# Why Aspect-Oriented Software Development and Model-Driven Development Are Not The Same – A Position Paper –

Dominik Stein and Stefan Hanenberg

Institute for Computer Science and Business Information Systems (ICB)

University of Duisburg-Essen, Germany

{dstein | shanenbe }@cs.uni-essen.de

## ABSTRACT

Aspect-Oriented Software Development (AOSD) and Model-Driven Development (MDD) are often said to be alike since both approaches are based on the selection of elements (i.e. join points in AOSD and model elements in MDD) and their subsequent adaptation (i.e. weaving in AOSD and transformation in MDD). But does this mean that AOSD and MDD are in fact two words for pretty much the same thing? In this position paper, we argue that there are essential differences between the aspect-oriented and the model-driven approach.

## 1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) and Model-Driven Development (MDD) are both concerned with the adaptation of an input system in order to receive an augmented/modified output system. In aspect-oriented literature, this process is referred to as *weaving*, while in the model-driven domain, this process is referred to as *transformation*.

From an abstract point of view, it looks like there are no significant differences between both approaches – except maybe that both approaches use a different terminology for the same conceptual idea: The adaptation of developer-specified elements. A close look to both approaches reveals, though, that they focus on different domains.

Aspect-oriented literature often refers to the term *separation of concerns* (cf. e.g. [2]). Following that idea, elements in the program *code* (implemented with a particular programming language) should always reflect on just one certain concern that the developer has in mind. Aspect-oriented extensions attach additional concerns to that code – concerns that do not comply with the primary concern, or the *dominant concern*, which has dictated a *dominant decomposition* [10] onto the program.

The focus in the model-driven world is slightly different. Model-driven development has a *model* (of a piece of software) in mind – a (generally) non-turing complete programming language, which possibly represents just a part of an application. The underlying intention for applying a model transformation is the creation of *machine-readable models that can be understood by automatic tools that generate schemas, code skeletons, testing models, test packs, and integration code for multiple platforms and technologies*[1].

[1] http://www.xpdian.com/ModelDrivenArchitecture.html

Hence, the goals and objectives of both technologies are most different. Nevertheless, it is still unclear if their underlying techniques are the same and whether both approaches can be considered equal.

In this position paper, we start in section 2 with an example from the aspect-oriented literature and discuss the aspect-oriented elements being used within this example. In section 3, we discuss how the corresponding example can be implemented using a model-driven approach. Then, we discuss the parallels and differences between the aspect-oriented approach and the model-driven approach. In section 4, we formulate our position by stating from the aspect-oriented point of view why “aspect-orientation is more than model-driven development” and by stating from the model-driven point of view “why model-driven development is more than aspect-orientation”. Finally, we conclude our position in section 5.

## 2. PERSONAL INFORMATION MANAGEMENT – AN ASPECT-ORIENTED EXAMPLE FROM THE LITERATURE

In order to exemplify the parallels and differences of aspect-oriented software development and model driven development, we make use of a simple example. The example has been inspired by [1] and realizes an access control policy for a Personal Information Management (PIM) system. The PIM system is intended to keep track of personal information, such as addresses, tasks, and daily assignments.

Figure 1 gives an overview to the core entities of the PIM system. The (singleton) `PIMSystem` is the general broker class that is used by a (singleton) `Person` to administer his/her various `PIMUnit`s, such as `Task`s, `Contact`s, and `Appointment`s. It is assumed that the system is designed for single-user usage only and does not implement any access control mechanisms. These are to be added to the system now by means of aspects in order to allow multiple-user usage.

### 2.1 Owner Management

The aspect-oriented solution (exemplified by a corresponding AspectJ implementation) realizes the owner management by an `OwnerManagement` aspect (based on a similar implementation given in [1]). This implementation is realized in AspectJ [7] and is illustrated in Figure 2: The aspect implements a couple of introductions, three of which augment the `PIMSystem` class with a `login` operation and an additional `currentUser` state –

together with a corresponding getter method. The other three introductions augment the PIMUnit class with an extra owner state – again, together with corresponding getter and setter methods.

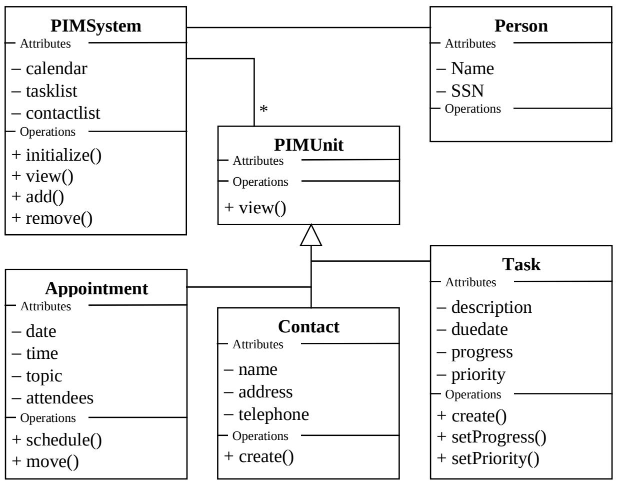


**Figure 1: A Sample Base System (cf. [1]).**

The introduced members are used by the after advice, which takes care of storing the currently logged on user (PIMSystem.getCurrentUser) to the owner attribute of a PIMUnit (PIMUnit.setOwner). Before doing so, it tests if the current user is already set (i.e. not null). In such a case, it asks the current user to login (by calling operation login of class PIMSystem).

```
aspect OwnerManagement(){
 private static String PIMSystem.currentUser ;
 public  static String PIMSystem.getCurrentUser() {...} ;
 public  static void   PIMSystem.login() {...} ;
 private String PIMUnit.owner ;
 public  String PIMUnit.getOwner() {...} ;
 public  void   PIMUnit.setOwner(String user) {...} ;

 pointcut authentifyUnit(PIMUnit pimUnit):
  (call(* Appointment.schedule(..)) ||
  call(* Contact.create(..)) ||
  call(* Task.create(..))) && target(pimUnit) ;

 after(PIMUnit pimUnit) : authentifyUnit(pimUnit) {
  if (PIMSystem.getCurrentUser() == null)
     { PIMSystem.login() ; } ;
  pimUnit.setOwner(PIMSystem.getCurrentUser()) ;
 }
}
```

**Figure 2: An Aspect-Oriented Implementation.**

The advice refers to the pointcut authentifyUnit that outlines the points in the execution of the program where a current user needs to be stored to the owner attribute of a PIMUnit. In particular, these are all method calls to operation schedule of class Appointment, to operation create of class Contact, and to operation create of class Task. At last, the instance of PIMUnit being called is exposed by the pointcut by means of AspectJ's target pointcut designator.

When taking a closer look to pointcut authentifyUnit, we can observe that all join points at which the after advice needs to be executed correspond to certain elements in the code (i.e. method calls to Appointment.schedule, Contact.create, and Task.create). In consequence, the adaptation (weaving) of the base system can be accomplished by simply inserting the core advice code to the places designated by the pointcut. These can be detected by a simple code analysis of the base classes and base methods. No problem so far!

### 2.2 Access Control

Now, we want to use the owner management data to ensure that particular PIMUnits (i.e. tasks, contacts, or appointments) may only be modified or viewed by their proper owner.

Thereto, an Authorization aspect defines a pointcut restrictAccess that picks out all invocations to methods whose access needs to be controlled. In particular these are all method calls to move operations of Appointment instances as well as all method calls to setProgress and setPriority operations of Task instances.

Furthermore, the pointcut makes use of the target pointcut designator to get a reference to the actual instance of the PIMUnit (i.e. or of its subclasses) being called. It uses this reference to verify if the owner of that (target) instance matches the user that is currently logged on the PIMSystem. To do so, AspectJ's if pointcut designator is used.

```
aspect Authorization(){
 pointcut restrictAccess(PIMUnit pimUnit):
  (call(* Appointment.move(..)) ||
  call(* Task.setProgress(..)) ||
  call(* Task.setPriority(..))) && target(pimUnit) &&
  if(!pimUnit.getOwner().equals(PIMSystem.getCurrentUser())
     || (PIMSystem.getCurrentUser()== null));

 void around(PIMUnit pimUnit) : restrictAccess(pimUnit) {
  System.out.println("Access Denied!") ;
 }
}
```

**Figure 3: An Aspect-Oriented Implementation**

Looking at the pointcut of this aspect, we can identify that the join points at which the around advice needs to be executed are (again) outlined by the characteristics of code elements (i.e. by the occurrence of method call statements Appointment.move, Task.setProgress and Task.setPriority). However, apart from that, the pointcut restrictAccess refers to the values of attribute owner of class PIMUnit as well as of attribute currentUser of class PIMSystem in its if pointcut designator (in order to evaluate if they match). These values are not known until runtime. Hence, in contrast to the previous adaptation, this one here cannot be effectuated until runtime.

## 3. PERSONAL INFORMATION MANAGEMENT – ATTEMPTING A MODEL-DRIVEN APPROACH

Let's have a look at how we could realize the Personal Information Management (PIM) example from the previous section with help of MDD.

### 3.1 Owner Management

Figure 4 outlines how the structural adaptations of the owner management aspect can be specified. The upper part ownerManagement_lhs depicts a *model query* (using the notational means presented in [9]). The model query selects all classes named PIMSystem and PIMUnit, and exposes them with help of identifier ?pimSys and ?pimUnit in its output parameter box (see lower right corner).

The lower part ownerManagement_rhs of Figure 4 depicts the affiliated model transformation which is to be performed at those selected model elements (the representation resembles pretty much conventional UML templates, except that the parameters ?pimSys and ?pimUnit are depicted differently). According to that adaptation specification diagram, the model elements exposed by ?pimSys are enhanced with a static and private attribute currentUser of type String, as well as two static and public operations login (returning nothing) and getCurrentUser (returning a value of type String). The model elements designated by ?pimUnit are augmented with a private attribute owner of type String, as well as two public operations getOwner (returning a value of type String) and setOwner (taking an argument of type String).

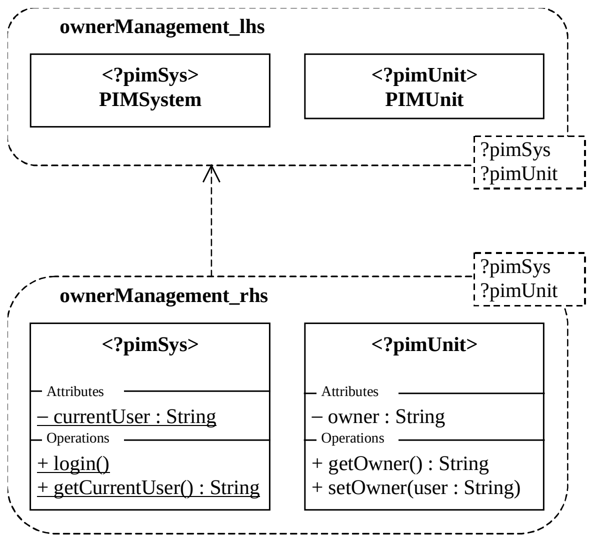


**Figure 4: Transformation of the Base Program's Structure.**

Figure 5 depicts the behavioral adaptations that are needed to realized the owner management aspect. The upper part pointcut_authentifyUnit, again, outlines a model query. It selects all method calls to operations named create or schedule, which are invoked on classes named Appointment, Contact, or Task. Both, method calls and corresponding receiver classes, are exposed by means of identifiers ?jp and ?pimUnit.

These model elements are then transformed as outlined in the bottom part afterAdvice_storeOwner of Figure 5. For that transformation, the method call ?jp is cut into two halves: refers to (and abstracts from) the sender class and the invocation of the method call; refers to (and abstracts from) the receiver class as well as the action being invoked. These two halves are arranged in such way that the original invocation is intercepted and redirected to class ?pimUnit[2], which then executes the original action. After that, the current user is stored to the owner attribute of the ?pimUnit (setOwner) – unless it is undefined (i.e. null) in which case the login operation of class PIMSystem is called. Finally, the control flow returns to the next action after the intercepted original method call ( ).

Taking a closer look at the OwnerManagement aspect presented in the previous section and the two model transformations presented here, we can observe that both adaptations are equivalent – the semantics of the resulting application are the same. In fact, the pointcut specification in the aspect-oriented approach corresponds directly to the model query in the model-driven approach. Also, the advice (which represents the join point adaptation in the aspect-oriented approach) corresponds to the transformation as specified in the model-driven approach. The only directly observable difference between both approaches is that the model-driven approach provides a visual representation of the selection and adaptation, while the corresponding aspect-oriented approach relies on plain code.

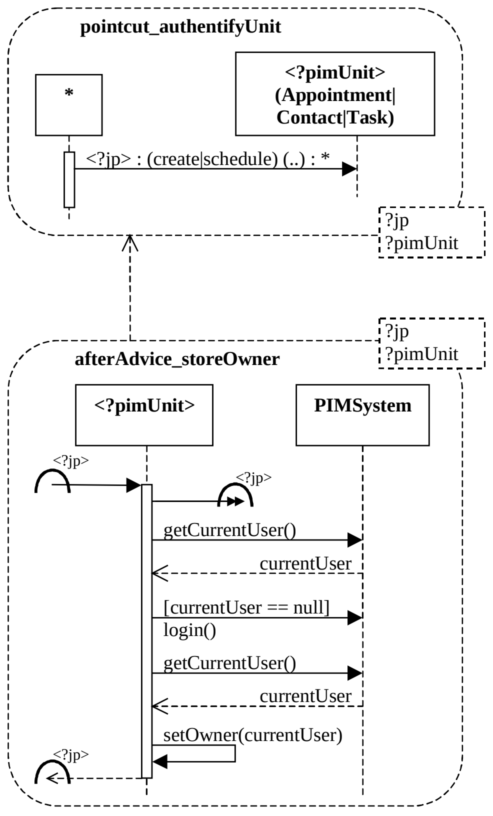


**Figure 5: Transformation of the Base Program's Behavior.**

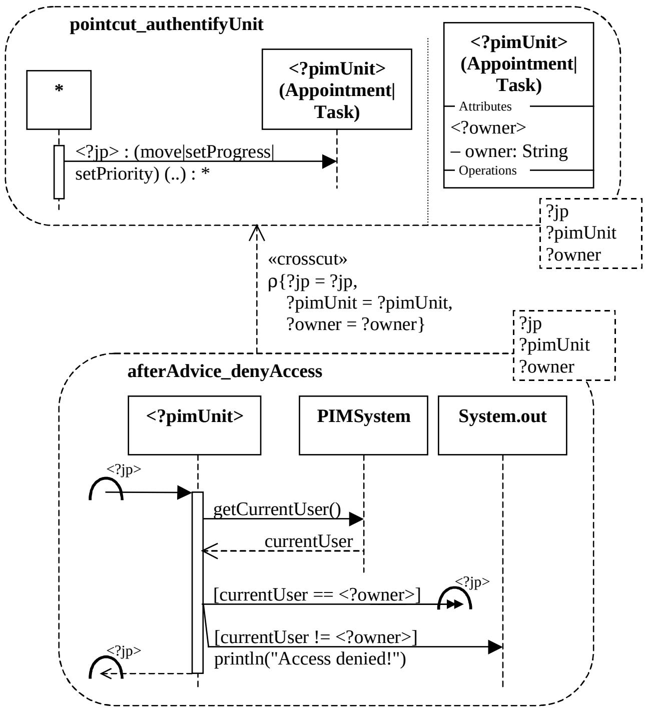
## 3.2 Access Control

When realizing the Authorization aspect with help of MDD, the core task is to evaluate if the owner of the PIMUnit matches the currentUser of the PIMSystem.

The bottom part of Figure 6 demonstrates a common way how this is done in MDD: First of all – i.e. right after the base program's behavior is intercepted at ?jp ( ) – the currentUser needs to be requested from the (singleton) PIMSystem class with help of operation getCurrentUser.

2 strictly speaking, there is no "redirect" in this case since ?pimUnit (by accident) refers to the receiver class of the original action.

Then, the `currentUser` is compared to the `?owner` attribute of the current `?pimUnit`. If it is alike, the program execution should proceed with the originally intercepted method call `?jp` ( ). If the values do *not* match, though, the message "Access denied!" is printed to `System.out` and the control flow is passed back again to the original caller ( ).



pointcut_authentifyUnit
*
<?pimUnit> (Appointment|Task)
<?pimUnit> (Appointment|Task)
Attributes
<?owner>
– owner: String
Operations
<?jp> : (move|setProgress|setPriority) (..) : *
?jp
?pimUnit
?owner
«crosscut»
ρ{?jp = ?jp, ?pimUnit = ?pimUnit, ?owner = ?owner}
?jp
?pimUnit
?owner
afterAdvice_denyAccess
<?pimUnit>
PIMSystem
System.out
<?jp>
getCurrentUser()
currentUser
[currentUser == <?owner>]
<?jp>
[currentUser != <?owner>]
println("Access denied!")
<?jp>

**Figure 6: Transformation of the Base Program's Behavior on the Specification Level**

As can be seen from the query model at the top of Figure 6, these adaptations are applied to all method calls to operations named `move`, `setProgress`, or `setPriority` that are invoked on class `Appointment` or `Task`. At the same time, the receiver class `?pimUnit` must provide a private `owner` attribute of type `String` (as this attribute needs to be compared to the `currentUser` in the adaptation).

Comparing the model transformation presented here with the `Authorization` aspect in the previous section, we can identify a subtle yet important difference: While in the aspect-oriented approach, the condition whether `currentUser` (of `PIMSystem`) matches `owner` (of `PIMUnit`) is specified within the join point selection, in the model-driven approach, this condition is specified within the join point adaptation. The semantic implications of either solution are identical – nevertheless, we must recognize that in the latter approach the constraints under which access to the `PIMUnits` should be denied are less obvious (as they are scattered across join point selection and join point adaptation rather than being nicely encapsulated in the join point selection).

## 4. WHY ASPECT-ORIENTATION AND MODEL-DRIVEN DEVELOPMENT ARE NOT THE SAME

Based on the examples presented in the previous sections, we now argue why and where we see parallels and differences between AOSD and MDD.

### 4.1 Parallels Between Aspect-Orientation And Model-Driven Development

As demonstrated by the owner management example, parallels between AOSD and MDD certainly exist. The implementations of this concern with either technology are almost equal. In particular, the conceptual distinction between query and adaptation are in both approaches the same. Accordingly, the conceptual models used by the developer (i.e. the selection of method calls and the subsequent adaptation of those method calls) are identical.

Nevertheless, it should be noted that this conceptual similarity results from a very specific application of the aspect-oriented approach: The selected join points from the execution of the program directly correspond to elements in the program code – so, no runtime-specific conditions (called *join point checks* [4] or *join point residues* [7]) are required. There are aspect-oriented systems that provide only those kinds of join points – these are systems with purely *static join point models* (cf. [5, 6]). In contrast to that, though, popular systems like AspectJ (also) provide a dynamic join point model which permits the specification of runtime checks within a join point selection.

Hence, only under the special circumstance that no runtime-specific condition needs to be checked (i.e. the aspectual adaptation refers only to specification-level join points), we consider the aspect-oriented approach and the model-driven approach to be equivalent.

### 4.2 Why AOSD Dominates The MDD Approach – An Aspect-Oriented Perspective

Of course, (as we have shown) runtime-dependent adaptations can (always) be transformed into specification-level adaptations such that the effects on the behavior of the final software system are the same. In fact, this is what aspect-oriented systems commonly do when they perform code transformations in order to weave in aspects to the base system (currently, most systems like AspectJ do weaving of aspects via code transformations): They identify places in the base system that potentially represent a join point (such places are called *join point shadows* in aspect-oriented literature [8]), and instrument them with *join point checks* that evaluate whether the runtime-dependent condition hold or not.

Aspect-oriented systems equipped with a *dynamic join point model* (cf. [5, 6]) provide special *abstractions* that permit to designate such runtime-level join points. In doing so, they are freeing the developers from the need to reflect on join point shadows and necessary join point checks themselves. Examples of such abstractions are the dynamic pointcut designators *this*, *target*, *args* and *if* in AspectJ, which permit to declare that a certain runtime specific condition needs to be fulfilled (at a particular join point shadow) for the aspectual adaptation to take effect. The situation for the MDD approach is different. In MDD, the developers are required to insert such join point checks "manually" – which means that they need to be specified as part of the adaptation. In consequence, the approach forces developers to separate the applicability constraints of an aspect (e.g. for the denial of an action) into the locations in the models where the aspect needs to take effect, as well as the residue (i.e. the join point check) that remains to be evaluated at that location.

Another consequence of this approach in comparison to the AOSD approach is that selections and adaptations are less reusable. In the aspect-oriented approach, an adaptation is independent of the possible runtime checks that need to be performed before it takes effect (since these are specified in the selection). Selections may evolve or may be overridden in subaspects. Nevertheless, the affiliated adaptations do not need to be changed. In the MDD approach, though, each selection that conceptually requires a corresponding runtime check also requires its own adaptation module (since it is necessary to consider the runtime conditions within the adaptation). Consequently, if the application evolves and new selections are needed that require additional runtime checks, the adaptation modules may need to be adapted.

Hence, from the aspect-oriented perspective, the aspect-oriented approach dominates the model-driven one because additional abstractions for the join point selection are provided, which allow an (almost) arbitrary combination of join point selections and join point adaptations.

### 4.3 Why MDD Dominates The AOSD Approach – A Model-Driven Perspective

It remains to mention in what respect MDD approaches improve over AOSD techniques, i.e. in what respect the model-driven approach dominates the aspect-oriented approach:

This pertains mostly to the capabilities of the adaptation means that can be applied to base applications. These are quite limited in the aspect-oriented approach. Most commonly, there are various *constructive adaptation mechanisms* (like before and after advice, as well as introductions in AspectJ) that permit to add some additional elements to a base application (i.e. the invocation of aspect-specific code, or the addition of aspect-specific structure, respectively). However, there are only limited means to specify *destructive adaptations*[3] (such as around advice in AspectJ that do not refer to the original join point by means of `proceed`).

The model-driven approach, in contrast to that though, principally permits to perform arbitrary transformations on a source model – in particular, it allows unlimited support for the removal of elements. By these means, the MDD approach could be used, for example, to implement an arbitrary *refactoring* [3] on the source model, by performing the corresponding behavior-preserving transformations. Aspect-oriented approaches like AspectJ do not support such transformations since it is not possible to remove a single method from a source program.

## 5. CONCLUSION

In this position paper, we argued why the aspect-oriented approach and the model-driven approach are not equal. We illustrated our argumentation by an example from the aspect-oriented literature and showed that it is not always possible to achieve the same result in an *appropriate way* with help of the means provided by model-driven development approaches. The main argumentation for this is that the aspect-oriented approach provides additional abstractions that permit to specify runtime-conditions within join point selections.

However, we also argued that the means to adapt a base application with aspect-oriented constructs are quite limited. Simple transformations like performing a rename method refactoring, for example, cannot be achieved via aspect-oriented techniques – but with model-driven techniques.

From our point of view, it is essential for further research on both approaches that they cannot be considered to be generally the same thing. This implies that it might be interesting to see in future whether one approach can benefit from the other, e.g. by providing more advanced transformation techniques to AOSD, or by introducing more advanced selection means to MDD.

[3] See [5, 6] for a discussion of the terms constructive and destructive adaptation.

## REFERENCES

[1] De Win, B., Joosen, W., Piessens, F., *Developing Secure Applications Through Aspect-Oriented Programming*, in: Filman, R., Elrad, T., Clarke, S., Aksit, M., Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005.

[2] Filman, R. E.; Elrad, T.; Clarke, S.; Aksit, M. (Eds.): *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[3] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W. F.; Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[4] Hanenberg, S.; Hirschfeld, R.; Unland, R.: *Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving*, 3rd International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, England, March, ACM Press, 2004, pp. 46-55.

[5] Hanenberg, S.; Stein, D.; Unland, R.: *Eine Taxonomie für aspektorientierte Systeme*, In: Liggesmeyer, P.; Pohl. K.; Goedicke. M. (Eds.): Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, March, Essen. Lecture Notes in Informatics 64, GI, 2005, pp. 167-178.

[6] Hanenberg, S.; Stein, D.; Unland, R.: *Roles From an Aspect-Oriented Perspective*, Views, Aspects and Roles Workshop, ECOOP 2005, Glasgow, UK, July 25, 2005.

[7] Hilsdale, E.; Hugunin, J.: *Advice Weaving in AspectJ*, 3rd International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, England, March, ACM Press, 2004, pp. 26-35.

[8] Masuhara, H.; Kiczales, G.; Dutchyn, C.: *A Compilation and Optimization Model for Aspect-Oriented Programs*, Proceedings of Compiler Construction (CC2003), LNCS 2622, Springer-Verlag, 2003, pp.46-60.

[9] Stein, D.; Hanenberg, S.; Unland, R.: *Query Models*, 7th International Conference on the Unified Modeling Language (UML 2004), Lisbon, Portugal, October 11-15, 2004, Springer, LNCS 3273, pp. 98-112.

[10] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S. M.: *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, In 21st International Conference on Software Engineering (ICSE), 1999, pp. 107–119.