-Oriented Software Development. October (2001)
8. Date C.: What not How: The Business Rules Approach to Application Development. Addison-Wesley Publishing Company (2000)
9. D'Hondt M., Gybels K., Jonckers V.: Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis. In Proceedings of ACM SAC, Nicosia, Cyprus (2004)
10. D'Hondt M., Jonckers V.: Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge. In Proceedings of AOSD, Lancaster, UK (2004)
11. Fleury M., Reverbel F.: The JBoss Extensible Server. In Proceedings of Middleware, Rio de Janeiro, Brazil (2003)
12. Forgy C. L.: OPSJ 4.1 Manual. Production Systems Technologies Inc. (2001)
13. Jinni: Jinni 2004 Prolog Compiler: A High Performance Java and .NET-based Prolog for Object and Agent-Oriented Internet Programming, User Guide. BinNet Corp. (2003)
14. JRules: JRules 4.0, Technical White Paper. ILOG (2002)
15. Kappel G., Rausch-Schott S., Retschitzegger W., Sakkinen M.: From Rules to Rule Patterns. In Proceedings of Advanced Information Systems Engineering (1996)
16. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier, J., Irwin J.: Aspect-oriented programming. In Proceedings of ECOOP, Finland (1997)
17. Matthijs F., Joosen W., Vanhaute B., Robben B., Verbaeten P.: Aspects Should Not Die. Aspect-Oriented Programming Workshop (ECOOP'97), Enschede, The Netherlands (1997)
18. Mens K., Michiels I., Wuyts R.: Supporting Software Development through Declaratively Codified Programming Patterns. In Proceedings of SEKE, Buenos Aires, Argentina (2001)
19. Ossher H., Tarr P.: Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. In Communications of the ACM (2001)
20. Pawlak E., Seinturier L., Duchien L., Florin G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In Proceedings of Reflection. Kyoto, Japan (2001)
21. Ross R. G.: Principles of the Business Rule Approach. Addison-Wesley (2003)
22. Rossi G., Schwabe D., Guimaraes R.: Designing Personalized Web Applications. In World Wide Web (2001) 275–284
23. Rouvellou I., Degenaro L., Rasmus K., Ehnebuske D., McKee B.: Extending Business Objects with Business Rules. In Proceedings of TOOLS, St-Malo, France (2000)
24. Suvee D., Vanderperren W., Jonckers V.: JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development. In Proceedings of AOSD, Boston, USA (2003)
25. Tarr P., D'Hondt M., Bergmans L., Lopes C.: 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 ECOOP, Cannes, France (2000)
26. Von Halle B.: Business Rules Applied. Wiley (2001)