Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design

Dominik Stein, Stefan Hanenberg, and Rainer Unland
Institute for Computer Science and Business Information Systems (ICB)
University of Duisburg-Essen, Germany
{ dominik.stein, stefan.hanenberg, rainer.unland }@icb.uni-due.de

ABSTRACT

When specifying pointcuts, i.e. join point selections, in Aspect-Oriented Software Development, developers have in different situations different conceptual models in mind. Aspect-oriented programming languages are usually capable to support only a small subset of them, but not all. In order to communicate aspect-oriented design among developers, though, it is inevitable that the underlying conceptual model used in its join point selections remains unchanged. As a solution to this dilemma, we detail three different conceptual models in this paper that are frequently used in aspect-oriented applications. These models are illustrated using sample implementations from existing literature. Then, we introduce corresponding modeling notations based on Join Point Designation Diagrams (JPDDs) which are capable to express join point selections complying to those models. Finally, we discuss the suitability of these notations to express a desired join point selection.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques. K.6.3 [**Management of Computing and Information Systems**]: Software Management – *software development*. D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation*.

Keywords

Aspect-Oriented Software Development; Aspect-Oriented Design; Query Models

1. INTRODUCTION

In Aspect-Oriented Software Development (AOSD [9]), design of join point selections is an essential task. Much research is accomplished in finding appropriate means that permit developers to specify such selections succinctly and concisely in an easy-to-understand and localized way (cf. e.g. [11, 6, 31]). An investigation of that research indicates that several different conceptual models exist which underlie join point selections. For example, sometimes developers think of join point selections as a query on object interactions, while in other cases they think of join point selections as a query on object states. The goal of the aforementioned research is to give developers appropriate abstraction means at hand that permit to reflect on these different conceptual models in the program code.

However, working with an aspect-oriented programming language that supports conceptual model A does not mean that join point selections of another conceptual model B cannot be implemented in that language. Indeed, developers are frequently forced to do so as they need to stick to a particular programming language. In consequence, join point selections usually need to be implemented by code abstractions that do not reflect on the underlying conceptual model – which makes it hard for maintainers or co-developers to grasp the actual intent of (and the actual conceptual model behind) the join point selection.

Aspect-Oriented Modeling (AOM) can help to resolve this dilemma. AOM can provide maintainers and co-developers with an abstract view of the code, helping them to grasp the inter-dependencies between multiple code abstractions implementing one join point selection as well as the general conceptual model behind it. Modeling – i.e. graphically visualizing (parts of) a complex system – has a long tradition in conventional software development. It has proven to help developers to reason about problems without having to deal with solution details. In this paper, we aim to bring forth the benefits of this way of (problem-oriented) modeling to the field AOSD.

Modeling approaches reflecting on aspect-oriented concepts are around for quite a while (see [3, 38, 16]). However, existing modeling approaches focus mainly on the way how aspects adapt the underlying application. Graphical visualization of join point selections is only rarely considered.

In this paper, we introduce novel modeling means that permit to visualize join point selections, and that help to reflect on the different underlying conceptual models. To do so, we investigate three different examples of join point selection from scientific literature, each based on a different conceptual model. We investigate the appropriateness of existing modeling means to represent each of these selections. As we identify flaws, we introduce new modeling means that overcome the identified insufficiencies. We exemplify how these new modeling means can be used to represent the join point selections under inspection, and discuss how they relate to their underlying conceptual models. Furthermore, we explicate how join point selections which are based on different conceptual models can be combined.

The remainder of this paper is structured as follows: In section 2, we outline the need for query models; that is, we elucidate the reasons why and the circumstances under which modeling of join point selections is beneficial. In section 3, we introduce «Join Point Designation Diagrams» (JPDDs [35]), an existing modeling approach to visualize join point selections based on the conceptual model of message sending. This notation is used as a starting point for our considerations in this paper. In section 3.6, we briefly outline why JPDDs (in their current state) are not heeded suitable to express each of the investigated sample join point selections. In section 4, 5, and 6, different join point selections are presented and possible ways of visualization are investigated. Each visualization is examined with respect to the selection's underlying conceptual model – that is, a control flow-oriented, a data flow-oriented, and a state-oriented conceptual model. If a mismatch between the conceptual model of the example and its visualization is detected, novel modeling means are introduced. Section 7 elucidates how the visualizations of two join point selections based on different conceptual models can be combined. Section 8 discusses related work. Section 9 concludes the paper.

2. THE NEED FOR QUERY MODELS

Join point selections are of pivotal importance to Aspect-Oriented Programming (AOP). Join point selections designate all those relevant points in a program (i.e. in its code, or during its execution) at which aspectual adaptations need to take place. Finding appropriate means to designate such sets of relevant join points (concisely and entirely in one place, so that the benefits of code reuse are at their maximum) is a highly active field of research in AOSD [11, 6, 31]. As an indication of this significant interest, different aspect-oriented systems came up with most various language constructs to specify such queries, e.g. pointcuts [21], traversal strategies [22], match or type patterns [39, 21], logic queries [11, 13, 31], applicability conditions [7], etc.

Each of these designation means comes with its own particular abstractions that permit to specify join point selections based on different selection constraints. For example, pointcuts in AspectJ [21] select method calls depending on (particular characteristics of) the control flow they occur in. Traversal strategies in Demeter/Java [22] select classes and objects based on their structural relationships. Match patterns in Hyper/J [39] designate method specifications based on their names. Logic queries in Sally [13] select classes based on particular attributes or methods that they contain. Applicability conditions in JAsCo [6] select objects depending on the state they are in, etc.

In a perfect world, we would have one "master" aspect-oriented programming language, equipped with appropriate abstraction means for all of the aforementioned join point selection constraints (maybe even more). The language would permit developers to combine those selection constraints in arbitrary ways so they can express exactly what they want to happen (i.e. what join points they want to select). Furthermore, the language would allow programmers to gather (all) relevant selection constraints in one place so that maintainers can easily determine and comprehend the intents of a selection.

In reality, though, developers often use one particular aspect-oriented programming language that does not provide (all) the appropriate abstraction means needed to capture and reflect the developers' intention. In consequence, developers are forced to come up with cumbersome workarounds – most commonly consisting of a group of abstractions that only co-jointly implement the developers' intentions. The impacts on the resulting program code are severe: Since the program's intents are disseminated across multiple abstractions, it is difficult for maintainers to understand what the programmers originally intended to do (i.e. what join points they wanted to select).

For example, a common workaround for state-based join point selections [6] in AspectJ is to use one pointcut/advice pair to monitor if the system has reached a particular state, and to use another pointcut/advice pair to actually realize the aspectual functionality that should be performed (only) in that state (we are going to further investigate such a solution later on). In order to fully understand under which circumstances the aspect affects the system, a maintainer must read and understand both pointcut/advice pairs and, moreover, must recognize their inter-dependencies.

The best solution to deal with such a problem would be to switch to a programming language that provides suitable abstraction means to capture the entire aspect applicability conditions. However, this is most oftenly prohibited by external requirements: Maybe large parts of the systems have already been implemented and their re-implementation would be too costly. Changing the used programming language might just shift the problem to another place, anyway – unless the new language is capable to express the selection criteria of *all* the join point selections which need to be implemented in the system. Even if multiple programming languages can be used in the same development project, the necessity to combine the join point designation means of two different languages may arise in order to select the right set of join points. Then again, maintainers need to explore the entire code so they won't miss an important hint to realize which join points are actually affected by the aspects.

Therefore, we advocate the need for query models, i.e. the visualizations of join point selections. Query models can help to provide developers with an abstract view on join point selections. Such visualizations are particularly useful when a join point selection needs to be implemented by means of a *group* of code abstractions. In that case, the abstract view can help maintainers to recognize the purpose of each of the code abstractions and can provide them with a broader picture on how they jointly fulfill a particular objective.

Contemplating on the pivotal importance of join point queries in aspect-oriented software development and the benefits of keeping it separate from the adaptation specification (cf. [14, 12]), we consider it advisable to have *distinct* design models that help to understand and reason about the conditions and constraints under which join points should be selected. With help of such query models developers can strictly focus on the selection constraints of their join point selections. Furthermore, having distinct query models helps developers to reflect on the dichotomy of join point selection and join point adaptation (or "quantification" and "assertion" [10]) in aspect-oriented programming. Finally, it permits to analyze the reusability of join point selections in different application contexts – independently from the adaptation that is associated with those selections.

3. JOIN POINT DESIGNATION DIAGRAMS

Interaction diagram-based[1] «Join Point Designation Diagrams» (JPDDs [35]) have been proposed as a modeling approach especially dedicated to the graphical representation join point

[1] in this chapter, whenever saying "JPDD", we refer to *interaction diagram-based JPDDs* as they have been introduced in [35]

selections. In particular, the notation provides graphical means to visualize join point queries based on the *lexical properties* [23, 26] of program elements as well as based on the *dynamic and structural context* [26, 11] they occur in. In the following, all relevant abstraction means that are necessary to understand the considerations in the remainder of this paper are introduced. A comprehensive introduction to JPDD is omitted here due to space limitations. The interested reader is pointed to [35, 36, 37] for further details.

3.1 General Syntax

JPDDs make use of (and partially adapt) the graphical symbols from UML class and object diagrams as well as from UML interaction diagrams, i.e. sequence diagrams (Unified Modeling Language [30]). These graphical elements may be arranged in JPDDs analogously to their equivalents in UML diagrams. Figure 1 shows a sample JPDD that outlines all graphical symbols relevant for this paper.

As shown in Figure 1, JPDDs are (most commonly[2]) rendered as dashed rectangles with rounded corners. They are given a name which is shown in their top left corner (`aSampleJPDD`), and they are given a parameter box at their lower right corner that lists a set of identifiers (cf. section 3.3) designating those elements that are exposed to aspectual adaptations (e.g. to some modification or enhancement realized by some introduction or advice, etc.). JPDDs may specify behavioral selection constraints and structural selection constraints. For the former, they make use of (partially adapted) interaction diagram symbols; and for the latter they make use of (partially adapted) object or class diagram symbols. A single JPDD may specify both behavioral and structural selection constraints at the same time. Consequently, symbols of either diagram type may be combined in a single JPDD, in which case they are separated by a dotted vertical line (see Figure 1 for an example).

3.2 Name/Signature Patterns And Wildcards

By default, every string in a JPDD is considered to be a name pattern for an element name. Name patterns may contain asterisk wildcards (`*`) to abstract from an arbitrary number of characters. A bare asterisk (`*`) is used as an all-quantifier, i.e. the precise name of the respective element is considered irrelevant for the selection. Method and operation signatures are constrained by signature patterns (which consist of multiple name patterns for the method/operation name, their arguments, and their argument types, etc.). Signature patterns may contain the wildcard "`..`" in their parameter list, which is used to abstract from an arbitrary number of arguments (in analogy to the semantics of "`..`" in AspectJ [21]).

Figure 1 illustrates how patterns and wildcards can be used to specify selection constraints on the lexical properties of a program element [23, 26]. The signature pattern on the right side, for example, is used to refer to all method calls (`<?jp>`) on operations whose names start with "`search`". The asterisk wildcard (`*`) is used here to denote that the operations' names may end arbitrarily. Furthermore, the selected operation calls (`<?jp>`) must take one argument of type `int`, they must return an argument of type `DiseaseType`, and they must be invoked on objects of type `DiseaseRepositoryDBMS`. In these cases, the bare asterisk (`*`) denotes that the precise names of the argument

[2] there exist other possibilities (cf. [35])

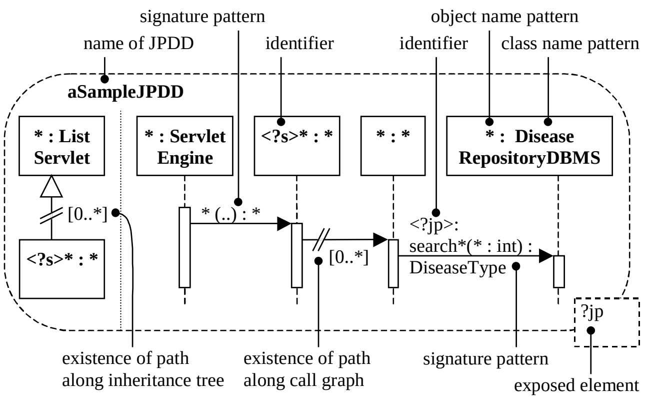


Figure 1: A Sample JPDD (cf. [35])

being passed and the object being called are considered irrelevant for the selection.

The wildcard "`..`" used in the signature pattern in the middle of Figure 1 determines that the number of arguments being passed to the operation is irrelevant for selection. The asterisks "`*`" in front of and after that parameter list pattern outline that the precise name and the return type of the operations being called are irrelevant, too.

3.3 Identifiers

Identifiers are enclosed in angled brackets (`< >`) and always begin with a question mark (`?`) so that they can be distinguished from ordinary name patterns. They are placed in front of the name or signature pattern of the element that they refer to. Identifiers may refer to any element that is rendered in a JPDD – e.g. classes, objects, messages, or stimuli[3], etc. In Figure 1, for example, identifier `<?jp>` refers to a stimulus, while identifier `<?s>` refers to an object. Identifiers are used to designate elements inside a JPDD which are to be exposed for aspectual adaptations. JPDDs provide a parameter box at their lower right corner that lists all identifiers of such exposed elements (`<?jp>` in Figure 1, for example). Apart from designating exposed elements, identifiers may also be used to interrelate elements in different sections of a JPDD. Such (interrelated) elements are deemed to be the same. For example, identifier `<?s>` in Figure 1 is used to designate an object (i.e. the same object) in the left and right part of the JPDD. The right part renders the behavioral selection constraints on that object, while the left part outlines the structural selection constraints.

3.4 Indirect Relationships

Figure 1 elucidates furthermore how selection constraints can be specified on the dynamic or structural context of join points [26, 11]. The indirect message symbol (⫽▶) in the middle of Figure 1, denotes that all selected method calls (`<?jp>`) need to occur in the control flow of another method invocation – namely in the control flow of an arbitrary method call invoked on an arbitrary instance (`<?s>`) by an (any) instance of `ServletEngine`. On the left side of Figure 1, the receiver instances (`<?s>`) of such method calls are further confined to instances of `ListServlet`, (or of one of its subtypes). This is accomplished by means of an

[3] A stimulus is a runtime event that complies to a particular message specification. In other words, a stimulus represents a *runtime instance* of a message specification.

indirect inheritance relationship (◁//–) which constraints that there must exist a path from one class to another class across the inheritance hierarchy. Both kinds of indirect relationships may be adorned with a multiplicity that indicates how many classes, objects, or method calls, respectively, may reside on the path from one element to the other.

3.5 Combination Relationships

Two JPDDs may be combined in a way that the selection constraints of one JPDD are relieved or restricted by the selection constraints of the other. This may be accomplished by means of special union (∪), confinement (∩), or exclusion (\) relationships (cf. [37]). Figure 2 illustrates how these combination relationships are established between two JPDDs: `JPDD_D` is connected to `JPDD_A` by means of a union relationship (∪) – meaning that the selection criteria of `JPDD_A` are included into `JPDD_D` as *alternative* selection criteria. Furthermore, `JPDD_D` is connected to `JPDD_B` by means of a confinement relationship (∩) – which means that the selection criteria of `JPDD_B` are included into `JPDD_D` as *additional* selection criteria. Finally, `JPDD_D` is connected to `JPDD_C` by means of an exclusion relationship (\) – denoting that the selection criteria of `JPDD_C` are included into `JPDD_D` as *exclusion* constraints. In all three cases, the ρ annotation defines how the elements of the including JPDD (`JPDD_D`) relate to the elements of the included JPDDs (`JPDD_A`, `JPDD_B`, and `JPDD_C`, respectively). This mapping is particularly useful (and necessary), if the elements of the including JPDD are named differently than the elements of the included JPDD (as with the exclusion relationship in Figure 2, for example).

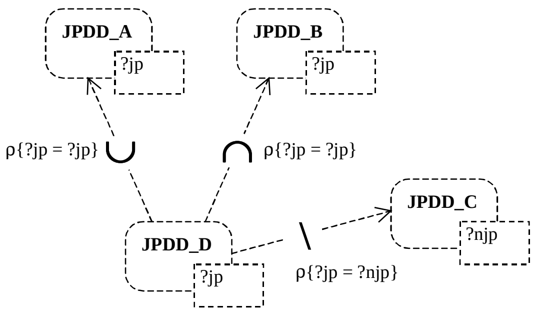


Figure 2: Combination Relationships (cf. [37])

3.6 The Problem With Current JPDDs

JPDDs have been originally developed for aspect-oriented systems whose conceptual view on join point selections is based on object interactions. Consequently, the notation presented in [35] has been based on class (and object) diagrams and interaction (i.e. sequence) diagrams. The focus of these diagrams is to render constituent object-oriented concepts, such as inheritance as well as message sending and reception. Hence, (interaction diagram-based) JPDD – as introduce in [35] – should be principally capable to express any query on object-oriented software artifacts. However (as we will show), from a conceptual modeling perspective, restricting the join point selection mainly to object-interactions is not always satisfactory. For example, if we contemplate on different system states and their transitions, or if we reason on the necessary steps to fulfill a particular task, the information which objects are involved and which messages they exchange is only of secondary interest.

In the subsequent three sections, we investigate under which circumstances the existing means (based on interaction diagrams) are sufficient to capture the conceptual idea behind a join point selection – and under which circumstances these means turn out to be insufficient. Based on this we propose new modeling means which are better suited to emphasize that conceptual view.

4. CONTROL FLOW MODEL

At first, we deal with the specification of join point selections whose conceptual model is based on control flows. We use an example taken from [27] that implements a unit test to verify if a newly registered user is actually stored into a database. To do so, the testing aspect logs if the database server really executes the corresponding request.

In [27], the testing aspect is implemented using an AspectJ pointcut that designates the following join points: It collects all executions of method `addUser` (taking two arguments of type `String`, and returning `void`) performed by instances of `DbServer`. These executions are collected only, if they occur in the control flow (`cflow`) of a method call to method `registerUser` (again taking two arguments of type `String`, and returning `void`) on instances of `AuthServer`.

```
pointcut remotePointcut():
cflow(call(void AuthServer.registerUser(String, String)))
&& execution(void DbServer.addUser(String, String));
```

The `cflow` pointcut designator expresses a chronological dependency between two given join points in which one must occur "between entry and exit" [2] of the other. With help of the means presented in section 3, this dependency is rendered as follows (see Figure 3): The JPDD outlines a method invocation from an arbitrary object of arbitrary type (`* : *`) on method `registerUser` (taking two arguments of type `String`) of an arbitrary object of type `AuthServer`. That object of type `AuthServer` hands over the control – via an arbitrary number of further objects (as indicated by the symbol –//–►, with the cardinality `[0..*]`) – to an object which invokes the method `addUser` (taking two arguments of type `String`) on an instance of `DbServer`. In its parameter box at the bottom right corner, the JPDD lists the element `<?jp>` that is returned, i.e. the execution of method `addUser` in this case.

When evaluating the appropriateness of the pointcut visualization shown in Figure 3, we need to identify the key characteristics of the `remotePointcut` join point selection described before. As such (i.e. as the key characteristic), we identify the relationship between the two method calls to `addUser` and to `registerUser`: According to the `cflow` selection criterion, the former method call is required to occur while the latter method call is still active (that is, while the latter method is still executing).

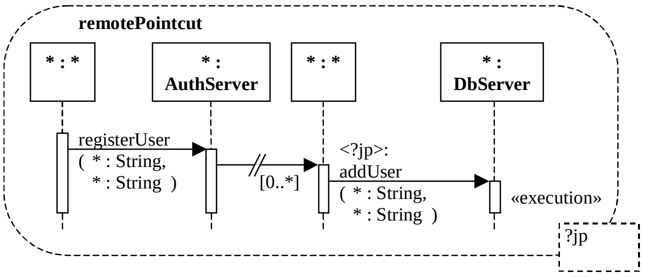

remotePointcut
* : *
* : AuthServer
* : *
* : DbServer
registerUser (* : String, * : String)
[0..*]
<?jp>: addUser (* : String, * : String)
«execution»
?jp

Figure 3: Logging the Insertion of New Users Into a Database

Looking at Figure 3, the query representation seems to emphasize that relationship appropriately: The JPDD outlines how control is passed from one instance to another by means of method invocations. It renders the chronological dependencies between method calls, and indicates how each method is invoked in the dynamic context of another. The activation bars on the lifelines of each object indicate that each method remains active until the termination of the subsequent methods.

In conclusion to these observations, we consider the interaction diagram-based modeling means of JPDDs to be suitable to represent join point selections which reflect on the (particular characteristics of a program's) control flow. Moreover, this representation seems appropriate for any join point selection which reflects on object interactions and their chronological dependencies. Hence, we deem interaction diagram-based JPDDs to be an effectual help for developers who need to understand join point selections relying on a conceptual model based on control flows – even if the implementation language in use does not provide corresponding abstractions (which has not been the case here, since AspectJ provides the `cflow` pointcut).

5. DATA FLOW MODEL

In this section, we investigate join point selections whose conceptual model focuses on data flows. To do so, we adopt an example taken from [25]. The example is part of a sanitizing aspect that quotes strings received from an untrusted party before sending them out to the HTTP stream. The goal is to avoid "cross-site scripting", i.e. the execution of malicious scripts originating from malintended thirds on the requesting machine.

In our adoption of the join point selection described in [25], we make use of three pointcuts: The first one (`clientString-Origin`) selects all calls to method `getParameter` of `Request` objects, which both take one argument of type `String` and return one value of type `String`. The second pointcut (`clientStringModification`) designates all method calls to (arbitrary) methods of objects of type `String` (or of one of its subtypes (`+`)) that return a `String` value; the number of arguments being passed is considered irrelevant (`..`). The third pointcut (`respondClientString`), finally, selects all invocation calls to print methods (`print*`) of `PrintWriter`s. These invocation calls need to be defined in class `Servlet` (or one of its subtypes (`+`)), and they need to pass one argument `o` of type `String`.

```
pointcut clientStringOrigin() :
call(String Request.getParameter(String))
after() returning (String o) : clientStringOrigin() {…}

pointcut clientStringModifications() :
call(String String+.*(..))
after() returning (String o) : clientStringModifications() {…}

pointcut respondClientString(String o) :
call(* PrintWriter.print*(String)) && args(o) &&
   within(Servlet+)
```

The three pointcuts are affiliated with three pieces of advice that jointly realize the sanitizing aspect as follows (precise implementation not shown here): The first advice (affiliated with the first pointcut) keeps track of all objects that originate from "untrusted" sources. In this case, the sanitizing aspect assumes that all data coming in via an HTTP `Request` needs to be considered "untrusted". The second advice (affiliated with the second pointcut) keeps track of all changes applied to "untrusted" objects, as well as of all objects that are newly generated from "untrusted" objects. For our investigations, we assume that any string modification and string reproduction in the base system is accomplished using the corresponding string methods provided by Java's `String` class (e.g. `concat`, `substring`, `replace`, etc.). The third advice, finally, (affiliated with the third pointcut) realizes the actual sanitization, and quotes all "untrusted" strings before they are being printed out to `PrintWriter`s – which take care of streaming out an HTTP response to the internet.

5.1 Conventional Representation

Considering this example from a modeling perspective, a one-to-one representation of the three distinct pointcuts using the means presented in section 3 is not appropriate. In particular, the representation wouldn't reveal the chronological dependencies between these pointcuts – therefore, the latter pointcut will not have much effect unless the former two have identified "untrusted" objects before, and have collected them in a list. What we would like to have is a visualization of all relevant selection criteria in *one* model.

Using JPDDs the aforementioned pointcut can be visualized as shown in Figure 4: An arbitrary object (of arbitrary type) sends a message named `getParameter` to an arbitrary object of type `Request`. It is providing an arbitrary argument of type `String`, and gets an object `<?obj>` in return. At a later point in time, control flow may be passed to the returned object `<?obj>` and a modified version of that object `<?obj>` may be returned. Finally, the control flow reaches an object of (sub)type `Servlet` which uses the previously retrieved object `<?obj>` as an argument to some invocation on operations starting with "`print`" on objects of type `PrintWriter`. Note how multiple indirection symbols are used to indicate that control flow (a) could originate either from the object requesting `<?obj>` or from any other arbitrary object (indicated by the interrupted lifeline). The control flow could (b) traverse only one or multiple objects and/or only one or multiple links before it finally reaches an instance of (sub)type `Servlet` (indicated by the interrupted message). The asterisk in front of a multiplicity tag of an interrupted message denotes that the subsequent control flow may occur once, multiple times, or

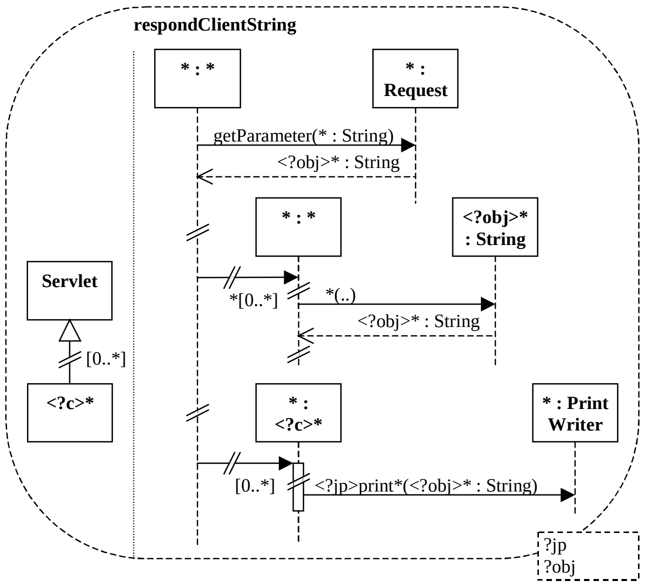


Figure 4: Intercepting the Printing of Strings from Untrusted Origin (A)

not at all. And finally, the invocation of the print operation could happen in (c) any activation of the ultimately reached instance (indicated by the interrupted activation bar). The JPDD returns the requesting object <?obj> as well as the stimulus <?jp> that caused the print method call (i.e. the join point).

Taking a critical look on the interaction diagram-based JPDD shown in Figure 4, we observe that – even though the chronological dependency between the invocation of method getParameter, the subsequent self-modification, and the ultimate invocation of a print method is well visualized – the actual selection criteria, i.e. the data flow of object <?obj>, is not sufficiently emphasized. One must carefully study the diagram to discover that the object <?obj> being returned by method getParameter and the object <?obj> being passed to the print method must be the same.

From a modeling perspective, this is not satisfactory. What we would like to have is an explicit visualization of object <?obj>, as well as of the different ways it is involved in each method. It should be easy to recognize that object <?obj> is (both) output from method getParameter and input to some print method. In interaction diagrams, however, input and output parameters are rendered as "annotations" to messages only. Hence, they are incapable to stress the fundamental significance of input and output parameters to the selection result of data flow-based queries. The use of interaction diagram-based JPDDs to represent selection constraints relying on a data flow-oriented conceptual model must therefore be considered inappropriate.

5.2 Improved Representation

In answer to our previous findings, we investigate a new modeling notation that is more suitable to express data flow-based join point selections than the interaction-based JPDDs.

Figure 5 shows a JPDD which is based on UML activity diagrams [30]. It outlines three (call) activities: getParameter (hosted by classifier Request), print* (hosted by classifier PrintWriter), and an arbitrary (*) activity (hosted by classifier String). These activities are connected by indirect transitions (of cardinality [0..*]), which means that multiple activities (i.e. none, one, or more) may take place between the former and the latter activity. Likewise, getParameter neither needs to be the first activity in the workflow; nor does print* need to be the last activity (as indicated by the indirect transition symbols from the initial state and to the final state, respectively). A self-transition from and to the arbitrary (*) activity hosted by the String classifier indicates that this activity may be executed multiple times, before the execution eventually proceeds with the print* activity. The activities are arranged in swimlanes, which indicate who is responsible for performing the respective activity: While we require the print* activity to be conducted by a Servlet instance, we make no restrictions concerning the initiations of activity getParameter and the arbitrary (*) activity. The actual data flow is represented using object flow symbols: Activity getParameter produces one object <?obj> of type String as output parameter, which eventually is passed to activity print* as an input parameter. In the meanwhile, the object <?obj> may be involved (and possibly modified) by one or more arbitrary (*) activities. Similar to the JPDD shown in Figure 4, the JPDD shown in Figure 5 returns a reference to the activity to crosscut <?jp>, as well as to the object <?obj> involved in that activity.

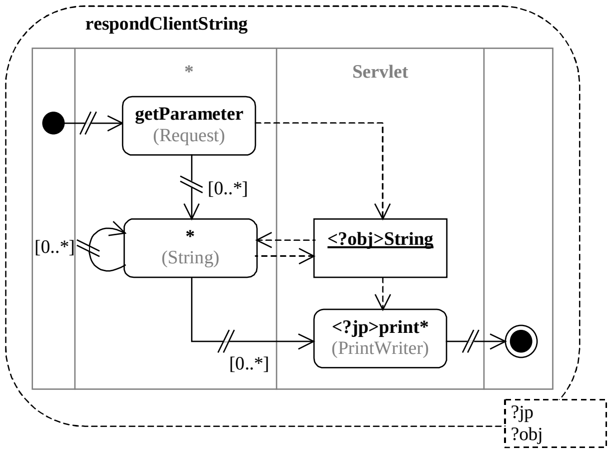


Figure 5: Intercepting the Printing of Strings from Untrusted Origin (B)

When comparing the activity diagram-based JPDD shown in Figure 5 with the interaction diagram-based JPDD shown in Figure 4, we can observe that the focus of the activity diagram-based JPDD is on workflow and data flow – rather than control flow. That means, that the activity diagram-based visualization neglects how program control is handled over from one activity to another (e.g. from getParameter to print*). Instead, it concentrates on the order of activities as well as on the data involved. The diagram emphasizes the mutual dependencies between different activities, as well as between activities and data. In consequence, the different roles of object <?obj> in either of the activities getParameter and print* is easy to conceive.

With help of activity diagram-based JPDDs, developers are thus capable to easily recognize the key selection constraints of data flow-based join point selections. The dedicated focus of activity diagram-based JPDDs on the dependencies between activities and data – such as the engagement of the same object in diverse operations, for example – makes them an appropriate means to represent join point selections which are based on data flow. Thus, they can help developers to comprehend the selection objectives of such join point selections – even if these objectives are disseminated across multiple code abstractions (as it has been the case in this example).

6. STATE MODEL

At last, we consider a persistency aspect which takes care of synchronizing a set of business objects with their database representation. The example is used to exemplify selections relying on a conceptual model of states and state transitions. It is taken from [33], and deals with the sub-task of trapping accesses to transient representations of deleted persistent objects that have not yet been collected by Java's garbage collector.

In order to realize the previously described aspect, two pointcuts are defined in [33]: The first one selects all executions of method delete, being invoked on instances of PersistentRoot (or subtypes thereof (+)). The second pointcut selects all relevant kinds of accesses to those same instances. These are, in particular, the invocation of setter and getter methods (no matter of their precise parameter list (..) and return type (*)) as well as the invocation of the object's toString method.

```
pointcut trapDeletes(PersistentRoot obj):
this(obj) &&
execution(public void PersistentRoot+.delete());
before(PersistentRoot obj) : trapDeletes(obj) {…}

pointcut detectDeletedObjects(PersistenRoot obj):
this(obj) &&
( execution(public * PersistentRoot+.get*(..)) ||
  execution(public * PersistentRoot+.set*(..)) ||
  execution(public String PersistentRoot+.toString()) );
before(PersistentRoot obj) : detectDeletedObjects(obj) {…}
```

The two pointcuts are used by two pieces of before advice which implement the interception as follows (precise implementation not shown here): The first advice hooks onto the first pointcut and marks the transient representation of the deleted persistent object. That mark is used by the second advice (which hooks onto the second pointcut) in order to check if invocations should be intercepted. With other words, the first advice is concerned with identifying all relevant objects that the second advice is supposed to affect: It is collecting all objects that are deemed to be in state "deleted". (In doing so, the first advice basically realizes some (application-specific) selection semantic – rather than adapting (or "advising") methods of the base program, which advice were originally intended for.)

6.1 Conventional Representation

As with the example presented in section 5, a one-to-one representation of the two distinct pointcuts cannot be considered appropriate. Such a representation would not reveal the chronological dependencies between these pointcuts – in that a join point designated by the first pointcut must have been selected before a join point designated by the second pointcut will be adapted. Therefore we make use of *one* model, again, which comprises all of the relevant selection constraints.

With help of the means presented in section 3, the pointcuts described above can be visualized as follows (shown in Figure 6): First, the JPDD renders a method invocation from an arbitrary object (of arbitrary type) to some method `delete` on an arbitrary object `<?obj>` of (sub)type `PersistentRoot`. After that, there are two alternative (`{xor}`) method invocations shown: One invocation to methods beginning with "`set`" or with "`get`" (taking an arbitrary number of parameters), and one invocation to method `toString` (taking no parameter at all, yet returning a value of type `String`). The methods need to be invoked on the same object `<?obj>` that already received the `delete` method (indicated by the interrupted activation lane). However, they do *not* need to be invoked by the same object that already sent the `delete` method (indicated by the interrupted lifeline). The JPDD returns the receiver object `<?obj>` as well as the stimulus `<?jp2>` that caused either of the latter (alternative) methods to execute (i.e. the join points).

Investigating the appropriateness of the JPDD shown in Figure 6, we can attest once more that an interaction diagram-based visualization of join point selections may be well-suited to point out to the chronological dependency between object interactions, i.e. the invocations of the setter and getter methods, method `toString`, and the (previously called) `delete` method. However, it fails to visualize the effects that such interactions may have on the system or object state. In this case, for example, the diagram does not outline the significant effects that the invocation of method `delete` on object `<?obj>` has on the object's future behavior (i.e. it is not supposed to answer to any more invocations).

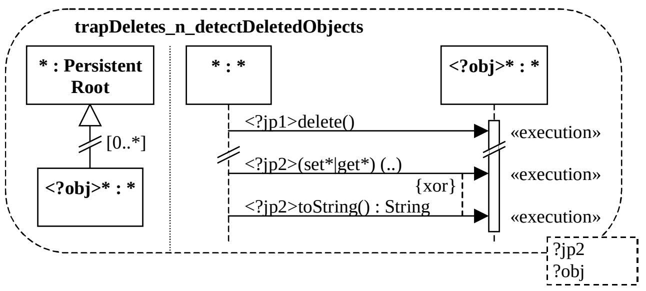


Figure 6: Trapping Attempts to Access Deleted Objects (A)

From a modeling perspective, this must be considered inappropriate since the selection criteria in the original problem was to select objects that have reached state "deleted" – rather than to select objects that have received a message invoking their method `delete`. We would like to have an explicit visualization of that state change: a visualization that emphasizes the pivotal importance of this criterion to the join point selection (and furthermore, to the aspectual adaptation that follows). Since the interaction diagram-based JPDD does not provide means to effectively indicate such a change in an object's state, it does not help developers to reflect on a join point selection that is based on a conceptual model that relies on object-states.

6.2 Improved Representation

Looking for modeling means that may overcome these problems, we identified UML state charts [30] as a promising solution. Figure 7 demonstrates how such a state chart-based JPDDs may look like for our sample join point selection: It renders an object `<?obj>` of type `PersistentRoot` (or one of its subtypes) and constraints its behavior using a state chart. That state charts consists of one arbitrary state (`*`) which has a transition to some other arbitrary state (`*`), named `<?deleted>`. The transition connecting these states is triggered by a method invocation on method `delete`. Once being in state `<?deleted>`, all method invocations to setter and getter method, as well as to method `toString`, are intercepted – and returned by the JPDD (`<?jp>`)[4]. The selected method invocations have no effect on the object's state, i.e. they are required to trigger self-transitions. Apart from the method invocations `<?jp>`, the JPDD returns the object `<?obj>` receiving those method invocations (similar to the JPDD shown in Figure 6). Note that in the considered example, state `<?deleted>` is characterized solely by the transitions it is connected to. No further restrictions are made concerning its name, or its entry, exit, and do action, etc.

Considering Figure 7, we observe that it emphasizes the relevance of state changes to the join point selection appropriately. In contrast to the interaction diagram-based JPDD shown in Figure 6, it abstracts from the general system behavior as a sequence of actions and focuses on the effects of such actions with respect to the system state. Consequently, the JPDD in Figure 7 reveals the semantics[5] of the operation `delete` as being a trigger to the state transition from any (arbitrary, or unknown) state (`*`) to state `<?deleted>`. Provided with this extra piece of information, it is

[4] From a modeling perspective, it probably would be advisable to select every (`*`) method invocation on the "deleted" object `<?obj>`.

[5] as it is seen from the perspective of the persistency aspect

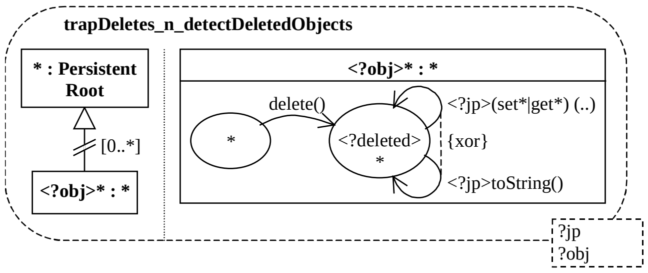


Figure 7: Trapping Attempts to Access Deleted Objects (B)

a lot easier for developers to comprehend the intention and the effects of this join point selection.

In conclusion, with help of these new state chart-based modeling means, developers are able to address the selection of events depending on the state of the system, or of a particular object, more appropriately. As such, state chart-based JPDDs can be seen as an appropriate visualization means for join point selections relying on system state, object state, and their transitions.

7. COMBINATION OF MODELS

Having discussed the various different conceptual models of join point selections and how such conceptions may be visualized appropriately, we now discuss how several conceptual models may be used jointly in one single join point selection and how that combination of conceptions can be represented visually. To do so, we revisit the example given in section 5: The sanitizing aspect considered there has to be extended in a way that it does not quote strings that already have been quoted (this example is again taken from [25] and slightly modified). Hence, we need to supplement the selection constraints concerning the data flow (that have been presented in section 5) with an additional constraint relating to the state of that data.

To implement this, we define an extra pointcut (`clientStringQuotation`) that selects all calls to a `quote` method executed on any (arbitrary) object (`*`), taking and returning a `String` value. The pointcut is affiliated with an advice that removes all "untrusted" objects from the collection of "untrusted" objects once they have been sanitized (i.e. quoted). (Recall from section 5 that the collection of "untrusted" objects is maintained by the pointcuts `clientStringOrigin` and `clientStringModification`).

```
pointcut clientStringQuotation() :
call(String *.quote(String))
after() returning (String o) : clientStringQuotation() {…}
```

Taking a closer look at the extended join point selection (implemented by the (four) pointcuts from this section and from section 5), we realize that we now need to represent selection constraints based on data flow (i.e. the engagement of "untrusted" objects in both operations `getParameter` and `print*`, for example) as well as on object state (i.e. all selected objects must be (still) "untrusted", i.e. not quoted). We are thus faced with the problem how to visualize that combination of selection constraints in a feasible manner.

As we have seen in the previous sections, selections based on data flow and based on object state are significantly different in nature. Accordingly, we are using different ways to represent them, each unveiling the individual essence of the respective kind of query: Data flow-based selection criteria are represented by using activity-diagram based JPDDs, while state-based selection criteria are represented with help of state chart-based JPDDs.

From a modeling perspective, it is now desirable to visualize the different selection constraints of the aforementioned query in their most appropriate ways – that is, using distinct diagrams (disregarding that the query is being specified by a single expression on implementation level). Doing so helps the designer to focus on the different selection constraints of the query one by one. Furthermore, it helps him/her to identify the actual intention of each selection constraints.

Representing different selection criteria of a query in different diagrams calls for a mechanism to relate those diagrams to each other. Therefore, we make use of a confinement relationship that we introduced in [37] to concatenate two JPDDs to each other. In Figure 8, we demonstrate how the different selection criteria of the query mentioned above are specified by means of distinct JPDDs, and how they are subsequently combined.

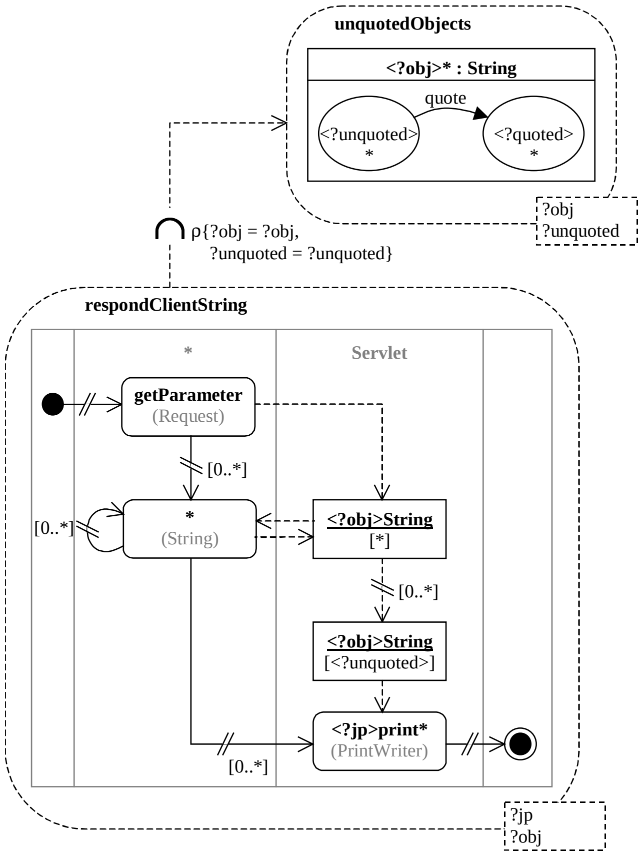


Figure 8: Avoiding Double-Quoting

The bottom JPDD in Figure 8 is quite similar to the JPDD shown in section 5. However, supplementary assertions are made concerning the state of object `<?obj>` (a) after being returned by activity `getParameter` and (b) before being passed to activity `print*`: In the former case (a), the state is considered irrelevant (`[*]`; could be omitted). In the latter case (b), the state is required to be `<?unquoted>`. Both object states are connected by an interrupted data flow symbol, emphasizing[6] that the object state

[6] A non-interrupted data flow symbol (used in a JPDD) would have the same meaning, yet is less illustrative.

could have changed arbitrary times in arbitrary manners. As we can see, `<?unquoted>` is an identifier which is further specified in the upper JPDD in Figure 8. That JPDD outlines a state chart for objects `<?obj>` of type `String`. According to that state chart, `<?unquoted>` refers to an arbitrary state (`*`) that has a state transition, triggered by an invocation of method `quote`, to some other arbitrary state (`*`), named `<?quoted>`. The JPDD exposes the object `<?obj>` as well as its state `<?unquoted>` in its parameter box. These exposed elements are related to the elements in the lower JPDD of Figure 8 by means of a confinement relationship. The mapping of the confinement is specified in the ρ annotation of the relationship.

It should be noted that it may be sufficient to look at the including JPDD only. In Figure 8, for example, it can be well observed from the bottom JPDD that object `<?obj>` must be in some state `<?unquoted>` in order to comply to the selection criteria. At this point, it may be neglected what the characteristics of state `<?unquoted>` are exactly. Only if this becomes essential for one or the other reason, we need to have a look at the details in the upper JPDD in Figure 8.

All of the previously presented JPDD diagram types may be combined similar to the mechanism shown in Figure 8. That way, combined selection may be represented in their most appropriate way – even if they are based on different conceptual models.

8. RELATED WORK

8.1 Visualizing Queries

The idea of visualizing queries is present in computer science for quite a while. The visualization means for join point selections presented in this paper relate to these existing means in various ways. For example, the principle idea of JPDDs relates to the idea of Query-By-Example (QBE) [40] as it is known from the database domain. In QBE, developers specify a query by defining sample entities which are then compared to existing entities in a database in order to find matches. Operators may be used to define permissible degrees of deviation in which selected entities may vary from those sample entities in order to be still selected. In JPDDs, too, sample patterns are defined that render the particular characteristics of those elements that should be selected. Possible deviations may be specified with help of operator symbols (such as wildcards or double-crossed lines/arrows, etc.).

Model-Driven Development (MDD), such as the Model-Driven Architecture (MDA [28]) of the Object Management Group (OMG), is another computer science domain that deals with the visualization of queries: In MDD, model transformations involve a query specification that designates all those elements in a model which have to be adapted. In currently proposed transformation techniques for MDD (such as MOLA [17] or the QVT-Merge submission [32]), query specifications are tightly coupled to their corresponding adaptations and cannot be considered in isolation (although the OMG explicitly calls for dedicated query support "to select and filter elements from models" in its "MOF 2.0 Query / Views / Transformation (QVT)" Request For Proposal (RFP) [29]). JPDDs improve this situation and consider model queries as first-class entities (i.e. entities that can exist on their own). Consequently, they permit to reuse existing query specifications in different application areas and permit to refine them in order to cope with new requirements or incremental software evolution.

8.2 Visualizing Join Point Selections

Most aspect-oriented modeling approaches are capable to express join point selections in one or the other way. In Theme/UML [3], for example, join point selections are represented by means of binding relationships. Such binding relationships are tagged with an annotation that minutely designates all join points (to which an aspectual adaptations should be applied) either by name, by enumeration, or by meta-query (see Figure 9; adopted from [5]). In contrast to JPDDs, join point designation is accomplished in a pure textual manner. As a result, the (structural) relationships between the individual join points being selected (i.e. classes and its methods) are not visualized. Moreover, no means are provided to express join point selection criteria with respect to the (earlier) behavior of a system. In consequence, neither of the previously described conceptual models of join point selection can be visualized in Theme/UML. Join point selections in Theme/UML can only refer to static properties of a system.

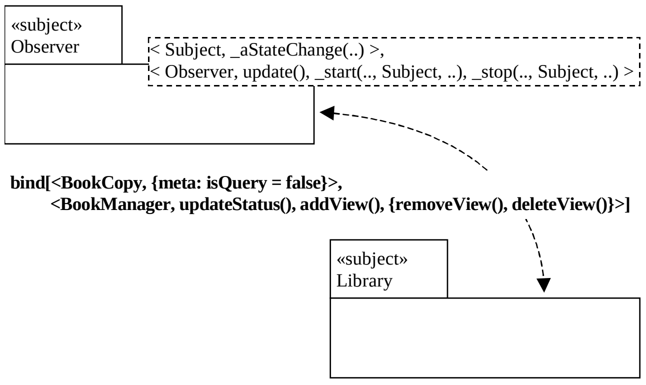
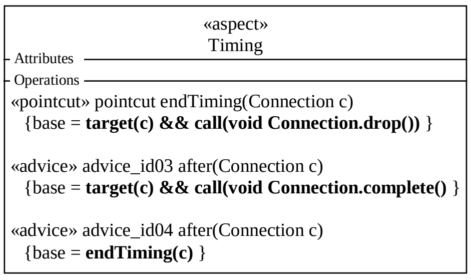

«subject»
Observer
< Subject, _aStateChange(..) >,
< Observer, update(), _start(.., Subject, ..), _stop(.., Subject, ..) >
bind[<BookCopy, {meta: isQuery = false}>,
<BookManager, updateStatus(), addView(), {removeView(), deleteView()}>]
«subject»
Library

Figure 9: Representing Join Point Selections in Theme/UML

The Aspect-Oriented Design Model (AODM) [38] makes use of AspectJ's core pointcut language to represent join point selections (see Figure 10). In doing so, the approach suffers from the same problems as the programming language when it comes to implementing join point selections of unsupported conceptual models. Moreover, by not providing a graphical representation, the approach does not facilitate or improve the identification of the (supported) conceptual models of a join point selection. Other solutions following the same idea, i.e. adopting the pointcut language of an existing aspect-oriented programming language (e.g. [16], [18], etc.), suffer from similar problems.



```
«pointcut» pointcut endTiming(Connection c)
  {base = target(c) && call(void Connection.drop()) }

«advice» advice_id03 after(Connection c)
  {base = target(c) && call(void Connection.complete() }

«advice» advice_id04 after(Connection c)
  {base = endTiming(c) }
```

Figure 10: Representing Join Point Selections in AODM [38]

Concern models in the superimposition approach [19] visualize both the join point selection as well as the join point adaptation in the same diagram. To distinguish between one and the other, join point selections are gray-shaded, while join point adaptations are solid black (see Figure 11). The approach allows in principle to use any type of UML diagram to represent a join point selection, and thus can reflect on each of the previously described conceptual models. However, the problem of this approach is that join point selections are being intermingled with their affiliated adaptations. Hence, join point selections cannot be reused in or refined for another application context, and developers cannot design their join point selections in isolation from the adaptation.

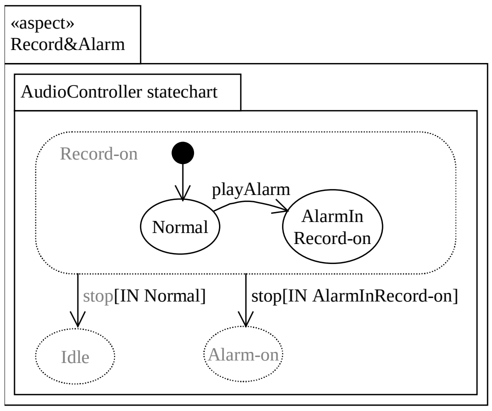


Figure 11: Representing Join Point Selections in Concern Diagrams [19]

With help of the Aspect-Oriented Statechart Framework (AOSF) [24], two (or more) independent state charts can be woven into one composite state chart, where each of the original state charts resides in its own orthogonal region. The crosscutting execution of the composite state chart is then realized by event reinterpretation, meaning that an event in one of the orthogonal regions is reinterpreted as an event with a different meaning in another orthogonal region. This event reinterpretation is visualized as shown in Figure 12. The bold lines and ellipses designate and relate the join points, i.e. the reinterpreted events, in each state chart, and thus can be seen as a join point selection.

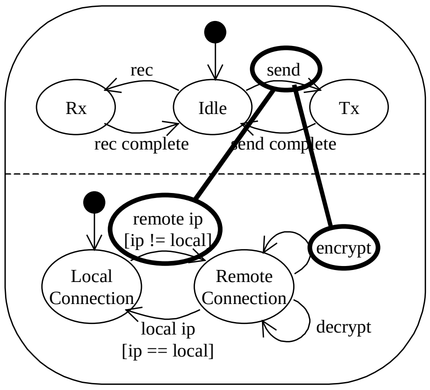


Figure 12: Representing Join Point Selections in AOSF [24]

The problem with visualizing join point selections by relating the join points in the base artifacts directly, is that they need to be specified ad-hoc and for a particular application only. No abstraction means are provided that allow to use an existing join point selection in another context. As with the previous approach, this does not promote the reuse and refinement of join point selections in another problem domain.

8.3 Join Point Selections In AOP Languages

The different conceptual models discussed in the previous chapters are realized in various different aspect-oriented programming languages. In the following, we give prominent examples of such programming languages for each conceptual model. As stated before, using a particular programming language realizing a particular conceptual model does not hinder application developers from specifying join point selections being based on another conceptual model. To demonstrate this fact, we give examples of such mismatches from scientific literature.

8.3.1 Control Flow Model

The pointcut mechanism in AspectJ allows to select distinct points in the dynamic execution of a program – which are interpreted as points in the dynamic call graph [20, 15]. Therefore, AspectJ can be considered to realize a control flow-oriented conceptual model of join point selection (and it provides dedicated selection means to reflect on this conceptual model, such as the `cflow` pointcut designator, for example). Similarly, *tracematches* [1] follow a control flow-oriented conceptual model: They permit to specify a sequence of object (inter)actions that must come to pass in a particular (partial) order during the dynamic execution of a program. Composition Filters [4] are another approach that implements a (somewhat restricted) control flow-oriented conceptual model: Its join point selections reflect on messages being sent to or by a given object.

8.3.2 Data Flow Model

The data flow-oriented conceptual model of join point selection has been recognized in [25]. That work extends AspectJ with a new `dflow` pointcut designator that allows developers to reflect on that conceptual model in a succinct and concise manner. The following pointcut demonstrates how that pointcut designator is used to express a similar join point selection as the one presented in section 5. As can be seen, the three pointcuts from section 5 are condensed to just one single pointcut, which – in particular – abstracts from the modifications that may be applied to "untrusted" objects before they are printed to the HTTP output stream. With help of the new pointcut designator, developers are able to express their actual join point selection objectives directly in the code.

```
pointcut respondClientString(String o) :
call(* PrintWriter.print*(String)) && args(o) &&
   within(Servlet+)
&& dflow[o,i]( call(String Request.getParameter(String))
   && returns(i) );
```

8.3.3 State Model

The need and presence of a state-based conceptual model of join point selections has been identified in [7], which introduces a formal approach to reflect on such state-based join point selections in aspect-oriented programming. Based on these findings, a Java-based implementation approach to state-based join point selections has been presented in [6]. The following code fragment demonstrates (by revisiting the example from section 6) how that implementation approach enables developers to define distinct transitions (`trapDeletes`, and `invalidTrans`) that must occur in a particular order (1. `trapDeletes` > 2. `invalidTrans`). As can be seen, with help of these means developers can express the relevance of (a sequence of) state changes to a join point selection directly in the code.

```
//defining a state/transition-based join point selection
hook DetectDeletedObjects {
  DetectDeletedObjects(delete(..args), anything(..args)){
    trapDeletes: execute(delete) > invalidTrans;
    invalidTrans: execute(anything) > invalidTrans;
  }
  after invalidTrans() {...}
} [...]

//binding join point selection to concrete method calls
DetectDeletedObjects ddo = new DetectDeletedObjects(
  void PersistentRoot+.delete(),
  { * PersistentRoot+.get*(*),
    * PersistentRoot+.set*(*),
    String PersistentRoot+.toString() }
) [...]
```

8.3.4 Mismatches in Design and Implementation

Implementing join point selections in one of the programming languages mentioned above does not necessarily mean that these join point selections make use of the same conceptual model as the underlying programming languages. There are many examples of join point selections whose conceptual models mismatch the ones supported by their implementation languages. In [34], for instance, an updating aspect is specified which is supposed to update objects to a database only if they have been modified (i.e. made "dirty"). The corresponding join point selection is implemented by using a method call pointcut designator in AspectJ – although the key selection criteria addresses a change in object state.

Another interesting example is given in [25], which introduces a `bypassing` supplement to the `dflow` pointcut designator discussed previously. With help of that supplement, developers are able to exclude particular data flows from the join point selection. In [25], the `bypassing` clause is used to avoid that "untrusted" objects which already have been quoted are (double-) quoted a second time. However, as we have seen in section 7, the conceptual model of that selection constraint is state-based. Hence, in this case, the data flow-oriented extension of AspectJ is used to realize a state-based join point selection.

The last example is taken from the motivating examples of tracematches [1]. Tracematches permit to bind variables across multiple object interactions, which makes it easy to use them to reflect on object state transitions (rather than message sendings). In [1], for example, tracematches are used to monitor the state of a database connection ("open", "closed") whenever a database query is issued (via that connection). Thus, in this case, the control flow-oriented join point selection means of tracematches is exploited to realize a state-dependent join point query.

9. CONCLUSION AND DISCUSSION

In this paper, we discussed the relevance and the impact of different conceptual models on the design of join point selections. We pointed out that different aspect-oriented programming languages support different conceptual models of join point selections, and consequently provide appropriate means to select join points in correspondence to those conceptual models. However, as soon as developers wish to specify join point selections relying on a conceptual model other than the supported one, they usually need to implement cumbersome workarounds which do no longer reflect on the conceptual model underlying the corresponding join point selection.

In this paper, we proposed a modeling approach to overcome this dilemma. We identified three different conceptual models for join point selections – namely control flow-based, data flow-based, and state-based join point selections. We presented three idiomatic implementations for each kind of join point selection in AspectJ, the presently most popular productive aspect-oriented programming language. We used an existing modeling means (i.e. interaction diagram-based JPDDs [35]) to visualize those join point selections, and investigated its appropriateness to capture the key selection criteria of each join point selection. While we identified interaction diagram-based JPDDs to render that key selection criteria for control flow-based join point selections appropriately, we identified them as being insufficient to visualize the key elements for data flow- and state-based join point selections. Based on our findings, we introduced activity diagram-based JPDDs to represent data flow-based join point selections and state chart-based JPDDs to represent state-based join point selections. Our subsequent evaluations have shown that these means are suitable to capture the key characteristics of the respective join point selections.

With help of the notation described in this paper, developers are able to reflect on the conceptual models of their join point selections – even if their programming language happens not to support it. This means on one hand developers may design join point selections independently of their (any) particular programming language. On the other hand, it helps developers to comprehend the objectives of existing join point selections – especially when there is a conceptual mismatch between a join point selection and its implementation.

The conceptual models of join point selection discussed in this paper have already been recognized by other researchers in the field of AOSD. As a solution, these researchers have developed new programming languages, or extended existing ones, so that developers may specify different join point selection following different conceptual models in a succinct and concise manner in the code. In contrast to that, our approach presented in this paper is to use aspect-oriented modeling means to help developers to understand and to communicate a particular join point selection and its underlying conceptual model – even, and in particular, if this join point selection is written in a programming language that does not support the respective conceptual model. In consequence, we relieve the developers from being forced to change their productive programming languages just to enable maintainers to identify the precise intention of their join point selections.

The notations we used to develop our new visualization means have been around in conventional software development for quite a while, and have been used there to reflect on the different conceptual models being discussed here. However, those notations did not address the design (and implementation) of join point selections as it is used in AOSD. In this paper, we therefore extended those modeling means so that they can deal with such selections. In doing so, we improve the current situation in aspect-oriented modeling as there is no or only limited modeling support given for the design of join point selections.

Furthermore, we recognize in this paper that none of the conceptual models described here comprises another. Hence, we expect developers to call for suitable selection means reflecting on all three of them – even in the same development project. The precise conceptual model that underlies a joint point selection depends ultimately on the application context or problem domain. The modeling notation presented in this paper helps developers to reflect on that conceptual model in their join point selections and permits to communicate join point selections among developers – independent of the implementation techniques being used.

ACKNOWLEDGEMENTS

We thank the participants and the reviewers of the European Interactive Workshop on Aspects in Software (EIWAS) 2005 [8] for their valuable remarks that have been a great help for writing this paper. We also thank the anonymous reviewers for their useful comments and suggestions.

REFERENCES

[1] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J., *Adding Trace Matching with Free Variables to AspectJ*, in: Proc. of OOPSLA'05, San Diego, CA, October 2005, ACM, pp. 345-364

[2] AspectJ Team, *The AspectJ Programming Guide*, v.1.2.1, http://eclipse.org/aspectj/

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

[4] Bergmans, L., *The Composition Filters Object Model*, Dept. of Computer Science, University of Twente, 1994

[5] Clarke, S., Walker, R.J. *Composition Patterns: An Approach to Designing Reusable Aspects*. in Proc. of ICSE '01, Toronto, Canada, May 2001, ACM, pp. 5-14

[6] De Fraine, B., Vanderperren, W., Suvée, D., Brichau, J., *Jumping Aspects Revisited*, DAW Workshop, at: AOSD 2005, Chicago, IL, March 2005

[7] Douence, R., Fradet, P., Südholt, M., *Composition, Reuse and Interaction Analysis of Stateful Aspects*, in: Proc. of AOSD 2004, Lancaster, UK, March 2004, ACM, pp. 141-150

[8] European Interactive Workshop on Aspects in Software (EIWAS), Brussels, Belgium, September 2005, http://prog.vub.ac.be/events/eiwas2005/

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

[10] Filman, R., Friedman, D., *Aspect-Oriented Programming is Quantification and Obliviousness*, in: [9], pp. 21-35

[11] Gybels, K., Brichau, J., *Arranging language features for more robust pattern-based crosscuts*, in: Proc. of AOSD 2003, Boston, MA, March 2003, ACM, pp. 60-69

[12] 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

[13] Hanenberg, S., Unland, R., *Parametric Introductions*, in: Proc. of AOSD 2003, Boston, MA, March 2003, ACM, pp. 80-89

[14] Hannemann, J., Kiczales, G., *Design Pattern Implementation in Java and AspectJ*, in: Proc. of OOPSLA'02, November 2002, Seattle, WA, ACM SIGPLAN Notices 37(11), pp. 161-173

[15] Hilsdale, E., Hugunin, J., *Advice Weaving in AspectJ*, in: Proc. of AOSD 2004, Lancaster, UK, March 2004, ACM, pp. 26-35

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

[17] Kalnins, A., Barzdins, J., Celms, E., *Model Transformation Language MOLA*, in: Proc. of MDA-FA '04, Linköping, Sweden, June 2004, Springer, LNCS 3599, pp. 62-76

[18] Kandé, M.M., PhD Thesis, EPFL, Lausanne, Swiss, 2003

[19] Katara, M., Katz, S., *Architectural Views of Aspects*, in: Proc. of AOSD 2003, Boston, MA, March 2003, ACM, pp. 1-10

[20] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W., *An Overview to AspectJ*, in: Proc. of ECOOP '01, Budapest, Hungary, June 2001, LNCS 2072, pp. 327-353

[21] Laddad, R., *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, Greenwich, 2003

[22] Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996

[23] Lieberherr, K., Lorenz, D., Mezini, M., *Programming with Aspectual Components*, TR NU-CCS-99-01, Northeastern University, 1999

[24] 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

[25] Masuhara, H., Kawauchi, K., *Dataflow Pointcut in Aspect-Oriented Programming*, in Proc. of APLAS '03, Beijing, China, November 2003, Springer, LNCS 2895, pp.105-121

[26] Masuhara, H., Kiczales, G., Dutchyn, Chr., *A Compilation and Optimization Model for Aspect-Oriented Programs*, in: Proc. of CC 2003, Warsaw, Poland, Apr. 2003, LNCS 2622, pp. 46-60

[27] Nishizawa, M., Chiba, S., Tatsubori, M., *Remote Pointcut – A Language Construct for Distributed AOP*, in: Proc. of AOSD 2004, Lancaster, UK, March 2004, ACM, pp. 7-15

[28] OMG, *MDA Guide Version 1.0*, 2003 (OMG Document omg/2003-05-01)

[29] OMG, *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, 2002 (OMG Document ad/2002-04-10)

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

[31] Ostermann, K., Mezini, M., Bockisch, Chr., *Expressive Pointcuts for Increased Modularity*, in: Proc. of ECOOP'05, Glasgow, UK, July 2005, ACM

[32] QVT-Merge Group, *Revised submission for MOF 2.0 Query / Views / Transformations RFP*, 2. March 2005 (OMG Document ad/2005-03-02)

[33] Rashid, A., Chitchyan, R., *Persistence as an Aspect*, in: Proc. of AOSD 2003, Boston, MA, March 2003, ACM, pp. 120-129

[34] Soares, S., Laureano, E., Borba, P., *Implementing Distribution and Persistence Aspects with AspectJ*, in: Proc. of OOPSLA '02 (Seattle, WA, Nov. 2002), ACM, pp. 174-190

[35] Stein, D., Hanenberg, S., Unland, R., *Query Models*, in Proc. of UML '04, Lisbon, Portugal, October 2004, Springer, LNCS 3273, pp. 98-112

[36] Stein, D., Hanenberg, S., Unland, R., *A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models*, in: Proc. of MDA-FA '04, Linköping, Sweden, June 2004, Springer, LNCS 3599, pp. 77-92

[37] Stein, D., Hanenberg, S., Unland, R., *On Relationships between Query Models*, in: Proc. of ECMDA-FA 2005, Nuremberg, Germany, November 2005, Springer, LNCS 3748, pp. 254-268

[38] Stein, D., Hanenberg, St., Unland, R., *A UML-based Aspect-Oriented Design Notation For AspectJ*, Proc. of AOSD '02; Enschede, Netherlands, April 2002, ACM, pp. 106-112

[39] Tarr, P., Ossher, H., *Hyper/J User and Installation Manual*, IBM Corp., 2000

[40] Zloof, M., *Query-by-Example: A Data Base Language*, IBM Systems Journal, Vol. 16(4), 1977, pp. 324-343