Visualizing Join Point Selections Using Interaction-Based vs. State-Based Notations Exemplified With Help of Business Rules

Dominik Stein, Stefan Hanenberg, Rainer Unland

University of Duisburg-Essen, Germany
Institute for Computer Science and Business Information Systems (ICB)
Schützenbahn 70, 45117 Essen
{ dominik.stein, stefan.hanenberg, rainer.unland }@icb.uni-due.de

Abstract: In Aspect-Oriented Software Development, the selection of join points is an essential part. Join point selections identify the points in a program (i.e. in its code, or during its execution) at which aspectual adaptations need to take place. In order to communicate such selections independent of the underlying aspect-oriented programming language, it is desirable to visualize join point selections in an appropriate way. In this paper we focus on the appropriateness of interaction diagram-based and state diagram-based visualizations of join point queries, exemplified with help of two business rule examples. As a result, we discover that even if join point queries are based on interactions in the base application, state diagram-based representations are needed to appropriately capture the selection semantic of that query.

1 Introduction

Aspect-Oriented Programming (AOP) [KLM+97] has begun to establish itself in industrial software development projects. Moreover, first attempts to use aspect-oriented programming to implement business rules have been conducted successfully [CDJ03]: The modularization means of aspect-orientation allow the encapsulation of individual business rules as distinct units, as well as the enforcement of such business rules under well-defined circumstances.

In particular, it is the applicability constraints of business rules that makes the use of Aspect-Oriented Software Development (AOSD) [FECA04] an appealing alternative to conventional programming techniques, such as object-orientation. Applicability constraints specify the circumstances to which a business rule applies (e.g. that a customer needs to buy at least 20 products per yearly quarter to be considered a frequent customer), as well as the situations at which the business rule should be evaluated or enforced (e.g. whenever the customer has bought a new product). Aspect-orientation provides a means, commonly referred to as *pointcut* [KLM+97], that allows the concise specification of such applicability constraints. Pointcuts (or more generally termed, join point selections) represent a key artifact in AOSD. Join point selection identify all relevant points in a program (here: points in the dynamic execution of the program) at which aspectual adaptations (here: business rules) are to be enforced.

Implementation experiences in AOSD have shown that for the sake of reusability it is beneficial to keep the pointcut specification separate from the adaptation specification (e.g. by defining an advice in an superaspect, whose (abstract) pointcut is detailed in an subaspect) [HaSc03, HaKi02]: Doing so allows easy application of existing aspects in different problem domains; query specifications can be refined individually (i.e. without considering the adaptations they are associated with) to meet new or supplementary requirements; existing query specifications can be reused to form new ones. Contemplating on these facts, we consider it indispensable to have *distinct* design *models* that help us understand and reason about the conditions and constraints under which join points should be selected – or, in other words, under which circumstances the aspectual adaptations need to take effect.

One way of visualizing join point selections are «Join Point Designation Diagrams» (or JPDDs in short, [SHU04]). JPDDs provide means to specify join point selections based on interaction diagrams (or, to be more precise, based on sequence diagrams). Such diagrams are very closely related to UML interaction diagrams; however, they have a different objective: While interaction diagrams in the UML are used to specify the behavior of a system (in terms of interacting objects), interaction diagram-based JPDDs render selection patterns of object interactions (that are meant to initiate some kind of adaptation). Interaction diagram-based JPDDs have been originally developed for aspect-oriented systems whose conceptual view on join point selections is based on object interactions. However, as it turns out, there are situations where that conceptual model does not match the intent of a join point selection specification. For example, there are situations where the conceptual model of a join point selection is state-based – rather than message-based. In such cases, it is desirable to have a new kind of JPDD that can cope with such (state-based) conceptual model of join point selection. In this paper, we develop a new kind of JPDD for representing state-based join point selections. We use it to visualize the applicability constraint[1] of a business rules. We compare that visualization to a conventional (interaction-based) one, and point out their differences as well as their benefits.

The remainder of this paper is structured as follows: At first (in section 2), we introduce the sample business rules that represent the "test case" for the remaining paper. The business rules themselves are implemented with help of AspectJ [Ladd03], which is currently the most prominent aspect-oriented programming language. Then (in section 3), we introduce the graphical concepts of «Join Point Designation Diagrams» as presented in [SHU04]. In section 4, we give graphical representations for the applicability constraints of the business rules from section 2 using the notation introduced in section 3, and we investigate its appropriateness with respect to the conceptual objectives of the business rules. After identifying various deficiencies, we present novel state diagram-based modeling means to overcome these shortcomings. Section 5 points to related work. Section 6, finally, gives a short summary of the outcomes and concludes the paper.

[1] Note that in the remainder of the paper we are going to use the (business rule-specific) term applicability constraints as a synonym for the (more general) term join point selection.

2 The Sample Business Rules

In the following, we take a closer look at the sample business rules – i.e. at the pointcuts that outline their applicability constraints – to be considered in this paper. The rules apply to the domain of an online shop.

Business rule #1: Customers who choose credit card payment have to cover an additional 2% service charge; enforce the rule when the product prices are retrieved while a customer is purchasing the items in his/her online shopping cart (and the final total amount needs to be calculated).

```
pointcut ac1(Customer c):
    if(c.cart.kindOfPayment = "CreditCard")) &&
    call(Float Product.getPrice()) &&
    cflow(execution(void Store.purchaseCart(c)) && args(c));
```

Pointcut `ac1` refers to all invocations of method `getPrice` on `Product` objects that occur in the control flow (`cflow(..)`) of method `purchaseCart`, being invoked on the `Store` object (i.e. that occur while method `purchaseCart` of the `Store` object is executing). In the pointcut, `c` refers to the `Customer` object that has been passed as an argument (`args(c)`) to the `purchaseCart` method. The `Customer` object `c` has a reference to the customer's `cart` object (it is assumed that each customer has exactly one online shopping cart). The `cart` object has an attribute `kindOfPayment` that indicates the way the customer likes to settle his/her account. The pointcut checks if the attribute `kindOfPayment` is set to "CreditCard". If this is the case, the pointcut "fires" – meaning that it executes the *advice* [KLM+97] that is affiliated with it (not shown here). That advice modifies the return value of method `getPrice` according to the business rule.

The applicability constraint of this business rule represents an example of (what we call) a control flow-based applicability constraint: It evaluates if a particular condition holds, i.e. whether the customer has chosen to pay by credit card, while some particular task is being performed, i.e. the purchase of the (contents of the) online shopping cart. We call these kinds of applicability constraint control flow-based because they relate to the execution progress of a program, e.g. they reflect on particular tasks that have been started, but have not completed yet, etc.

Business rule #2: Customers that register to the company's bonus program receive a 10% discount on their first purchase after registration; enforce the rule when the price of a product is retrieved (so that the customer considers the discounted price rather than the original price when comparing product prices at different online shops).

```
pointcut ac2_a(Customer c):
    call(void Store.register2BP(c)) && args(c);

pointcut ac2_b(Customer c):
    if(NewlyRegisteredCustomers.contains(c)) && this(c) &&
    call(Float Product.getPrice());
```

```
pointcut ac2_c(Customer c):
    execution(void Store.purchaseCart(c)) && args(c);
```

To realize this business rules, three AspectJ pointcuts are needed. Pointcut `ac2_b` actually outlines the applicability constraints of the business rule: It refers to a `Customer` object (`this(c)`) invoking method `getPrice` on a `Product` object. The customer object `c` needs to be contained in collection `NewlyRegisteredCustomers` which holds references to all newly registered customers that haven't purchased any (further) products yet (that is, since their registration to the bonus program). If this is the case, the pointcut "fires". Similar to pointcut `ac1`, pointcut `ac2_b` is associated with an advice that modifies the return value of method `getPrice` according to the business rule.

Pointcut `ac2_a` and pointcut `ac2_c` are needed to maintain the list of all newly registered customers (i.e. the collection `NewlyRegisteredCustomers`). Pointcut `ac2_a` refers to all invocations of method `register2BP` on the `Store` object. The pointcut exposes the `Customer` object `c` being passed to the method as an argument (see pointcut signature), so that the affiliated advice can append it to the collection of `NewlyRegisteredCustomers`. Pointcut `ac2_c` refers to all executions of method `purchaseCart` of the `Store` object. This time, the `Customer` object `c` being passed to the method is exposed so that the affiliated advice can remove it from the collection `NewlyRegisteredCustomers`[2].

The applicability constraint of business rule #2 exemplifies a state-based applicability constraint: It evaluates if the current customer is among the "newly registered" ones. We call these kinds of applicability constraint state-based because they relate to a particular object and attribute value setting that the system must be in. In the example, a collection object (`NewlyRegisteredCustomers`) has been used to capture the state information, and identify the state, that we are interested in. Of course, other implementations would have been possible, e.g. providing and setting a flag for each customer object, comparing date and time of the customer's subscription to the bonus program and of his/her last purchase, etc.

The aforementioned business rules are partially inspired by [CDJ03]. The way we implemented them is slightly different, though: In [CDJ03], aspects are used to merely "glue" the business rules, being encapsulated in distinct classes, to the right places in the target application. Hence, while the events at which the business rules need to be enforced (e.g. whenever the customer has bought a new product) are being identified by the pointcuts in the aspects, the conditions to which the business rules apply (e.g. that a customer needs to buy at least 20 products per quarter to be considered a frequent customer) are checked using if-statements in the business rule classes. In contrast to this, we chose to aggregate both specifications into the pointcut declarations in the aspects. To do so, we make use of AspectJ's if-pointcut designator, which has been introduced to As-

[2] It should be mentioned that the `Customer` object `c` should be removed *after* the execution of operation `purchaseCart` – so that the discount is granted even if operation `getPrice` is called during that execution, too. For reasons of simplicity, we are going to neglect this detail in the rest of the paper.

pectJ since version 1.0alpha1. As a result, the applicability constraints of business rules are neatly located in just one place rather than being spread across different software artifacts.

3 Overview to JPDDs

«Join Point Designation Diagrams» (JPDDs) [SHU04] represent a modeling means to visualize selection queries over software artifacts in general, and join point selections in aspect-orientation in particular. They provide a couple of abstractions to specify selection constraints on classes, objects, attributes, methods, relationships, messages, etc. (see Fig. 1). In particular, they allow developers to specify "incomplete", or partial, characteristics of the elements to select: For example, wildcards (`*`) can be used to require that element names need to begin with certain characters (illustrated by class name pattern "`Con*`" in Fig. 1, left part, which makes use of a wildcard to refer to all class names beginning with "`Con`"). Other wildcards (`..`) can be used to abstract over an arbitrary set of method parameters in a method signature (demonstrated by signature pattern "`run`" in Fig. 1, left part, which refers to all operations named "`run`" that have at least three parameters: the first one needs to be of type `Integer`; the last one needs to be of type `String`; in-between (no matter where exactly) there must be a parameter of type `Real`). Indirect relationship symbols may be used to require a path between elements rather than a direct link (exemplified by the relationship between class pattern "`C`" and

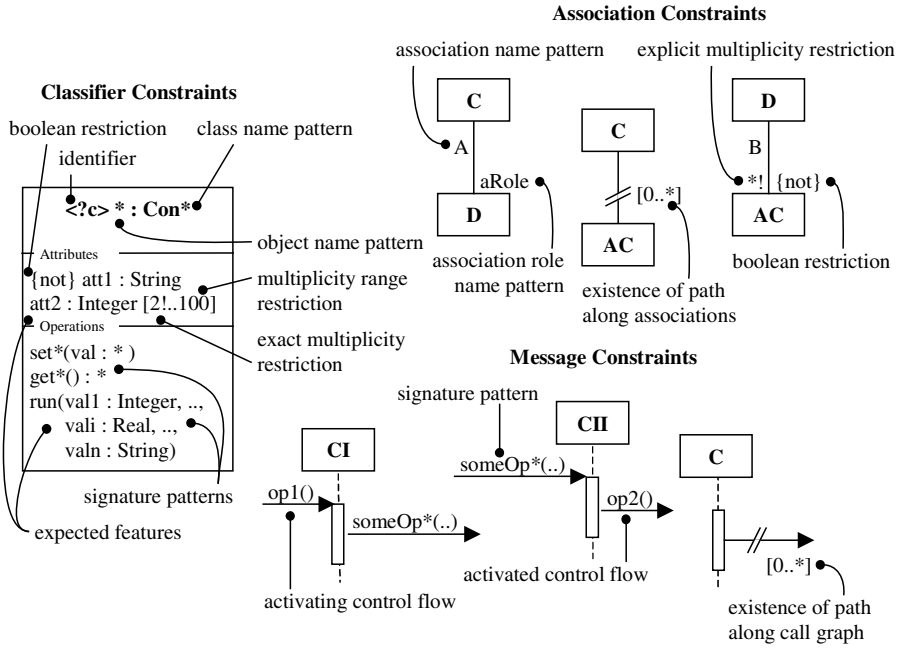

Fig. 1. Specifying selection constraints with «Join Point Designation Diagrams» (cf. [SHU04])

class pattern "AC" in Fig. 1, top right part, which denotes that a class named "AC" must be reachable from a class named "C" in order to meet the selection criteria – irrelevant of how many classes need to be passed during the traversal[3]; further examples for indirect relationships in call graphs will be investigated in the next sections); etc.

In JPDDs, elements can be given an identifier, which are prepended by a question mark (?) and entangled in angle brackets (< >) – see the class identifier <?c> in Fig. 1, left part, for example. Such identifiers can be used in general to refer to a selected element. This is particularly useful to identify those elements in the JPDD that are to be exposed by the JPDD for further processing (in which case they are listed in a parameter box in the lower right corner of a JPDD).

By default, JPDDs render minimum requirements that must be satisfied by an element in order to be selected. For example, class pattern <?c> in Fig. 1, left part, selects all classes that possess (at least one) matching attribute or operation for each attribute pattern or operation pattern, respectively, given in the class pattern. However, selection criteria may be negated, too – as done with attribute "att1" in Fig. 1, left part, which causes every class possessing such a (matching) attribute *not* to be selected. At last, multiplicities of attributes (and association ends) may be constrained either to meet concrete values, or not to exceed or under-run some maximum or minimum bound, respectively. The lower multiplicity bound of attribute "att2" Fig. 1, left part, for example, must equate "2"; its upper bound, however, may match any number up to "100".

JPDDs are based on the graphical elements of the Unified Modeling Language (UML) [OMG03], i.e. they share their symbols as well as their abstract syntax (i.e. its metamodel). However, their semantics differ considerably: In contrast to conventional UML diagrams, JPDDs are not meant for designing and constructing new software entities such as classes, associations, messages, etc. Instead, they reflect on existing entities and allow to express situations and conditions upon which actions of any kind need to be taken. By doing so, JPDDs complement the existing modeling facilities of the UML with a novel means to express application constraints *visually* – rather than textually, as can be done help of tagged values or constraints written in the Object Constraint Language (OCL) [WaKl98].

4 Representing Applicability Constraints

In the following, we present and investigate the ways in which the business rules from section 2 can be represented with help of the notational means outlined in 3.

4.1 Control Flow-Based Applicability Constraints

We start out with the control-flow based applicability constraint that business rule #1 applies to.

[3] The precise number of hops can be restricted by the multiplicity tag being attached to the relationship.

Fig. 2 outlines a graphical representation of the applicability constraints of business rule #1. In its left part, the JPDD outlines the point in (run)time to which the business rule applies. JPDD `pointcut_ac1` identifies this point with identifier `?jp` and returns it in its parameter box (see bottom right corner). `?jp` refers to a method call to an object of `Product`, invoking a method named `getPrice` that takes no parameter and returns a value of type `Float`. That method call needs to occur in the control flow (⤳) of a method call to an object of `Store`, invoking a method named `purchaseCart` that takes one argument of type `Customer`. That argument is given an identifier `?c`, which is exposed in the parameter box of the JPDD for further processing. The identifier `?c` is furthermore used in the right part of the JPDD, which outlines the object and attribute value conditions to which the business rule applies: The identifier is used to require that the `Customer` object being passed as an argument to the `purchaseCart` method call (identified by `?c`) must maintain a relationship to a `Cart` object. That `Cart` object must possess an attribute `kindOfPayment` whose value equates to "CreditCard".

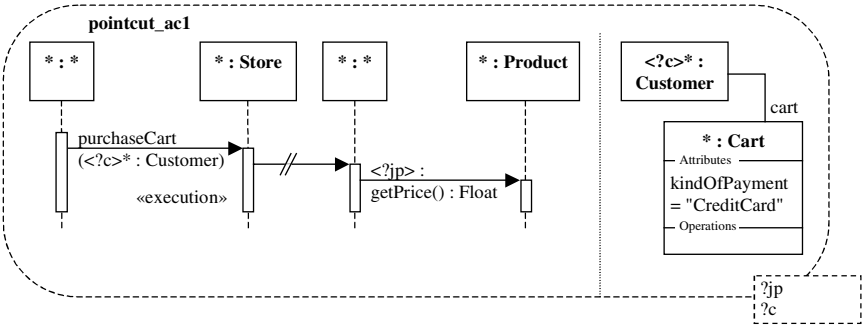

Fig. 2. Modeling the applicability constraints of business rule #1

Looking at Fig. 2 we can assess that the sequence diagram notation proves to be feasible to express control flow-based applicability constraints. The message arrows indicate in what order which tasks are processed, and which task invokes another; the activation bars of each of the participating objects signify what tasks are completed and what are still being executed. By looking at these means, it is easy to recognize that the business rule applies to all invocations of `getPrice` while `purchaseCart` is executing. Furthermore, we can tell from the object diagram being attached to the sequence diagram that the kind of payment must be chosen to be "by credit card" at this point[4]. Hence, the combination of sequence diagram-based and object diagram-based JPDDs can be considered suitable to express applicability constraints that require some condition to hold while some task is being accomplished. The sequence diagram helps the reader to perceive what (inter)actions occur, in what order, and what further (inter)actions they in-

[4] Not to mention the general abstraction mechanisms of JPDDs that permit to disregard all irrelevant information, such as what interactions occur between the invocations of `purchaseCart` and `getPrice`, or what else attributes there may be in `Cart` objects and what values they may have, etc.

duce; while the object diagram is capable to render a particular object and attribute value setting that must be met.

Although the sequence/object diagram notation is generally feasible to express control flow-based applicability constraints, there is a subtlety worth mentioning with this particular example: The condition specified in the object diagram-part of Fig. 2 renders an invariant that is supposed to hold at any point in the sequence diagram-part. This is no problem as long as the preferred kind of payment is chosen before the cart is actually purchased (i.e. before task `purchaseCart` is started). However, what if it is part of the `purchaseCart` task to let the customer choose or confirm his/her preferred kind of settlement?

In that case (i.e. if the condition may change during the execution of the task of interest), the description of the task and of the condition must be separated: Fig. 3, for example, outlines how the condition that must hold (i.e. the object diagram-part of Fig. 2) and the task that is being executed (i.e. the sequence diagram-part of Fig. 2) are segregated into two distinct JPDDs. Afterwards, they are re-joined using an annotated relationship, which indicates at what point (`@ jp`) in the sequence diagram-based JPDD the constraints of the object diagram-based JPDD should apply[5]. Hence, with help of that relationship, we require that the condition (i.e. that `kindOfPayment` must equate to "CreditCard") must be met at the method calls to operation `getPrice` only – rather than throughout the entire sequence specification[6].

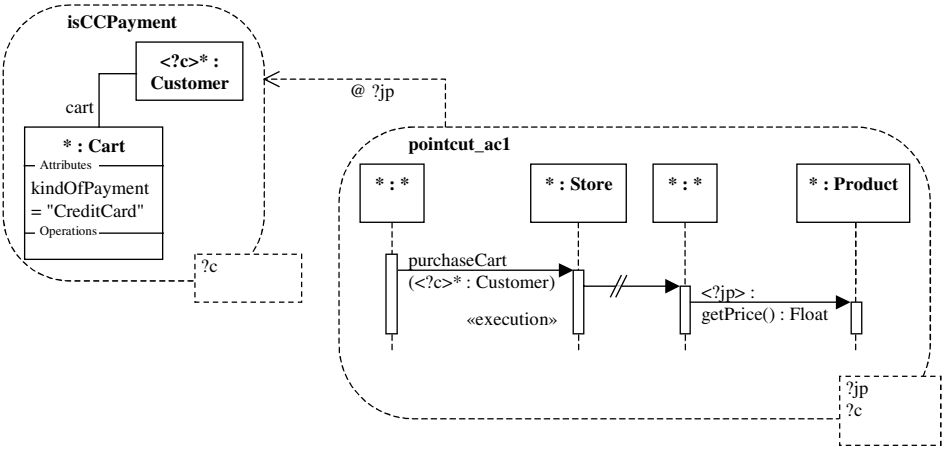

Fig. 3. Coping with possibly different object and attribute value settings over time

[5] As such, the @ annotation can be seen as an extension to the confinement relationship that has been explicated in [SHU05].

[6] It should be mentioned that this complies to the AspectJ semantics of pointcut `pointcut ac1` outlined in section 0.

The conclusion that can be drawn so far is that the sequence/object diagram notation of JPDDs can be seen capable to render control flow-based applicability constraints. Special care must be taken, if the condition that needs to hold during the execution of a task may vary during that task.

4.2 State-Based Applicability Constraints

After having investigated the graphical representation means for control flow-based applicability constraints, we are no going to take a closer look at the state-based applicability constraints of business rule #2.

4.2.1 Discussing the Inappropriateness of Sequence Diagram-Based Visualizations

Fig. 4 shows graphical representations of the pointcuts needed to enforce business rule #2. JPDD `pointcut_ac2_a` refers to all method calls to objects of `Store` that invoke operation `register2BP` taking one argument of type `Customer`. These method calls – together with the corresponding arguments passed – are exposed by the JPDD

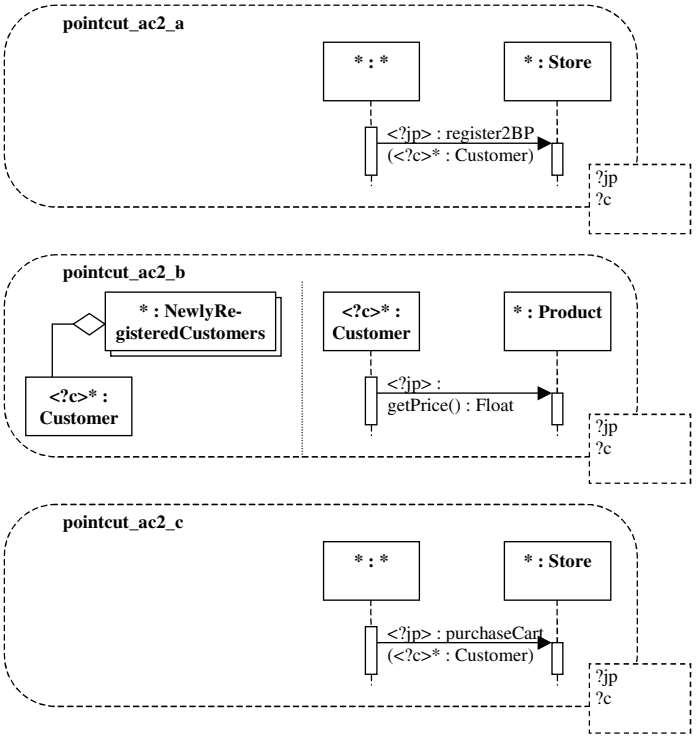

Fig. 4. Modeling the pointcuts needed to enforce business rule #2 (using separate diagrams)

using identifier `?jp` and `?c`. Similar to that, JPDD `pointcut_ac2_c` selects and exposes all method calls (`?jp`) to objects of `Store` that invoke operation `purchaseCart` taking an argument (`?c`) of type `Customer`. JPDD `pointcut_ac2_b`, finally, designates all method calls from `Customer` objects to `Product` objects that invoke method `getPrice`, taking no argument and returning a value of type `Float` (see right part of the JPDD). The method calls – together with their respective sender objects – are exposed using identifier `?jp` and `?c`. Identifier `?c` is furthermore used to require that the sender objects must be contained in the collection `NewlyRegisteredCustomers` (see left part of the JPDD).

The graphical representation of the applicability constraints of business rule #2 – or rather, of the pointcuts used to implement them – shown Fig. 4 cannot be considered satisfactory: We see three distinct diagrams, and nothing indicates that they are related to each other or that they share a common objective. At best, we can guess from JPDD `pointcut_ac2_b` that the business rules should be enforced if a customer is among the "newly registered" customers. However, we cannot tell under which circumstances such customers are to be considered "newly registered", and when they are not.

To overcome these deficiencies, the separate JPDDs from Fig. 4 are merged into a single and consolidated JPDD in Fig. 5. The JPDD selects (and exposes) all method calls (`?jp`) from `Customer` objects to `Product` objects invoking method `getPrice` (which takes no argument and return a value of type `Float`). These method calls need to occur after the invocation of operation `register2BP` on `Store` objects. However, they must *not* occur after a method call to operation `purchaseCart` of `Store` objects (i.e. there must not be such method call between the invocation of `register2BP` and the

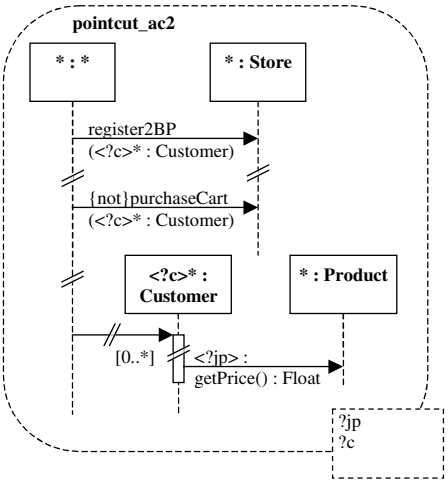

Fig. 5. Modeling the applicability constraints of business rule #2 in a consolidated diagram

method call to select (`?jp`)). Finally, the argument (`?c`) passed to both the `register2BP` method call and the `purchaseCart` method call must coincide with the sender object (`?c`) of the method call of interest (`?jp`).

Fig. 5 improves over Fig. 4 in that it emphasizes the (chronological) order of the individual method calls (`register2BP`, `purchaseCart`, and `getPrice`) so that their relation to each other is made well perceivable. Furthermore, we can tell from the diagram that the business rule is to apply (i.e. at each method call to `getPrice`) only, if a customer has registered to the bonus program (i.e. has called `register2BP`), but hasn't purchased (any items using) his/her online shopping cart yet (i.e. hasn't called `purchaseCart`). Hence, unlike to Fig. 4, we have all the information at hand that is necessary to understand the applicability constraint of the business rule.

Nevertheless, the visual representation given in Fig. 5 does not appropriately reflect on the business rule-specific interpretation of that information. That is, it is not the method calls being invoked on the `Store` objects that are of interest. Rather, it is what these method calls mean to the `Customer` object being passed as argument (i.e. that s/he is considered to be in state "newly registered", or not). In Fig. 5, however, these semantic implications of the method calls are not shown.

4.2.2 Investigating a State Diagram-Based Solution

Fig. 6 outlines the applicability constraints of business rule #2 using a state-based notation. The state diagram describes the possible states that `Customer` objects may have – seen from the perspective of the business rule. The state being of particular interest to that rule is state `?newlyRegistered`. This (business rule-specific) state is primarily characterized by its incoming and outgoing transitions: Method call `register2BP` from Fig. 5 denotes the incoming transition, while method call `purchaseCart` from Fig. 5 denotes the outgoing transition. While the `Customer` object (`?c`) is in state `?newlyRegistered`, any product price information s/he is requesting is supposed to

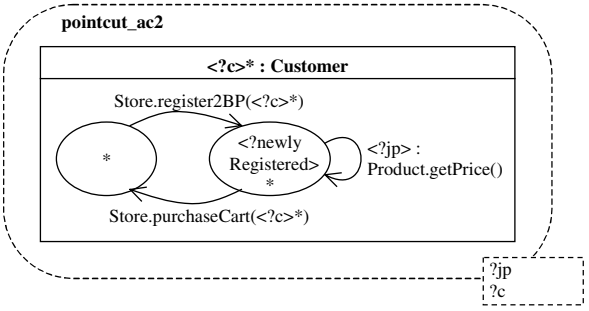

Fig. 6. Modeling the applicability constraints of business rule #2 in a state-based diagram

be selected (and exposed) by the JPDD for further processing (i.e. for the modification according to the business rule).

The state diagram-style notation used in Fig. 6, finally, draws the reader's attention to the appropriate facts. Method calls `register2BP` and `purchaseCart` are considered state transitions of a `Customer` object rather than (plain) invocations on a `Store` object. As such, their implications on the (business rule-specific) state of `Customer` objects become explicit. Likewise, the method calls at which the business rule need to be enforced (`getPrice`) are rendered with respect to the state of the `Customer` object. At last, it is easily perceivable under which conditions the `Customer` objects enter the state that the business rule applies to, and under which conditions they leave that state again.

As a conclusion, we can state that the sequence diagram-style of conventional JPDDs to render behavior is not sufficient to render state-based applicability constraints. A new – state diagram-based – notation is therefore necessary to allow developers to emphasize the relevance of state transitions and states to the applicability constraints of business rules.

5 Related Work

The work described in this paper is closely related to other aspect-oriented modeling approaches such as Theme/UML [BaCl05], AODM [SHU02], or aspect-oriented software development with use cases [JaNg05]. All of these approaches provide dedicated means to designate sets of join points to which aspectual adaptations are to be applied. In contrast to the work presented here, though, the approaches make use of textual notations only – often they simply employ AspectJ's proper pointcut language. Hence, no visualization means are provided that could help developers to understand where, when, and under what circumstances the aspectual adaptations actually take effect. JPDDs, and the extensions to them described in this paper, should be easily combined with those approaches – simply by replacing the textual declarations with their graphical counterparts in terms of JPDDs.

Furthermore, the state diagram-style extensions to JPDDs presented in this paper relate to state diagram-based aspect-oriented modeling approaches, such as [EAB02] and [MBAE04]. In contrast to JPDDs, though, such approaches focus on the representation of the overall aspect functionality. That is, they provide means to describe both the aspectual adaptations as well as the circumstances under which these adaptations need to take effect in a single diagram. No support is given to reason about the join point queries (i.e. the applicability constraints of those aspectual adaptations) in isolation.

Finally, as JPDDs have been presented in this paper to describe applicability constraints of business rules, JPDDs need to be related to other graphical means that represent constraints – such as Constraint Diagrams [Kent97], for example. Such Constraint Diagrams are based on Venn Diagrams, and are capable to visualize invariants based on objects and object states. In contrast to JPDDs, Constraint Diagrams lack graphical facilities to

reflect on control flow, or on the sequence of tasks – which is indispensable for the visualization of control flow-based applicability constraints such as those presented in section 4.1. Furthermore, they are not designed to represent queries. Consequently, they do not provide means to identify (select and expose) program elements to which the business rules apply.

6 Discussion and Conclusion

In this paper, we have investigated two variants, i.e. interaction diagram-based and state diagram-based notations, to represent join point selections. We did so by visualizing the applicability constraints of two sample business rules, which have been implemented using AspectJ pointcuts. We started out with visualizing both applicability constraints using an interaction diagram-based representation. We observed that while in the one case the interaction diagram-based representation proved to be suitable to render the essential information, in the other case it failed to capture the key objective of the applicability constraints (or join point selections) – even though both applicability constraints were based on the interaction, or communication, between entities of the base application (i.e. the system to which the business rules were applied to).

We referred to the former kind of applicability constraints (/join point selections) as control flow-based applicability constraints. Such constraints are characterized by the fact that some condition must hold while some task is being accomplished (another term for such constraints could be process- or progress-based applicability constraint). To the latter kind of applicability constraints (/join point selections), we referred to as state-based applicability constraints. These constraints (merely) required that the system is in a particular state (for the business rule to take effect).

We observe that the "state" which the latter kind of applicability constraint is referring to is business rule-specific – rather than inherent to the base application. That means that particular interactions, or messages, in the base applications may signify state transitions from the perspective of the applicability constraint and the objects it refers to (whereas this is not necessarily the case from the base application's perspective). While on programming level, an implementation quirk (such as setting object flags, or gathering the respective objects in an extra collection; see section 0) may be considered a reasonable workaround to capture the missing (business rule-specific) state information, modelers require dedicated means to render that state information appropriately for the communication with others. This paper has outlined an approach to do so.

References

[BaCl05] Baniassad, E., Clarke, S., Aspect-Oriented Analysis and Design - The Theme Approach, Addison-Wesley, 2005

[CDJ03] Cibrán, M.A., D'Hondt, M., Jonckers, V., Aspect-Oriented Programming for Connecting Business Rules, in: Witold Abramowicz, Gary Klein (eds.), Proc. of BIS 2003, Colorado Springs, USA

[EAB02] Elrad, T., Aldawud, O., Bader, A., Aspect-Oriented Modeling: Bridging the Gap Between Design and Implementation, in: Proc. of GPCE'02 (Pittsburgh, PA, October 2002), LNCS 2487, pp. 189-201

[FECA04] Filman, R., Elrad, T., Clarke, S., Aksit, M., Aspect-Oriented Software Development, Addison-Wesley, 2004

[HaSc03] Hanenberg, S., Schmidmeier, A., AspectJ Idioms for Aspect-Oriented Software Construction, in: Proc. of EuroPLoP'03, June, 25-29, 2003, Irsee, Germany, pp. 617-644

[HaKi02] Hannemann, J.; Kiczales, G.: Design pattern implementation in Java and AspectJ. Proc. of OOPSLA 2002, November 4-8, 2002, Seattle, Washington, USA. SIGPLAN Notices 37(11), ACM, S. 161-173.

[JaNg05] Jacobson, I., Ng, P.W., Aspect-Oriented Software Development with Use Cases, Addison-Wesley Longman, 2005

[Kent97] Kent, S., Constraint Diagrams: Visualizing Assertions in Object-Oriented Models, in: Proc. of OOPSLA 1997 (Atlanta, Georgia, Oct. 1997), ACM pp. 327-341

[KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwing, J.: Aspect-Oriented Programming. In Aksit, B; Matsuoka, V., (Hrsg.): Proc. of ECOOP 1997, LNCS 1241, Springer, 1997, S. 220-242.

[Ladd03] Laddad, R., Aspectj in Action: Practical Aspect-Oriented Programming, Manning Publications, Greenwich, 2003

[MBAE04] Mahoney, M., Bader, A., Aldawud, O., Elrad, T., Using Aspects to Abstract and Modularize Statecharts, in: Workshop on Aspect-Oriented Modeling, UML '04 (Lisbon, Portugal, October 2004)

[OMG03] OMG, Unified Modeling Language Specification, Version 1.5, 2003 (OMG Document formal/03-03-01)

[SHU02] Stein, D.; Hanenberg, S.; Unland, R.: A UML-based aspect-oriented design notation for AspectJ, In: Kiczales, G. (Hrsg.): Proc. of AOSD 2002, Enschede, The Netherlands, April 22-26, ACM, 2002, S. 106 - 112.

[SHU04] Stein, D., Hanenberg, St., Unland, R., Query Models, in: Proc. of UML 2004, October 2004, Lisbon, Portugal, LNCS 3273, pp. 98-112

[SHU05] Stein, D., Hanenberg, S., Unland, R., On Relationships between Query Models, In: Hartman, A., Proc. of ECMDA-FA 2005, Nuremberg, Germany, November 7-10th, 2005, LNCS, to appear.

[WaKl98] Warmer, J., Kleppe, A., The Object Constraint Language: Precise Modelling with UML, Addison-Wesley, 1998