Visualizing Join Point Selections for Stateful Aspects

Dominik Stein, Stefan Hanenberg, and Rainer Unland
Institute for Computer Science and Business Information Systems (ICB)
University of Duisburg-Essen, Germany
{dstein | shanenbe | unlandR}@cs.uni-essen.de

ABSTRACT

In this position paper we propose an approach to graphically represent applicability conditions for stateful aspects. We briefly explicate why we consider the visualization of such application constraints in model artifacts as beneficial for AOSD. Then, we develop graphical means to visualize these constraints following a reverse engineering approach with help of two examples.

1. NEED FOR QUERY MODELS

Queries on join points (i.e. join point selections) are an essential part of AOSD. Join point queries are necessary to identify all 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 is a highly active field of research in AOSD [5, 6, 2, 12]. As an indication of this significant interest, different aspect-oriented systems came up with most various means to specify such queries, e.g. pointcuts [10], traversal strategies [11], type patterns [10], logic queries [5, 8, 12], applicability conditions [3], etc.

Implementation experiences in AOSD have shown that for the sake of reusability it is beneficial to keep the query specification separate from the adaptation specification (e.g. by defining an advice in an superaspect, whose (abstract) pointcut is detailed in an subaspect) [9, 7]: 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 the pivotal importance of join point queries in aspect-oriented software development and the benefits of keeping it separate from the adaptation specification, we consider it advisable to have *distinct* design *models* that help us understand and reason about the conditions and constraints under which join points should be selected.

Such design models seem particularly useful when dealing with *runtime queries*, i.e. with the selection of dynamic join points based on runtime information. Examples of such queries are the selection of objects depending on their state, or the selection of messages depending on (certain characteristics of) the control flow they occur in. In these cases, join points are usually not selected based on singular and/or instant facts, but also based on incidents or circumstances that occurred earlier. Graphical representations have the potential to explicate these facts – and their chronological dependencies – better than textual notations, helping us to grasp the whole picture all at once, rather than putting together the relevant information code line by code line.

Join Point Designation Diagrams (JPDDs) [13] have been introduced as a novel modeling means to represent join point quries (a) graphically and (b) separately from the adaptation specification in (c) distinct model artifacts. JPDDs provide abstractions to specify queries on classes, their features and relationships, as well as on messages in a program's control flow. They lack dedicated support for the visualization of applicability conditions for stateful aspects, currently, though. This is the kind of runtime join point query that shall be considered in the following.

2. STATEFUL ASPECTS

Stateful aspects are recently gaining much attention [3, 4]. Stateful aspects are special (compared to conventional aspects) in that they take effect only after (if) the system has reached (is in) a particular state rather than affecting to just any (designated) event right from the start (until program termination).

Using conventional aspect-oriented approaches, stateful aspects need to be implemented in two steps: at first, a flag is set once the respective state is reached; then, once the flag is set, the crosscutting modifications (e.g. the crosscutting behavior) is put to action.

[2], for example, present a sample implementation of a simple publish-subscriber-protocol in JAsCo [14]. They define two hooks Subscribe and Publish (see following snippets 1 and 3), each of which is bound to particular operations by means of connectors (see snippet 2 and 4). The first hook is responsible for identifying the point in time beginning from which publications should occur. The second hook nominates those points that will actually cause the publication to occur (once the first hook has been passed).

```
1. hook Subscribe {
     Subscribe(subscribe(..args))
        { execute(subscribe); }         [...] }

2. SubscribeManager.Subscribe subscribe =
     new SubscribeManager.Subscribe(
        boolean PSComponent.subscribe());

3. hook Publish {
     Publish(topublish(..args))
        { execute(topublish); }         [...] }

4. PublishManager.Publish publish =
     new PublishManager.Publish(
        void ComponentX.update*(*));
```

[1] present another example that prevents the disposal of dirty documents, i.e. modified documents that haven't been saved to disk yet. The example makes use of three pointcuts (see snippets 5, 6 and 7): The first one designates those points causing the "dirty" flag to be set; the second one designates those points causing the "dirty" flag to be unset; and the third one refers to all points that ought to be intercepted (in case the "dirty" flag is set).

```
5. pointcut makeDocumentDirty():
     call(void Editor.edit());

6. pointcut makeDocumentClean():
     call(void Editor.save()) ||
      call(void Editor.create());

7. pointcut disposeDocument():
      ( call(void Editor.quit()) ||
      call(void Editor.create()) ) &&
      if(documentDirty);
```

Implementing such state dependent aspectual behavior with help of multiple hooks or pointcuts (and their corresponding advice) is cumbersome, error-prone, and difficult to discern (particularly in retrospect). This is because the causal dependency between the different hooks and pointcuts (and their corresponding advice) is not obvious nor easy to detect. Close inspection of the code is necessary.

Novel approaches, such as presented in [2] and [1], promise to free programmers from the need to keep track of the system state manually; moreover, their (pointcut) notations allow to indicate the aspect's dependency on a particular system state explicitly in the program code.

This paper develops a notation that enables developers to visualize the state dependent selection of join points. By doing so, the notation aims to foster the use and specification of stateful aspects in MDSD.

3. STATE-BASED QUERY MODELS

JPDDs provide abstractions to specify queries on classes, their features and relationships, as well as on messages in a program's control flow. They lack dedicated support for the visualization of applicability conditions for stateful aspects, currently, though. In the following, a notation is developed that enables developers to visualize the state dependent selection of join points.

Starting point is a one-to-one visualization of the (conventional) hooks as specified in [2] (see Figure 1): The first hook designates the point in time beginning from which publications should occur. The second hook nominates those points that will actually cause the publication to occur (once the first hook has been passed).

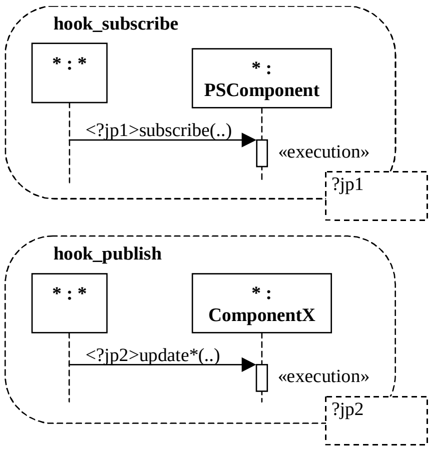

Figure 1: Two (Seemingly) Independent Join Point Designation Diagrams

As with the implementation (cf. [2]), the dependency between the hooks (in the sense that the latter hook will lead to effects only if the former hook has been passed) is not visible.

Figure 2 improves over this situation by merging both representations of Figure 1 into a single representation. That resulting diagram reflects on the temporal dependency between the occurrences of the hooks: It selects the second hook only if the first hook has occurred before.

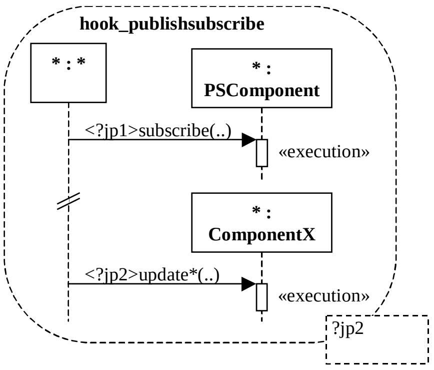

Figure 2: Representing Temporal Dependencies Between Hooks

While this representation efficiently visualizes the dependency between the hooks, it cannot be considered satisfactory from a modeling perspective. This is because what we see is two *messages* sent from any object to instances of PSComponent and ComponentX, respectively. What we are actually dealing with is a *stateful aspect*, though. No indication is made concerning the state the system (or the involved objects) are in.

Figure 3 makes explicit what is hidden in the diagram shown in Figure 2: The first message actually indicates a state change from state "silent" to state "publish" (at least seen from the perspective of the aspect). Once in that state, any occurrence of the second message (causing a self-transition) should be intercepted.

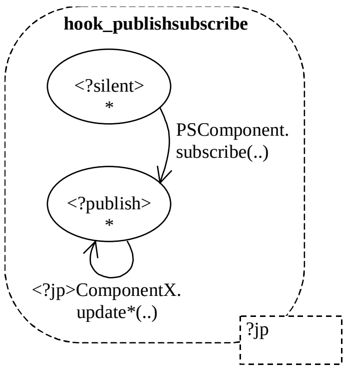

Figure 3: State-Based Selection of Hooks

The state-chart based visualization of the join point selection improves over the interaction-diagram based visualization since it emphasizes the dependency of the selection on the current system state. That way it helps to identify the corresponding aspect as a *stateful aspect*. Using the novel implementation techniques presented in [2], the representation can be furthermore mapped one-to-one to a corresponding aspect implementation.

```
8. hook PublishSubscribe {
      PublishSubscribe(subscribe(..args),
                       topublish(..args)) {
         Waiting: execute(subscribe) > Publish;
         Publish: execute(topublish) > Publish;
      }                [...] }
```

```
9. StatefulPublishManager.PublishSubscribe ps =
      new StatefulPublishManager.PublishSubscribe(
         boolean PSComponent.subscribe(),
         void ComponentX.update*(*) );
```

4. ANOTHER EXAMPLE...

Another example is taken from [1] and visualized in Figure 4: Three pointcuts are shown; the first one designates points at which some "dirty" flag should be set; the second one designates points at which that "dirty" flag should be unset; and the third one designates all points at which the aspectual behavior is actually to be performed (in case the "dirty" flag is set).

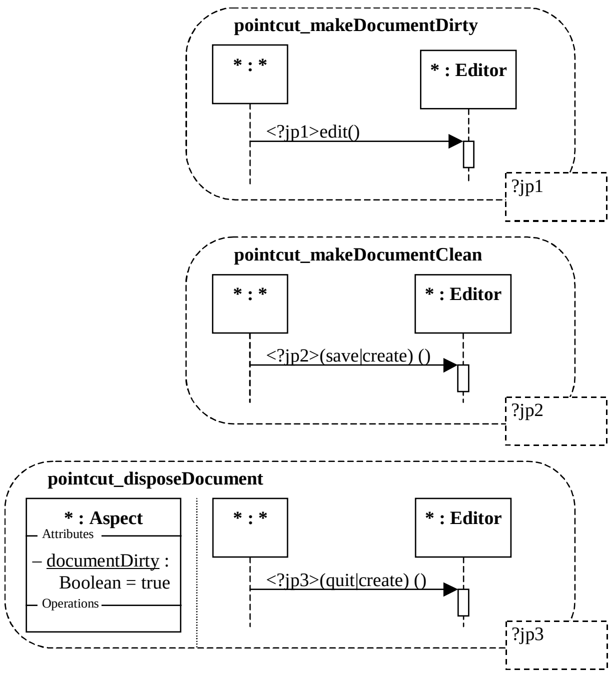

Figure 4: Three (Seemingly) Independent Join Point Designation Diagrams

Similar to the first case, the single specification of each pointcut does not reveal their temporal inter-dependencies. Figure 5 does a better job in this respect as it explicates in which order the selected join points must (and must not) occur.

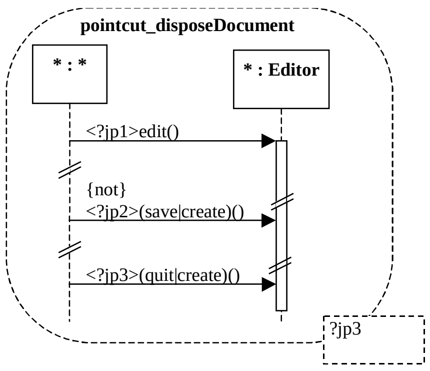

Figure 5: Representing Temporal Dependencies Between Join Points

However, even though the visualization is close to the corresponding program code using the implementation approach presented in [1] (cf. [1]), it does not contemplate on the fact that we are dealing with a *stateful aspect*. Instead, it reflects on the *communication* with an Editor instance and hooks onto messages being sent to that instance. From a modeling perspective, this is not satisfactory.

```
10. pointcut disposeDirtyDocument():
      (call(void Editor.quit()) ||
         call(void Editor.create())) &&
      afterend(call(void Editor.edit()) &&
      beforeend(call(void Editor.create()) ||
         call(void Editor.save()));
```

Figure 6 shows a state-chart based representation in which the method invocations are interpreted as trigger events to state transitions between states "clean" and "dirty". This representation helps developers to reason on the selection of join points depending on the *state* the system (the document/the aspect) is currently in.

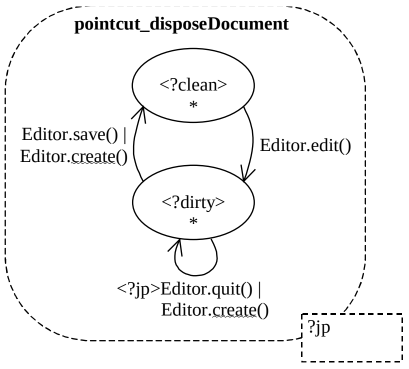

Figure 6: State-Based Selection of Join Points

In doing so, the representation reveals a (supposed) imprecision in the implementation (cf. [1]): Once in state "dirty", an invocation to method "create" could escape its interception and, instead, trigger a state transition to state "clean". To prevent this, the actual implementation relies on the internals of AspectJ, which executes multiple pieces of advice – if applied to the same join point – according to the order they are specified.

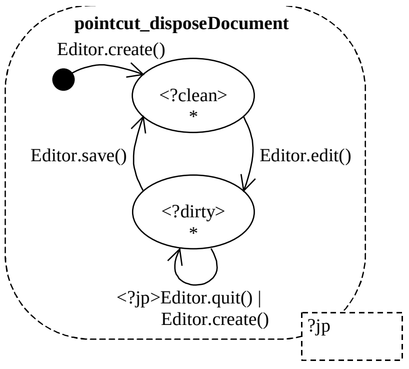

Figure 7: State-Based Selection of Join Points

We think that models, such as in Figure 7, may help to clarify such complications by explicating what is actually going on – for sake of the comprehensibility to both the new and unaware as well as the experienced yet forgetful software developer.

5. REFERENCES

[1] Bockisch, Chr., Mezini, M., Ostermann, K., *Quantifying over Dynamic Properties of Program Execution*, DAW Workshop, at: AOSD 2005, March 2005, Chicago, IL

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

[3] Douence, R., Fradet, P., Südholt, M., *Composition, Reuse and Interaction Analysis of Stateful Aspects*, AOSD'05, Lancaster, UK, March 2004, ACM, pp. 141-150

[4] *Dynamic Aspects Workshop*, March 15, 2005, AOSD'05, Chicago, IL, http://aosd.net/2005/workshops/daw/

[5] Gybels, K., Brichau, J., *Arranging language features for more robust pattern-based crosscuts*, in: Proc. of AOSD'03, March 17-21, 2003, Boston, MA, ACM, pp. 60-69

[6] Hanenberg, S., Hirschfeld, R., Unland, R., *Morphing Aspects: Imcompletely Woven Aspects and Continous Weaving*, in: Proc. of AOSD '04 (Lancaster, UK, March 2004), ACM, pp. 46-55

[7] 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

[8] Hanenberg, S., Unland, R., *Parametric Introductions*, in Proc. of AOSD 2003, March 17-21, 2003, Boston, Massachusetts, ACM, pp. 80-89

[9] 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

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

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

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

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

[14] Suvée, D., Vanderperren, W., Jonckers, V., *JAsCo: An Aspect-Oriented Approach Tailored for Component-Based Software Development*, in: Proc. of AOSD'03, March 2003, Boston, USA, ACM